Design and Implementation of a Java-based Distributed Debugger Supporting PVM and MPI

Xingfu Wu [1,2] *Qingping Chen* [3] *Xian-He Sun* [1]

[1] *Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803*

[2] *National Research Center for Intelligent Computing Systems, Chinese Academy of Sciences*

[3] *Department of Computer Science, University of Science and Technology of China*

Abstract This paper presents a client-server debugging model, describes the design and implementation of a portable, scalable, practical distributed debugger based on this model, and compares it with the related work. The distributed debugger based on the sequential debugger gdb or dbx can efficiently support debugging various PVM and MPI programs, and its interfaces are implemented by Java. Therefore, it meets High Performance Debugging Standard's three general goals concerning parallel and distributed debuggers. In addition, it is portable, easy to learn and easy to use.

Keywords: Distributed debugger, message passing, PVM, MPI, Java.

1 Introduction

Several message passing-based software systems which support parallel and distributed computing, such as the popular PVM [1] and MPI [2], have been developed in recent years. These systems are mainly based on standardized Unix systems, use standard sequential language C or Fortran to construct the portable communication primitive library based on standard communication protocols (TCP/IP) with high efficiency for expressing parallel algorithms wisely and validly. However, these systems require the users explicitly assign special data to some processes, thus the deadlock, communication mismatch orders, idle waiting, access conflict, and resource contest may exist in users' parallel programs. A portable, scalable distributed debugger is in demand.

Although some sequential debuggers such as gdb [3] and dbx [4] are supported on multiple platforms, there are no published standards of semantics for debuggers, hence their implementations may vary widely. In the serial programming community, this situation has not been so bad. Serial programmers may continue working on a system for extended periods of time, and can get used to a favorite debugger and not have to worry about changing tools frequently. Within the parallel programming community, a lack of standards has resulted in the quite different scenario because of rapidly changing hardware and software environments. Few debuggers are supported across more than one platform and debuggers are generally criticized for poor usability. The HPD (High Performance Debugging) standard [5] is expected to make a major contribution in solving these problems. The HPD Forum [5] sponsored by the Parallel Tools Consortium established three general goals concerning parallel and distributed debuggers:

- Parallel and distributed debuggers should satisfy basic debugging requirements of high performance computing application developers;
- Parallel and distributed debuggers should be usable – in the sense of easy to learn and easy to use – by these application developers;
- Parallel and distributed debuggers should be consistent across any platforms, so that users of one standard-conforming debugger can switch to another with little or no effort.

To meet the three goals, this paper presents the design and implementation of a portable, scalable, practical distributed debugger DCDB (Dawning Cluster DeBugger) supporting PVM and MPI programs in detail. Section 2 describes the design model, method and Java-based framework of the distributed debugger DCDB from the three aspects: portability, scalability and practicability. Section 3 discusses how the DCDB supports PVM and MPI programs, and shows several views of the debugger. Section 4 compares our work with the related researches. Section 5 concludes this paper.

2 Design Framework of the Distributed Debugger DCDB

In general, debugging parallel programs is divided into two parts: correct debugging and performance debugging. Parallel programs are much more difficult to develop, debug, maintain, and understand than their sequential counterparts. One reason is the difficulty in establishing correctness - which must take into account temporal conditions: liveness, deadlock-freeness, process synchronization and communication, this is often called correctness debugging. Another reason is the diversity of parallel architectures and the need to produce a highly

efficient program fine-tuned to the specific target architectures. The impact of task granularity on a parallel algorithm, the properties of the memory hierarchy, and the intricacies involved in the exploitation of multilevel parallelism, should all be carefully analyzed and used to devise a transformation strategy for the program. The adaptation of an initially inefficient algorithm to a specific hardware is often called performance debugging [6], a term that suggests that the correctness criteria for a parallel algorithm should include requirements for its performance on a given architecture. Therefore, correctness debugging is an essential part of the development process for parallel programs, and most of research efforts naturally focused more heavily on correct debugging. Here, the distributed debugger is a correct debugging tool.

At present, there are often two main methods to develop distributed debuggers:

- Use and extend the functions of sequential debuggers to develop distributed debugger
- Develop general-function distributed debuggers without using any sequential debuggers.

The first method is more popular. For example, P2D2 [7], Mantis [8], and Xmdb [9] are the typical examples using the first method. It is more difficult to use the second method to develop a distributed debugger, and such a distributed debugger is dependable to special platforms, and is not portable, such as CM-5's data parallel debugger Node Prism [10], Dolphinics's TotalView [11], and Intel's interactive parallel debugger IPD [12].

Cluster systems have good scalability, and each node is a complete computer system. Users often are familiar with sequential debuggers of such a complete computer, such as dbx, gdb, which is used in Unix systems, such as IBM's AIX, SUN's SUNOS, Solaris, HP's HPUX, Digital Unix, SCO Unix, FreeBSD, LINUX, and so forth. Therefore, It is indeed worthy of developing a portable, scalable, and user-friendly distributed debugger based on the general sequential debuggers on cluster systems.

The distributed debugger DCDB is to support any Unix network computing environments, such as Networks of Workstations, Networks of Computers, etc. On the Dawning 2000 cluster system [13], we developed the distributed debugger DCDB based on sequential debuggers, and used Java to implement other debugging functions and interfaces. Therefore, the distributed debugger DCDB can be executed on any Unix platforms with Java.

The distributed debugger DCDB provides a portable graphical user interface which is easy to learn and use, and supports debugging (*C*, *Fortran*) PVM and MPI parallel programs. It can implement the distributed debugging by extending the functions of the sequential debuggers. Its advantages are:

1) It uses many sequential debugging commands which users know, and the users can use it without spending much time to learn and use it.
2) It not only efficiently uses current existing debugging techniques, but also simplifies the design and implementation of the distributed debugger.

The distributed debugger DCDB is divided into four levels from top to bottom, its design framework is shown in Figure 1.

L 1. User graphical interface level
L 2. Network communication level
L 3. Sequential debugger level
L 4. Parallel program level

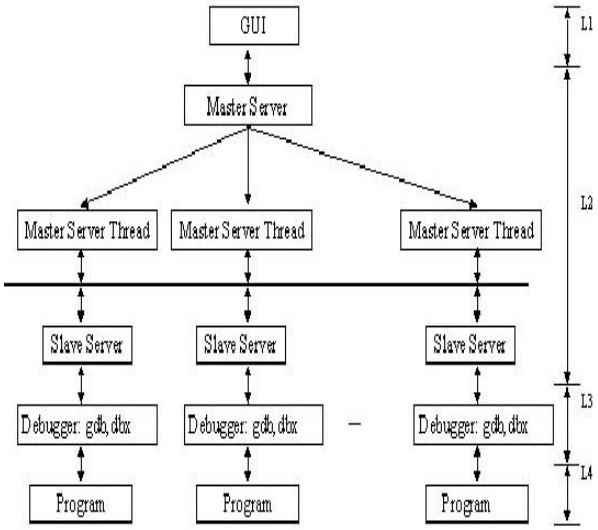

Figure 1 Framework of DCDB

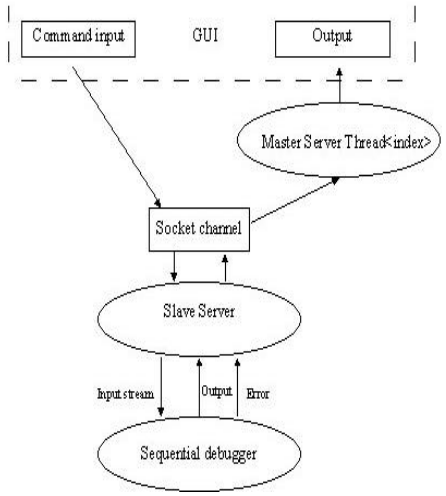

Figure 2 Main dataflow graph of DCDB

Generally, the users only see the GUI (Graphical User Interface) of the debugger. The debugging command which the user inputs can be sent to the sequential debugger through Socket Channel. Master Sever is the Socket's sever, Slave Server is the Socket's client. The results, which the sequential debugger executes a debugging command, are returned through Socket Channel, and are displayed in the GUI. The main dataflow graph of the DCDB is shown in Figure 2.

The DCDB has the following main features:

1) Portability
   The DCDB was implemented by Java, and could support debugging various PVM and MPI programs. Because Java language is independent to any platforms, and PVM and MPI have good portability, therefore, the distributed debugger DCDB is portable, and can support debugging parallel programs in heterogeneous network computing environments.
2) Scalability
   Group management function of the DCDB is very strong. It can send a debugging command to many nodes in a group simultaneously. It is very convenient for the users to use the function to debug their programs. The DCDB may efficiently support debugging PVM and MPI programs with various sizes, especially, using the group management function. When the debugged parallel program creates many parallel sub-processes, the DCDB can display their source codes in the debugging windows in parallel, and any debugging windows may be open or closed according to the users' requirements.
3) Practicability
   The DCDB has a user-friendly graphical user interface, such as simple operation graphical interfaces, simple window contents, simple and understandable command windows, short and clear hints, etc. The users can choose to use their favorite sequential debugger such as dbx or gdb. The DCDB can greatly reduce the effect of the execution behavior of PVM or MPI programs using the debugger to debug them.

3 Implementation of Distributed Debugger DCDB

The DCDB is a distributed debugger based on the sequential debugger dbx or gdb in cluster systems. It supports debugging not only SPMD or MPMD (C or Fortran) PVM programs, but also C or Fortran MPI programs. Because Java language has good portability, and can support multiprocess control, multithread control, socket communication, Input/Output redirection, etc., we use it to implement overall graphical interfaces and low-level debugging interfaces of the DCDB, so that the DCDB has good portability. In the following subsections, we shall discuss the implementation of DCDB in detail.

3.1 Debugging process of the DCDB

The DCDB often executes a monitoring program Master Sever in the local machine, and the program is in charge of receiving the connection requests from other programs. When receiving a connection request, the Master Sever shall create a new thread Master Server Thread, and makes the true client/server connection with that program which sent the request (shown in Figure 1).

The debugging process of the DCDB mainly includes environment parameter configuration, creation of process group, starting the debugging, finishing the debugging, and exiting the DCDB.

1) Parameter configuration

The users' system environments are often very different, such as their home directories, the sequential debuggers which they would like to use, and so on. Therefore, before the users use the DCDB, they need configure various parameters according to their habit. These parameters are:

(1) The number of processes (the maximum number of debugging processes in DCDB is not limited, but the number should not be less than that of parallel processes.)
(2) The directory path of PVM or MPI source codes
(3) The whole directory of a sequential debugger, such as /usr/bin/dbx or /usr/bin/gdb
(4) The setup of debugging commands of a sequential debugger

When the first configuration is finished, it will be used as a default configuration. When the distributed debugger is executed again, these parameters need not be configured again unless reconfiguration is needed.

2) Creation of a process group

The DCDB provides the function of process group management. If some processes are set as a group, then it is valid for all processes of the group to send debugging commands to the node group. Therefore, using a debugging command can control all processes of the same group. The DCDB can use the global condition control to efficiently monitor the real-time process of a program execution.

3) Starting the debugging

The DCDB can support debugging C or Fortran PVM and MPI programs. It provides a user-friendly interface to choose the types of debugging programs (such as PVM, MPI) and the types of programming languages (such as C, Fortran). It also provides the debugging window of each

process, and sends some processes a debugging command to let them do it, and displays the execution results of these processes on the debugging windows. The DCDB provides a replay function to treat the indetermination of parallel execution process based on event traces.

4) Finishing the debugging

The process of debugging a program is a process of continually finding the errors of the program and modifying them. When the debugging is finished, the DCDB can automatically terminate all debugging processes (local and remote processes), and clear various debugging "garbage", so that none can affect the next debugging. If this debugging fails, the DCDB provides a function to automatically clear various debugging "garbage". For example, the "Clear" button shown in Figure 3 is for this function.

5) Exiting the DCDB

This means exiting the whole DCDB debugging environment.

3.2 The support for debugging PVM programs

In this subsection, we depict how the distributed debugger DCDB supports debugging PVM programs.

1) Modifying the default debugger of PVM

In PVM system, the default sequential debugger is assigned in the shell file "debugger" of the directory $PVM_ROOT/lib. In the DCDB, we modified the shell file so that when creating a new process, in fact, it executes the Slave Server program. This program is in charge of communication between the local process and remote process(es), and is the client of the communication channel. The Slave Server program can get the sequential debugger chosen by the users, and starts a new process under the control of the sequential debugger.

2) Modifying PVM source codes

In PVM programs, the users only specify PvmTaskDebug in pvm_spawn(), do not need other modification of the source code. If PvmTaskDebug is specified in pvm_spawn(), PVM runs $PVM_ROOT/lib /debugger, which opens a debugging window in which it runs the task in a sequential debugger.

3) Compiling PVM source codes

In order to support source-level program debugging, when PVM programs are compiled, only –g option is used and the PVM standard subroutine libraries are linked. No other special library is needed.

4) Debugging

When users start to use the DCDB to debug a PVM program, they need to input the debugging program name in the main interface of DCDB, the DCDB can execute a Slave Server program in the local machine, and the debugging program name is regarded as a parameter of the program. Then, the Slave Server program requests to make a connection with the local monitoring program Master Server. The monitoring program can create a new thread Master Server Thread to make the true client/server connection. After the connection is made, the Slave Server runs the executable in a sequential debugger. From the connection channel, the local Master Server Thread can get the debugging program name on a node and the node's IP address, regards them as the titles of the respective debugging windows, and displays the respective source codes in their debugging windows. It can also get the source code of an executable from the node which the program are being executed on, then returns it to the local node and displays it in its debugging window.

At this time, the user may use the DCDB's user interface to input some debugging commands, such as setting breakpoints, running step by step, etc. These commands are sent to the sequential debugger through the connection channel between the local Master Server Thread and Slave Server. After the sequential debugger executes the user's requests, the output results are returned through the connection channel and are displayed in the debugging information columns.

When the PVM program runs the subroutine pvm_spawn(), PVM may run a Slave Server program on any node of the cluster system. Similarly, these Slave Server programs can also request to make connections with their local monitoring programs, and each local monitoring program shall create a new thread Master Server Thread to make the true client/server connection. If the users want to control many processes at the same time, they need to set and choose the suitable process group. If so, the debugging commands can be sent to many sequential debuggers through many connection channels.

3.3 The support for debugging MPI programs

In this subsection, we describe how the distributed debugger DCDB supports debugging MPI programs.

1) Modifying MPI source codes

Do not need any modification of the MPI source code.

2) Compiling MPI source codes

In order to support source-level program debugging, when MPI programs are compiled, only –g option is used and the MPICH standard subroutine libraries are linked. No other special library is needed.

3) Debugging

When users start to use the DCDB to debug a MPI program, they need to input the debugging program name in the main interface of DCDB, the DCDB can execute a Slave Server program in the local machine, and the debugging program name is regarded as a parameter of the program. Then the Slave Server program requests to make a connection with the local monitoring program Master Server. The monitoring program can create a new thread Master Server Thread to make the true client/server connection. After the connection is made, the Slave Server runs the following command:

mpirun –debugger –np procnum program –p4norem,

(Where "debugger" is a sequential debugger the users chose, such as dbx or gdb; "procnum" is the number of processes; "program" is the executable; The point of this option "–p4norem " is to enable the user to start the remote processes under his favorite debugger.) and sends its output results to the Master Server Thread. The thread shall start a Slave Server on the specified node. The Slave Server requests to make a connection with its local monitoring program, and runs the executable in a sequential debugger. From the connection channel, the local Master Server Thread can get the debugging program name on a node and the node's IP address, regards them as the titles of the respective debugging windows, and displays the respective source codes in their debugging windows. It can also get the source code of the executable from the node which the program are being executed on, then returns it to the local node and displays it in its debugging window.

At this time, the user may use the DCDB's user interface to input some debugging commands, such as setting breakpoints, running step by step, etc. These commands are sent to the sequential debugger through the connection channel between the local Master Server Thread and Slave Server. After the sequential debugger executes the user's requests, the output results are returned through the connection channel and are displayed in the debugging information columns.

If the users want to control many processes at the same time, they need to set and choose the suitable process group. If so, the debugging commands can be sent to many sequential debuggers through many connection channels.

3.4 Several main views of the DCDB

The distributed debugger DCDB can support the source-level multiprocess debugging. Its debugging window is divided into four parts from top to bottom: source code path, debugging command input, source code browser, and debugging information output (shown in Figure 3). The user can input or modify the path of his source code, and types a debugging command or clicks a debugging command button. We shall show some views of the DCDB for debugging a PVM program and a MPI program as follows.

1) Debugging PVM programs

This example shown in Figure 3 is a Master-Slave *Fortran* PVM program. Its master program is master1.f, and the executable is fmaster1. Its slave program is slave1.f, and the executable is fslave1. The fmaster1 is executed on the master node 10.10.10.101, and the fslave1 is executed on the slave nodes: print, compass, and paper, respectively. In Figure 3, these buttons of the DCDB's main interface stand for the debugging windows, where P0 is the debugging window Debug_Window0, P1 is the Debug_Window1, P2 is the Debug_Window2, and P3 is the Debug_Window3. The number of these buttons should not be less than that of real processes. The button P4 and P5 are not used, thus they are marked by red color, and are not clickable. The master process P0 is marked by green color. The slave processes P1, P2, P3 are marked by yellow color.

2) Debugging MPI programs

This example shown in Figure 4 is a *C* MPI program. The source code is systest.c, its executable is systest. The systest program is loaded to the nodes: *compass*, *paper*, *print*, respectively. Thus, the three buttons P0, P1 and P2 are clickable, the rest are not clickable. In Fiugre 4, only the button P1 and P2 are clicked, therefore, only the debugging windows Debug_window1 and Debug_ window2 are shown.

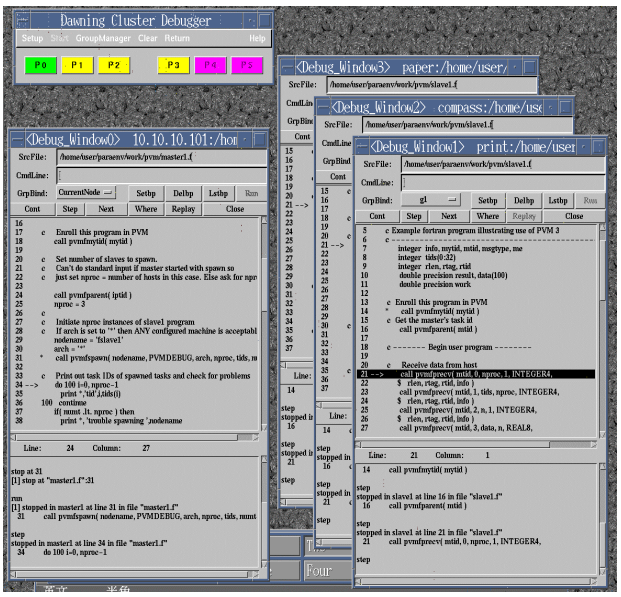

Figure 3 The DCDB views for debugging PVM programs

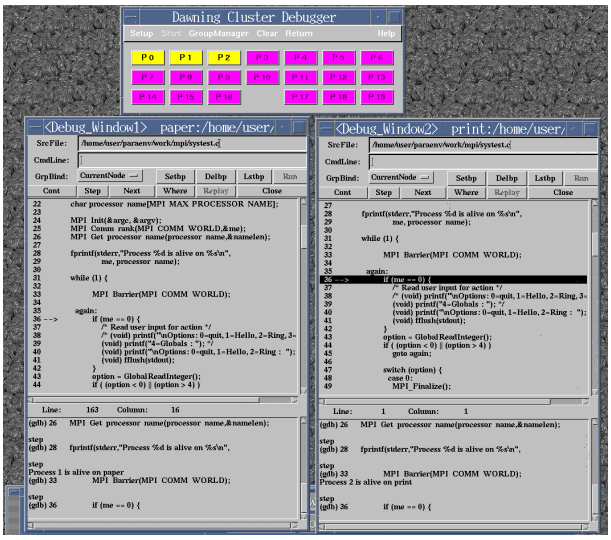

Figure 4 The DCDB views for debugging MPI programs

4 Comparison with related work

There are two research works: P2D2 [7] and Mantis [8] close to our work. They are both based on the sequential debugger gdb, and are implemented by Motif. Mantis only supports debugging Split-C programs. P2D2 is only used on IBM SPs, SGI workstations and Origins, and Linux systems. Its prototype design claims that it can support debugging PVM and MPI programs. Xmdb is a simulated parallel debugger supporting PVM programs on a single processor. It requires that a PVM program must be compiled by linking its special library before the program is debugged, so that the PVM program can be instrumented. Thus, it affects the execution behavior of the PVM program to some degree. The DCDB can greatly reduce the effect of the execution behavior of PVM or MPI programs using the debugger to debug them in the run-time PVM or MPI environment.

TotalView [11] is based on X window systems. It can support debugging PVM and MPI programs on IBM AIX, Compaq Digital UNIX, SGI IRIX, SUN SUNOS and SOLORIS systems, and is only available on homogenous computer systems. It does not support heterogeneous computer systems. Our DCDB can efficiently support heterogeneous computer systems. Data parallel debugger Prism [10] is develop for CM-5 machine. It is neither portable nor general-purpose. Our DCDB is implemented by Java, and can be executed on any Unix platforms with PVM or MPI systems. Therefore, it is portable and general-purpose.

5 Conclusions

This paper describes the design and implementation of a Java-based distributed debugger DCDB, which supports debugging PVM and MPI programs. We develop the distributed debugger by extending the functions of current existing sequential debuggers, and use Java to implement the overall interfaces of the DCDB so that our distributed debugger can meet High Performance Debugging Standard's three general goals concerning parallel and distributed debuggers, and is portable, easy to learn and easy to use. Anyway, it is the first attempt for us to use Java to implement the distributed debugger. Many works will be further done, such as communication optimization, the support for Microsoft NT cluster system.

References

[1] A. Geist, et al., PVM: Parallel Virtual Machine - A Users's Guide and Tutorial for Networked Parallel Computing, *the MIT Press*, 1994.

[2] W.Groupp and E.Lusk, User's Guide for MPICH, a Portable Implementation of MPI, *Argonne National Laboratory*, USA, 1994.

[3] R. Stallman and C. Support, *Debugging with GDB*, Cygnus Solutions, Inc., 1994.

[4] Sunsoft, Inc., *Solaris Application Developer's Guide*, 1997.

[5] Parallel Tools Consortium, *HPD (High Performance Debugging) Version 1 Standard: Command Interface for Parallel Debuggers*, http:/www.ptools.org/hpdf/draft/, Sep. 1998.

[6] Xingfu Wu, *Performance Evaluation, Prediction and Visualization of Parallel Systems*, Kluwer Academic Publishers, ISBN 0-7923-8462-8, Boston, 1999.

[7] D.Cheng and R.Hood, A Portable Debugger for Parallel and Distributed Programs, *Proc. of Supercomputing'94*, Nov.1994. See also http://science.nas.nasa.gov/Groups/Tools/ Projects/P2D2/.

[8] S.S.Lumetta, *Mantis: A Debugger for the Split-C Language*, University of California at Berkley, Tech. Report #CSD-95-865, 1995.

[9] Damodaran-Kamal, *Xmdb Version 1.0 User Manual 1.2*, Los Alamos National Laboratory, 1995.

[10] Think Machines Corporation, *Prism 2.0 Release Notes*, May 1994.

[11] Dolphin Interconnect Solutions, Inc., *TotalView Multiprocess Debugger*, Release 3.7, http://www.dolphinics.com/.

[12] Intel Corporation, *iPSC/2 and iPSC/860 Interactive Parallel Debugger Manual*, April 1991.

[13] Xingfu Wu, Qingping Chen, Xiao Hu, Yonggang Hu, Ming Zhu, and Jiesheng Wu, *Design and Implementation of Cluster System-oriented Parallel Programming Environments*, Technical Report, National Research Center for Intelligent Computing Systems, Chinese Academy of Sciences, 1998.