Programming and Computer Software, Vol. 31, No. 1, 2005, pp. 20–28. Translated from Programmirovanie, Vol. 31, No. 1, 2005.
Original Russian Text Copyright © 2005 by Kalinov, Karganov, Khorenko.

An Approach to the Development of Debuggers That Use Semantics of Constructs of Parallel Programs

A. Ya. Kalinov, K. A. Karganov, and K. V. Khorenko

Institute for System Programming, Russian Academy of Sciences,
ul. Bol'shaya Kommunisticheskaya 25, Moscow, 109004 Russia
e-mails: ka@ispras.ru; kostik@ispras.ru; kostja@ispras.ru

Received May 27, 2004

Abstract—In the paper, a new approach to the development of interactive debuggers for message-passing programs is suggested. The basic idea of the approach is to design a debugger specific to a particular language or a parallel programming library and to use information about the semantics of constructs used in the parallel program for processing commands of the step-by-step execution and data representation. The development of the user interface and internal debugger structure, as well as their implementations in the debuggers for mpC programs and programs using the MPI library, are considered.

1. INTRODUCTION

In spite of the rapid development of the hardware for parallel computing during last years, the software tools and software development environments for parallel programs are developing much slower. In view of high complexity of parallel programs, the problem of their efficient debugging is very important. As noted in [1, 2], the existing parallel debuggers do not fully satisfy the users' needs.

The existing debuggers for parallel programs are either universal (like TotalView [3], Prism [4], or DDT [5]) and do not use any program-specific information (for example, in the case of the MPI [6], do not display groups and communicators), or are special-purpose tools, like XMPI [7], that do not ensure efficient debugging of parallel applications.

In this paper, we consider debugging of programs for systems with distributed memory, which are based on the message-passing model. For the debugging method, the interactive debugging has been selected as the simplest and commonly accepted method.

Using the experience accumulated in the development of the parallel programming environment mpC Workshop [8, 9], which includes an interactive debugger for parallel programs in mpC [10, 11], we suggest a new approach to the design of interactive debuggers for parallel applications.

The basic idea of the approach suggested is to sacrifice debugger universality for improved convenience and the efficiency of the debugging process. The debugger should be a specialized tool designed for debugging programs from a particular class (for example, for programs that use the MPI library) and should use information about the semantics of the constructs used in the program for improving the efficiency of the data representation and controlling the processes of the program being debugged.

The paper is organized as follows. In Section 2, main problems arising in the development of an interactive debugger for parallel programs are outlined. Section 3 gives a brief overview of the most advanced interactive debuggers. The suggested approach to the construction of the user interface and organization of parallel step-by-step execution of the program processes is described in Sections 4 and 5. Section 6 describes an implementation of the suggested principles, the mpC Workshop debugger. Section 7 discusses the implementation of the suggested approach in the MPI debugger.

2. PARALLEL DEBUGGING PROBLEMS

From the standpoint of the interactive debugging, the most apparent difference between sequential and parallel programs consists in the irreproducibility of traces of the parallel program execution in successive program runs. This is a well-studied problem (see, e.g., [1, 12]), which can be solved by writing down the sequence of the events during the first run and reproducing the trace in the course of the debugging. In this paper, we assume that this mechanism can be implemented independently of others and will not discuss this problem.

Another group of problems is associated with the organization of the user interface, i.e., the representation of data about the states of the program and control of the execution of parallel processes. As discussed in [13, 14], the requirements for the user interface of a parallel debugger are as follows:

• *Scalability.* A debugger should be capable of debugging both "small" parallel programs and those

0361-7688/05/3101-0020 © 2005 MAIK "Nauka/Interperiodica"

consisting of many processes. In practice, this means that the debugger can work with groups of processes as easily as with the individual processes of the program. The scalability of the debugger interface is required in the following two cases: when displaying large amount of data about the states of the program and when controlling many processes taking part in the program execution. Examples of the user interfaces with low and high scalability are those in the TotalView [3] and Prism [4, 15] debuggers, respectively.

• *Flexibility.* The user interface of a debugger should provide the user with convenient and flexible means for controlling information representation in order to get the required data in the desired form: it should contain filters (to display what is needed and to discard unnecessary things) and be capable of presenting information in different degrees of detail. The user interface should also be highly customizable.

• *Fast access to major functions.* The debugger's basic functions, as well as other frequently used operations, should be easily accessible to the user and require minimum efforts for their execution. Such operations, as a rule, are placed on the debugger toolbar and are associated with hot key combinations. However, in view of historical reasons, many operations in Prism [4] are accessible only by invoking the corresponding instructions from the debugger command line.

• *Source-level debugging.* The debugger should support the debugging in the same terms that are used in the program, i.e., at the same abstraction level as that in the program. For example, if the program is written in FORTRAN, there is no sense to display the assembler code, since the abstraction level of such a code is very different such that the code does not give the programmer any useful information. Similarly, when debugging an MPI program, there is no sense to show the internal implementation of functions from the MPI library: calls of the communication library should be base primitives in the step-by-step execution of the processes.

• *Use of information about objects of the program.* To facilitate the debugging process, the debugger should maximally use a priori information about the program to be debugged. For example, in the case of a program in mpC, the debugger should show the current partition of the computational space into nets and subnets; in the case of debugging an MPI program, the structure of groups and communicators must be presented. Many debuggers (for example, Prism and TotalView) have means for displaying message queues of the MPI library; however, only the XMPI debugger [7] is capable of displaying information about derived MPI data types, about communicators, and about the membership of processes in groups.

• *Integration of components.* Various components of the debugger should be integrated in a single debugging environment and support operations involving several components simultaneously. For example, the following operations should be supported: "show the group containing the specified process" (transition from the program text window to the group window) and "show the position of the given process in the code" (transition from the group window to the program text window).

The above requirements may seem trivial; however, none of the known debuggers meets all of them. In Section 4, we suggest an interface structure that meets these requirements.

Another, less evident, problem arising in the debugging of parallel programs is the consistency of the parallel stepping commands. If we treat the parallel step of the program as a set of concurrent steps of standalone processes, we face the problem of the step completion: in a parallel program, some processes may not complete the step if they are waiting for interactions with other processes. The "manual" processing of the communication dependences between the processes at each step is a rather cumbersome and annoying operation, which, however, can be automated, as described in Section 5.

3. SURVEY OF KNOWN DEBUGGERS

Let us consider the existing approaches to the organization of the interactive debugging of parallel programs on the examples of the advanced parallel debuggers TotalView [3], Prism [4], DDT [5], and XMPI [7].

TotalView.

This is a universal debugger supporting many languages and parallel programming libraries. It uses the most universal approach: each process is opened in a separate window. This makes it possible to display the maximum amount of information about a particular process but considerably complicates the debugging of applications consisting of many processes. As a result, the interface of the debugger possesses low scalability and poor flexibility.

The debugger can use some specific information about the program. For example, in the case of the debugging of an MPI program, TotalView allows the user to look through MPI message queues; when debugging an OpenMP program, it shows OpenMP threads and correctly places breakpoints on the set of threads if the specified line is located inside the parallel block.

The implementation of the parallel step command in TotalView makes use of the synchronous scheme: a step of the parallel program is assumed to be completed only when all processes taking part in the execution of the command accomplish the execution of the step. In the case of the blocking of the parallel program processes, it is possible to pause the execution; however, in this case, only the current state of the program blocked inside the communication function is displayed, which, most often, does not give the user sufficient information about the causes of the blocking, thus making it impossible to eliminate the cause of the blocking and to continue the execution of the program.

Prism.

This debugger was originally designed as a maximally scalable debugger with an interface similar to that in a sequential debugger [15]. It has an advanced command-line control language and a graphic interface; unfortunately, many debugger functions can be accessed only through the invocation of the commands of the internal debugger control language. The use of this approach opens wide opportunities for the automation and customization of the debugging process (for example, the creation of user macros and scripts); however, this makes the use of the debugger more complicated.

Prism introduces the new notion, *pset*, which denotes a group of processes that can be treated as a single entity. The user can define arbitrary groups and work with them instead of separate processes. The majority of the debugger commands (including the data display commands) can be applied to such groups. The set of processes included in *pset* may depend on the program state (for example, on values of variables) and change dynamically in the course of the debugging. This gives the user a very powerful and flexible method for controlling large groups of processes.

The debugger can also use some information about objects of the program. For example, when debugging programs that use Sun MPI, it is possible to look through MPI message queues with different degrees of detail and message-sorting modes; when debugging programs that use the scientific library Sun S3L, the visualization of distributed arrays is possible.

Unlike TotalView, Prism implements the asynchronous model of the parallel step: the step command is delivered to all processes that completed the previous step and control is immediately returned to the user. The determination of the moment of the step completion and the analysis of the resulting program state are done by the user manually. There is also a possibility to wait for the completion of the above-specified processes.

DDT.

This is a universal debugger with an advanced user interface. It supports groups of processes under the condition that the groups are created manually by specifying numbers of the desired processes in the parallel program. The information about the program state is displayed only for one selected process, except for the state "ready/blocked," which is displayed in color in the group window for all processes simultaneously.

The debugger does not support the representation of objects of the parallel program, except for standard means for representing variables, arrays, and call stacks of separate processes.

XMPI.

This debugger has been developed by the LAM MPI team; therefore, it supports the debugging of programs that use only this implementation of the MPI communication library. It does not support the majority of standard functions of an interactive debugger, such as step-by-step execution of program statements or setting up breakpoints, but uses the internal information about the communication library. XMPI makes it possible to visualize states of the processes, statistics of transmitted messages, message queues, and the processes belonging to a group or communicator. It also has means for visualizing derived MPI data types and for writing and visualizing the trace of the program execution.

4. SUGGESTED APPROACH TO THE INTERFACE ORGANIZATION

A natural way to simplify the debugging of parallel programs and reduce the time required for the user to become accustomed to the parallel debugger is to extend the traditional functionality of the sequential debugger to the case of parallel programs. The debugger is assumed to have a graphic user interface with a main window displaying the text of the debugged program and several windows for displaying additional information. Traditionally, the left border of the program text window is used for placing and displaying breakpoints and current positions of processes of the parallel program.

The simplest extension of the debugger functionality to parallel programs is to add windows displaying objects of the debugged program. For example, when debugging MPI programs, there should be windows that show groups, communicators, and other opaque MPI objects; when debugging programs in mpC, the debugger should have windows presenting computational networks created in the program. Note that the objects global for the entire program (e.g., communicators in MPI) should be displayed in a separate window, and the objects that are local for the processes should be displayed in a window that allows the user to specify the process to be displayed or to compare values on several processes.

4.1. Displaying Call Stacks

For displaying call stacks in a parallel program, the notion of a *call tree* can be used, which is similar to that in Prism [15]. The root of the tree is a call of the function `main()` (for programs in C) that is common for all processes of the program. The nodes of the next level are various functions called by the processes of the program from the functions of the previous level. For example, if all processes from `main()` call the same function, the root of the call tree has exactly one descendant.

Such a scheme guarantees the most efficient representation of the call stacks, since all "identical" calls are automatically glued together, so that the number of leaves in the tree is equal to the number of different stacks in the processes of the parallel program. If many processes execute the same code, the number of leaves

is considerably less than the total number of processes of the parallel program.

4.2. Structure of the Computing Space

An important element of the interface is a window showing the structure of the computing space (the partition of the set of all processes of the program into groups and subgroups), for example, the list of the created communicators in MPI or the hierarchy of network objects in mpC. First, it gives an idea of the current structure of the program computing space. Second, it displays information about the structure of each network object (processes included into an mpC communicator or an mpC network, their order and coordinates in the mpC network, attributes and topologies associated with the MPI communicator, etc.). Third, such a window provides a convenient interface for invoking operations on network objects (e.g., through a context menu).

As a rule, a communicator or mpC network object created in the program bears certain semantic information for the programmer: the processes in the communicator or network either solve a common problem or are combined on the basis of some other criterion. This assumption allows us to introduce various filters for displaying information and controlling program processes, which are associated with semantically meaningful groups of processes, and the integration of various debugger components makes it possible to execute these operations for a minimum number of actions.

4.3. Display of Variables

To display values of variables of the debugged program, the following scheme is used: the values are placed into a table whose columns correspond to the processes of the parallel program and rows correspond to scalar values of the variables (such a scheme is implemented in mpC Workshop [8], see Section 6). Such a data display method allows the user to simultaneously watch the values of variables on several processes, which considerably facilitates searching for errors resulting from incorrect values of variables on certain processes of the program.

To make such a display scalable, the concept of a data display filter is introduced: in the watch window, only a certain group of processes, rather than all of the program's processes, is displayed. The filter can be specified manually by listing the processes, or by a certain group of processes (see below), or by a communicator or network. Since a communicator (network) is a group of semantically related processes of the program, filtering values by this group is one of the most meaningful ways to restrict the amount of the displayed information.

4.4. Process Control

One of the key issues in the design of parallel debuggers is the control of processes of the parallel program. To control the processes, the following commands are used:

• continue program execution,

• make one step (with or without entering the functions),

• finish function,

• proceed to a specified position.

Note that the breakpoint and watchpoint hits are possible on different processes of the program.

In the step-by-step execution of the program statements, there are two information flows between the user and debugger, namely, the flow from the user to the debugger, which determines what groups of processes have to execute the step command, and the flow from the debugger to the user, which determines the resulting state of the specified processes after the execution of the step command (in particular, the current positions of the processes in the program text).

To control the processes, the concept of *pset* is introduced. Like in Prism, *pset*, or a *process set*, is a named group of processes that can be treated as a single entity for the debugger's commands. A group of processes can be specified as a static list of the process ranks or be defined by a Boolean expression (membership function), which may depend on other groups (through set-theoretic operations), on the state of the process, or its data. A group of processes can be re-evaluated at the user's request, which allows the user to create dynamic groups, depending on the process states. We suppose that an automated group re-evaluation (for example, after completing the step) can be implemented only as an additional option, since, in this case, the user, in fact, looses control over the process membership, which results in a confusion and debugging errors.

At the beginning of a debug session, there should be, at least, one predefined group of processes that includes all processes of the parallel program. Each group can be "active" or "inactive." The union of all active groups is referred to as a *control set*; the processes from this set perform the user's commands. Such a scheme provides a very flexible and scalable method for controlling the processes of the program being debugged: even if the program consists of many processes, the number of "roles" played by the processes is not great. Each role is made to correspond to a named group of processes and is controlled as one process. The possibility of selecting active and inactive groups makes it possible to perform fast set-theoretic addition and subtraction of groups of processes without creating new named groups.

Taking into account that groups of processes (psets) correspond to certain semantically unique groups of processes of the parallel program (for example, processes executing common computations in parallel), we

see that psets can also be used for creating data display filters and that the debugger should support a more close integration with the program being debugged. For example, it should permit the creation of groups of processes by MPI communicators or by mpC networks (the developers of Prism also arrived at this conclusion [16]; however, this functionality has not been implemented yet in any known debugger).

It follows from the above discussion that the debugger must have a separate window for the work with the groups of processes, which displays the list of the created groups and the processes belonging to them and has facilities for creating, deleting, and editing the existing groups. Each group should have an "active/inactive" checkbox and a means for re-evaluating groups of processes and creating display filters by the given group (e.g., through a context menu).

4.5. Presentation of Process States

To present states of the processes determined by the debugger after the execution of the step command (or after a stop caused by any other reason), the notion of a debug cursor is introduced. The state of the processes of the controlling set (i.e., the processes that have completed the command) is presented by means of a colored cursor pointing to the current position of the process in the program text. The green color means that the process has performed successfully the previous commands and is ready to perform the next ones; red means that the process has not completed the execution of the previous command, was blocked inside the communication function, and waits for interactions with other processes.

The yellow color is used to make the control of processes more flexible. It is set by the user for the process that is ready for the execution but should not perform the step commands. This third state provides a convenient way to hold some process from performing steps in the step-by-step debugging of the program. If several cursors of one color stand on the same line of the source code, they are grouped into one *compound cursor*, which also improves the scalability of the debugger.

The above-described scheme of the interaction of the user with the debugger may seem too complicated and excessive; nevertheless, it provides a very flexible and convenient way of controlling processes in the debugging of parallel programs. This scheme maximally uses information about internal objects of the program and makes it possible to perform the majority of frequently used operations in one action. Under certain assumptions about the debugged program, in particular, for a well-structured SPMD program and an appropriate set of groups of processes, the debugging of a parallel program becomes very similar to the debugging of an ordinary sequential program.

5. SUGGESTED APPROACH TO THE IMPLEMENTATION OF THE STEP COMMAND

When debugging a sequential program, the semantics of the step command is simple and clear: execute statements of the current line and break the program execution. The step may not be completed by virtue of some "sequential" reasons, such as, for example, infinite loops, long computation, or waiting for an exchange with the environment.

In the case of a parallel program, the parallel step can be defined as a set of concurrent sequential steps of a group of processes of the program. In addition to "sequential" causes, the step may be not accomplished because of "parallel causes," which are associated with the interactions of processes of the parallel program. For example, if a process executing a step is waiting for a message from another process that has already accomplished its step but has not reached the point where the corresponding message is to be sent, then the first process cannot accomplish its step during the current step of the parallel program.

As follows from the discussion of the debuggers in Section 3, there exist two basic schemes of the parallel step, synchronous (the step is considered completed when all processes accomplished the step) and asynchronous (the control is returned not waiting for the step completion) schemes. Each scheme has its own advantages: in the synchronous scheme, the moments of the beginning and completion of the step are clearly defined, the state of the program before and after each step is clear, and the current position of the debug cursor is always in the program code; in the asynchronous scheme, there are no delays, the debugger is not blocked in the case of the process blocking, and the debugger can work simultaneously with the execution of processes of the program being debugged.

Nevertheless, in both schemes, the analysis of the causes of the step incompleteness is to be done by the user. At the same time, the analysis of the majority of "parallel" causes of the parallel step incompleteness can be automated on the basis of the communication model of the program execution. In this case, the debugger analyzes interactions between the processes of the parallel program and, at each step, constructs the communication dependency graph for the processes.

If the completion of a step by one of the processes depends on the accomplishment of another process that has already completed its step or has been blocked because of its dependence on a third process, then the first process is also blocked on the current step. Such a mechanism makes it possible to automatically process the majority of "parallel" causes of the step incompleteness (excluding communications not provided in the program model) and find deadlocks in the program.

Using the above-described mechanism of the analysis of communication dependences of the processes, we can define the parallel step. Let us assume that, to com-

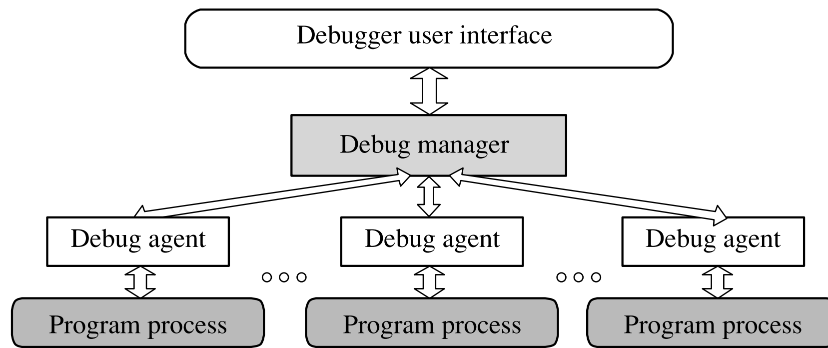

Fig. 1. Architecture of the mpC Workshop debugger.

plete the parallel step by a group of processes, each process from the group that accomplished the execution of the previous steps must execute the sequential step in its program. The parallel step is considered completed if each process of the program either (a) has completed the execution of its sequential step (which, possibly, was caused by one of the previous commands of the step-by-step execution) or (b) has been blocked when interacting with another process (or processes) and cannot continue the program execution until the user performs certain actions concerning other processes.

The problem of the command accomplishment is typical not only of the step command but also of other debugger commands associated with the execution of the statements of the program being debugged and can be solved by means of the same mechanism of the communication dependences analysis.

The suggested scheme of a “smart” parallel step is a natural extension of the semantics of the synchronous step to the case of interacting processes; it has all advantages of the synchronous scheme. Similar to the traditional synchronous step, the suggested scheme of the step execution does not terminate in the case of infinite loops or unexpected blocking on the external communication. In such cases, the debugger behaves like any debugger with the synchronous step scheme; however, in the case of the exchange by means of the message passing mechanism (which is the most typical case in parallel programs), the parallel step will be processed correctly.

6. IMPLEMENTATION OF THE SUGGESTED APPROACH IN THE MPC WORKSHOP

The above-discussed approaches to the organization of the user interface and the step-by-step program execution have been implemented in the debugger included in the integrated development environment (IDE) mpC Workshop [8, 9], which is designed for the development of parallel programs in the mpC language [10, 11].

The architecture of the mpC Workshop debugger is shown in Fig. 1. The debugger consists of two main parts: the debugger client (debugger user interface) and the debugger server, which, in turn, consists of two—distributed (a set of agents) and undistributed (debug manager)—parts. The user interface interacts with the server part of the debugger through a network, which allows the user to debug remote programs (including those in a different operating system), work with different servers from one desktop, and so on.

To implement the above-described concepts, the server part consists of the distributed and undistributed parts. The distributed part is a set of debug agents corresponding to the processes of the program being debugged. Each debug agent controls one process of the parallel program, performs commands of the debug manager, and returns information about the state of the process to the manager.

The undistributed part of the debugger server is a dedicated process of the debug manager, the purpose of which is to interact with the user interface, control the debug agents, and analyze communication dependences in the debugged program. The debug manager receives commands from the user interface, translates them into appropriate commands for the corresponding agents, and sends these commands to the required agents. The agents execute the required operations and send the results of the execution to the debug manager, which, in turn, collects the commands from the agents, analyzes their results, generates the response, and sends it to the user interface.

Note that the debug manager includes the communication model of the parallel program execution and, based on this model, analyzes communication dependences during the program run. Analyzing communication dependences between the processes and results of the command execution by different agents, the debug manager determines which processes are blocked waiting for interactions with other processes of the program and, thus, determines the moment when the parallel step is completed.

The main window of the integrated development environment of mpC Workshop is shown in Fig. 2. It displays the source code of the program; control objects, such as breakpoints; and the current positions

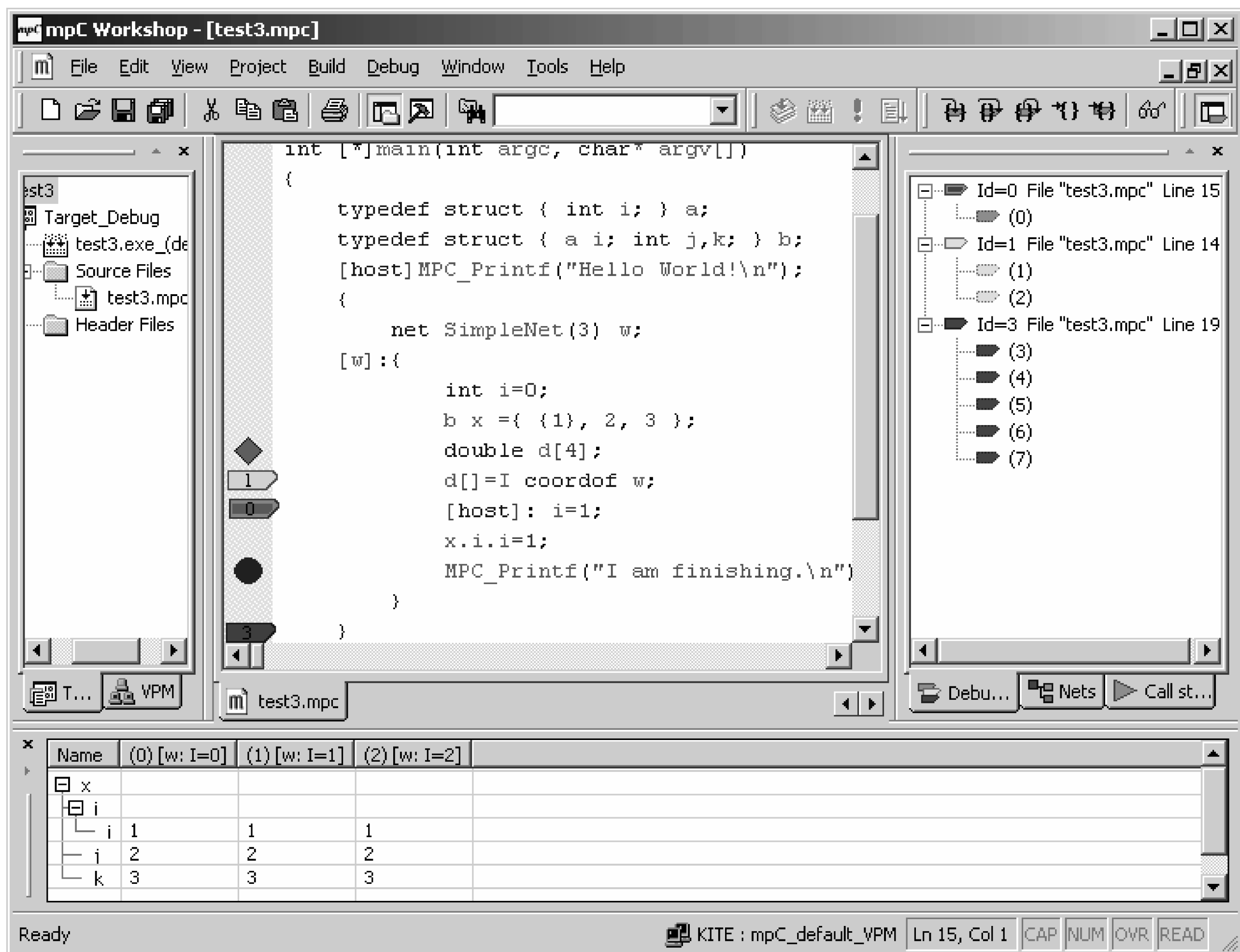

Fig. 2. The main window of the debugger (cursor window).

of the processes of the parallel program in the source text. Additional windows show project files, values of program variables, and other information.

To control the processes, a simplified scheme not supporting groups of processes has been used. Actually, there is only one group that contains all processes of the parallel program. The scheme implements the mechanism of color cursors showing the status of the processes, which was described in Section 4. For example, in Fig. 2, one can see three cursors: the green cursor in line 15 corresponding to one process (rank 0), the yellow cursor in line 14 corresponding to two processes (ranks 1 and 2), and the red cursor in line 19 corresponding to the remaining five processes of the parallel program. As can be seen from the figure, the cursors are identified by the minimal rank of the process associated with this cursor.

The right window in Fig. 2 is the debug cursor window; it shows all cursors, their colors, and positions in the program text. By means of this window, the user can divide or combine the green and yellow cursors standing in one position or change the cursor color from yellow to green and vice versa.

In the lower debugger window in Fig. 2, values of the program variables are shown. Here, we can see values of the structure *x* on the processes belonging to the network *w* (ranks 0, 1, and 2). The values corresponding to other processes have been blocked by the user filter. The same structure *x* without the filter is shown in the lower debugger window in Fig. 3. As can be seen, only the values corresponding to the processes of the mentioned network make sense, whereas the values of the structure fields on other processes cannot be computed.

The right window in Fig. 3 is the computing space hierarchy window. It shows the list of all processes of the program and the hierarchy of the created networks. In the given example, only one network *w* containing three processes with the one-dimensional coordinate system (the variable *I* taking values from 0 to 2) has been created. This window makes it possible to create filters by networks or lists of processes and applying these filters to various windows (display of variables, cursors, values of watchpoint expressions, and the like).

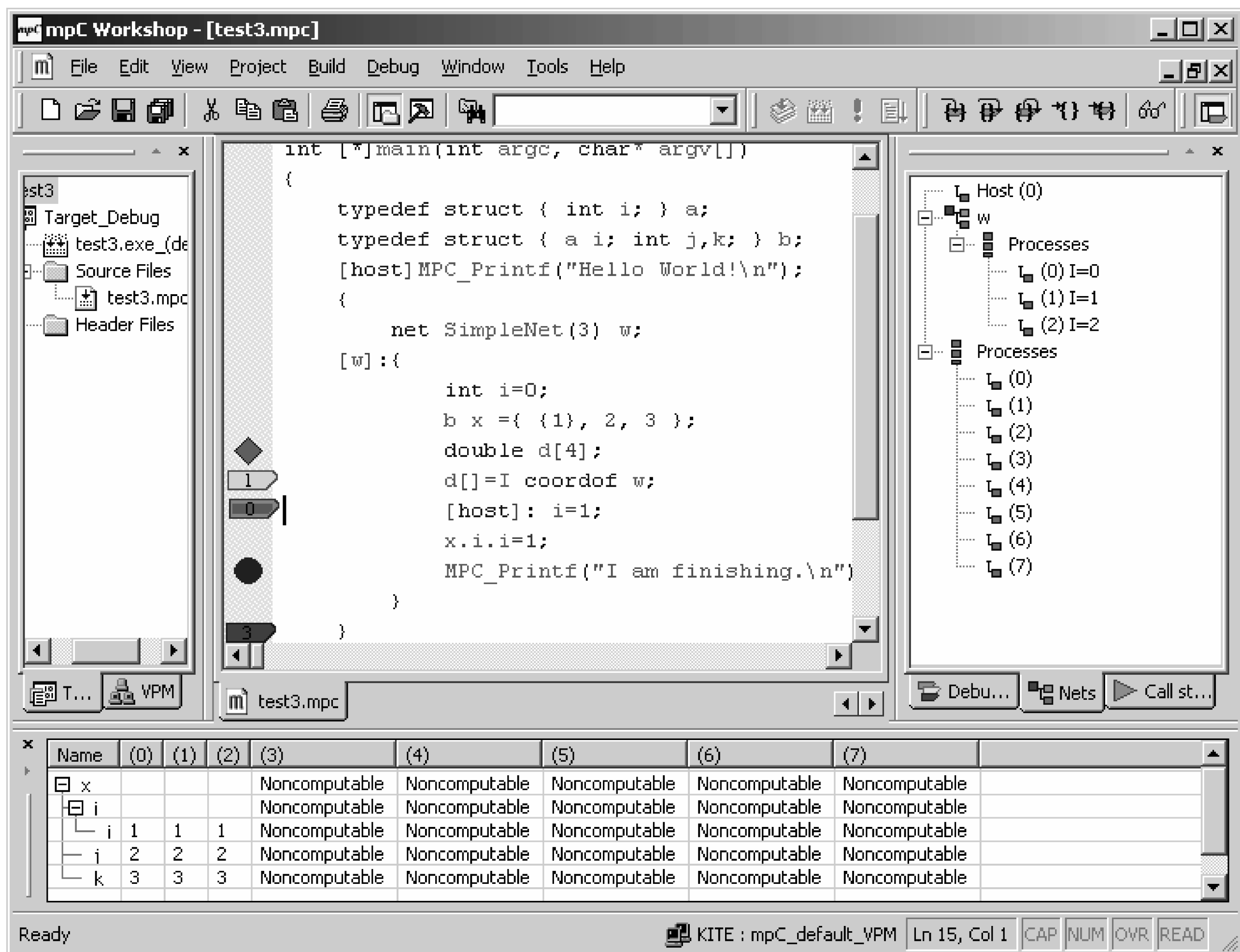

Fig. 3. The main window of the debugger (network window).

All components of the debugger are closely integrated and provide the user with the operations based on the interaction of various components: there exist functions of transition to the process position from the cursor window, setting a filter to a window of variables by a cursor (from the cursor window) or by a network (from the network window). According to users feedback, the debugger provides convenient means for debugging parallel programs in the mpC language.

7. DEBUGGER IMPLEMENTATION FOR MPI PROGRAMS

To implement a debugger for MPI programs that is based on the suggested approach, certain support from an MPI library implementation is required. Currently, the support of debuggers for MPI programs is confined to the interface [17] developed for TotalView, which allows the user to obtain information about the state of internal queues of MPI messages. This information is not sufficient for the implementation of the suggested approach.

As practice shows, the display of MPI communicators, groups, and data types does not require any specific support from the MPI library: all desired information can be obtained in some external (with respect to the library) way (for example, by placing breakpoints on calls of MPI functions, by analyzing the arguments passed, or by performing the MPI actions on creating groups or types). The membership of processes in groups and communicators can also be obtained by means of the standard MPI function `MPI_Group_translate_ranks()`.

Additional difficulties appear when creating new communicators (especially if `MPI_Comm_split()` is used): it is required to correctly identify the created communicators and to have consistent information about the structure of the partition of the set of all processes of the program into the communicators. This problem is also solved by placing breakpoints at the entry to and exit from the functions creating the communicators and by collecting information about input and output values of the functions on one process. To name the communicators, it is suggested to use an

index in the global debugger communicator table and to allow the user to add arbitrary text labels to each communicator.

To implement the suggested scheme (Section 5) of the synchronous parallel step with automated processing of communication dependences between the processes of the parallel program, the existing information about the MPI implementation is not sufficient. In particular, in accordance with the MPI standard [6], the function `MPI_Send()` can be implemented in two ways: with or without the buffering of the message sent, while the particular send mode is arbitrarily selected by a given implementation of the MPI library. In the case of message buffering, the sending operation is local and is completed independently of the result of the corresponding operation of receiving this message by another process (hence, this is not a communication dependence that can result in blocking). Otherwise, the operation is not local and cannot be completed if the operation of receiving this message by another process has not been completed.

Thus, the implementation of the suggested scheme of the parallel step execution requires an access to the internal information about the implementation of the MPI library: there should exist an interface capable of determining the locality of each message sending operation and the topology of the implementation of collective communications (for correct determination of dependences between the processes).

8. CONCLUSIONS

In the development of debuggers for parallel programs, there is a serious contradiction between the debugger universality and the level of service provided for the user. In view of the high complexity of parallel programs, the convenience and efficiency of work with the parallel debugger is very important, which brings us to the conclusion that the debugger universality should be sacrificed for the sake of functionality and convenience in debugging programs that are written in a particular language or use a particular library of parallel programming. In this paper, we have suggested an approach to the development of an advanced parallel debugger that takes into account specific features of the parallel program.

The suggested approach consists in the use of an a priori information about the structure of program objects and communication model of the program execution (the message passing model is implied) for the sake of the automation of displaying data and controlling the debugging process. The paper generalizes the experience of the development of the existing debuggers for parallel programs and suggests a more advanced scheme of the debugger implementation, which is sufficiently universal and can be used in the debugging of programs that use various communication libraries and languages.

The approach described in the paper has been implemented (with certain simplifications) in the mpC Workshop debugger, and the applicability and efficiency of the ideas used in the approach have been verified in practice. Currently, we are developing a new improved debugger for MPI programs.

REFERENCES

1. Huselius, J., Debugging Parallel Systems: A State of the Art Report, *MRTC Report no. 63*, 2002.
2. Kranzlmuller, D., Event Graph Analysis for Debugging Massively Parallel Programs, *PhD Dissertation*, 2000.
3. Etnus: TotalView Users Guide, Version 6.2, 2003.
4. Sun Microsystems: Prism 7.0, User's Guide, 2003.
5. Streamline Computing: DDT 1.6, Users Guide, 2004.
6. The MPI Standard, Version 1.1, June 12, 1995.
7. XMPI—A Run/Debug GUI for MPI, http://www.lam-mpi.org/software/xmpi.
8. Kalinov, A., Karganov, K., Khatzkevich, V., Khorenko, K., Ledovskikh, I., Morozov, D., and Savchenko, S., The Presentation of Information in mpC Workshop Parallel Debugger, *Lecture Notes in Computer Science* (PaCT-2003), Berlin: Springer, 2002, vol. 2763, pp. 497–502.
9. Kalinov, A., Karganov, K., and Khorenko, K., Towards the Proper "Step" Command in Parallel Debuggers, *Proc. of the PADTAD-2002*, 2004.
10. Lastovetsky, A., Arapov, D., Kalinov, A., and Ledovskikh, I., A Parallel Language and its Programming System for Heterogeneous Networks, *Concurrency: Practice and Experience*, 2000, vol. 12, no. 13, pp. 1317–1343.
11. Lastovetsky, A.L., *Parallel Computing on Heterogeneous Networks*, Hoboken, NJ: Wiley, 2003.
12. Wang Feng, An Hong, Chen Zhihui, and Chen Guoliang, Completely Debugging Indeterminate MPI/PVM Programs, *Chinese J. Advanced Software Research*, 2001.
13. Miller, B.P., What to Draw? When to Draw? An Essay on Parallel Program Visualization, *J. Parallel Distributed Computing*, 1993, vol. 18, no. 2.
14. Francioni, J., Determining the Effectiveness of Interfaces for Debugging and Performance Analysis Tools, in *Debugging and Performance Tuning for Parallel Computing Systems*, IEEE, 1996, pp. 127–141.
15. Sistare, S., Allen, D., Bowker, R., Jourdenais, K., Simmons, J., and Title, R., A Scalable Debugger for Massively Parallel Message Passing Programs, in *Debugging and Performance Tuning for Parallel Computing Systems*, IEEE, 1996, pp. 145–160.
16. Sistare, S., Dorenkamp, E., Nevin, N., and Loh, E., MPI Support in the Prism Programming Environment, *Proc. of Supercomputing'99*, 1999.
17. Cownie, J. and Gropp, W., A Standard Interface for Debugger Access to Message Queue Information in MPI, *PVMMPI'99*, 1999.

Spell: Ok