Cluster Computing 5, 365–375, 2002


# Design and Development of a Scalable Distributed Debugger for Cluster Computing

XINGFU WU
*Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208, USA*

QINGPING CHEN
*Department of Computer Science, University of Science and Technology of China, Anhui, PR China*

XIAN-HE SUN
*Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA*

**Abstract.** Debugging is an essential part of parallel and distributed processing. However, developing parallel and distributed debugger is difficult. This is especially true for cluster computing where heterogeneity presents. In this paper, we first give a survey of the current debugging techniques and existing tools, and then present a client–server debugging model. Based on this model, we discuss the design and development of a practical scalable distributed debugging system for cluster computing in detail, and give two case studies to show how the distributed debugging system efficiently supports debugging message-passing programs such as various MPI and PVM programs. The newly developed distributed debugger is based on the sequential debugger gdb and dbx. It has the capability of scaling to handle hundreds of processes. Its interfaces are completely implemented in Java, and its graphical user interface is the same on all computing platforms. In addition, it is portable, easy to learn and use.

**Keywords:** distributed debugger, cluster systems, message passing, PVM, MPI, Java



## 1. Introduction

Computer system design is dominated by the dramatic rate of advance in small desktop systems in recent years. For large-scale systems such as Massively Parallel Processors (MPPs), due to their long engineering lag time and high cost, etc., they do not fit in with the rapid development of processor chips. Generally, before a MPP is completely developed, its processor chips are beginning to date. Therefore, at present, cluster systems are poised to become a primary computing infrastructure from low end interactive computing to demanding serial and parallel applications.

Software environments for cluster systems often play an important role in efficiently utilizing the advanced cluster architecture. Several message passing-based software systems to support cluster computing, such as the popular MPI [7] and PVM [6], 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 users explicitly assign special data to some processes, the deadlock, communication mismatch orders, idle waiting, access conflict, and resource contest often exist in the users' parallel programs. Thus, as an essential part of the development process for parallel programs, a portable, scalable, distributed tool for correctness debugging is in demand.

This paper presents a survey of current debugging techniques and existing tools, and the design and implementation of a portable, scalable, practical distributed debugger CDB (dawning Cluster DeBugger) for cluster computing in detail. Section 2 discusses the current debugging techniques and tools. Section 3 mainly describes the design model, method and framework of the distributed debugger CDB from the three aspects: portability, scalability and practicability. Sections 4 and 5 discuss how the CDB efficiently supports message-passing programs such as MPI and PVM programs, and shows several views of the debugger implemented in Java. Section 6 concludes this paper.

## 2. Current debugging techniques and tools

In general, parallel and distributed program debugging can be divided into two categories: correctness 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 a given specific target architecture. 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 [9,23], a term that suggests that the correctness criteria for a parallel algorithm should be pre-assumed and include requirements for its performance on a given architecture. Therefore, correctness debugging is an essential part of the development process for parallel programs, and initial research efforts are naturally focused more heavily on correctness debugging.

A programmer debugging parallel or sequential program always wants to know which functions and/or program blocks the program enters and what the values of variables are. A simply way to get the information is to insert "print" statements in the program that report what section of the program is executing and/or output the values of the specific variables. This approach requires no skills or tools beyond what the programmer is using to write the program, but the programmer must decide in advance which variables and/or execution statements to print and where to put the print statements. Inserting such a print statement requires re-editing, re-compiling, and re-executing the program. In addition, if the program outputs a great deal of data, finding the information of interest can be tedious.

There are many sequential debuggers to be used on sequential computers for many years. Although some sequential debuggers such as gdb [19] and dbx [20] are supported on multiple platforms, there are no published standards of semantics for debuggers. Therefore, debugger implementations are subject to considerable variation in both the kinds of commands that are available and what specific actions are performed in conjunction with any particular command. 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, 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 a quite different scenario because of rapidly changing hardware and software environments. Few debuggers are supported across more than one platform, and the parallel debuggers are often criticized for poor usability. From the users' viewpoint, it is not effective to have to learn a new debugger for every new computer. To date, there is no parallel debugger that behaves consistently across various different architectures and operating systems, nor one that is considered easy to learn and use. Thus, for the users, easy-to-learn-and-use and cross-platform compatibility have become key considerations in user decisions about whether to adopt new parallel environments and tools. From the parallel computer manufactures' viewpoint, there has not been any real economic advantage to expending efforts so that debuggers will be consistent with those on competitors' machines. As a result, each new machine presents one debugger that has features incompatible with earlier version, or a totally new debugger. This lack of portability and standard makes programming on cluster systems a hard task, especially under a heterogeneous environment. The High Performance Debugging (HPD) standard [5,10] is expected to make a major contribution in solving these problems.

The High Performance Debugging Forum (HPDF) is a collaborative effort involving both researchers and commercial debugger developers in the area of parallel debugging, as well as representatives of High Performance Computing (HPC) user organizations. It was established in March 1997, and is sponsored by the Parallel Tools Consortium. Its overall goal is to define standards relevant to debugging tools for HPC systems. The HPD standard attempts to be both architecture- and operating system-neutral, in the sense that it should be possible to build a standard-conforming debugger on a wide variety of different computing systems. The HPD Forum 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.

Since the HPD standard attempts to address the needs of HPC application developers and to be both hardware- and operating-system-neutral, it assumes that programs need to be system-independent. Explicit parallelism is assumed as the basic programming model. The standard applies to both distributed-memory programming (like multiple processes cooperating via message-passing libraries such as MPI or PVM) and shared-memory programming (like multiple threads of execution in a single address space such as HPF). The standard may also be useful for implicit parallel programs, but the issues of how to map from runtime information or intermediate-level information to original user source code are not addresses in the standard. The HPD standard does not address interpreted languages that typically are packaged with a built-in debugger such as Java. It only defines a standard command-based (i.e., non-graphical) interface for parallel debuggers, and no future versions would deal with such issues as graphical interfaces and support for debugging optimized code.

Effective user debugging of parallel and distributed code or sequential code has been a topic of theoretical and practical interest in software development and parallel and distributed communities for several decades, yet the state of the art is still highly uneven today. Pancake and Netzer [17] collected a list of references including technical reports, journal and conference papers, and Ph.D. dissertations dealing with parallel debuggers published before the middle of 1993. Because of the nature of rapidly changing high performance computing hardware and software environments, most of them are out of

date today. Therefore, we present a brief survey of the current literature and practice on parallel and distributed debuggers as follows.

TotalView [4] is a commercial parallel and distributed debugger, and is being actively developed currently. It is based on X window systems, and has been ported to multiple platforms, languages and parallel programming models with great effort. It also has the versions for message-passing (such as MPI, PVM) or shared-memory (such as HPF). It is only available on homogenous parallel computer systems. TotalView's user interface caters primarily to the expert user, and source display windows are tied to individual processors.

P2D2 (Portable Parallel/Distributed Debugger) [2,11] and Mantis [15] are both based on the sequential debugger gdb, and are implemented by Motif. Mantis only supports debugging Split-C programs. P2D2 adopts a client-server approach to provide a uniform interface for various platforms, communication libraries, and programming models. Its prototype can support debugging PVM and MPI programs. By specifying protocols for interaction between a user interface client and a debugger server, P2D2 tries to separate the development efforts for the two parts to achieve portability.

PDBX and XPDBX [12] only run under IBM POE environment, and are based on only the IBM AIX dbx debugger. Xmdb (Message based DeBugger) [3] 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.

Some distributed debuggers have a front-end to act as a unified user interface and to serve as an agent for scattering or gathering debugging operations across the collection of processes [8,16]. But such a front-end is tedious to implement for heterogeneous computation because it has to account for subtle differences in the input and output formats of the debuggers.

Node Prism [18,21] extends the data parallel debugger Prism for CM-5 machine to support the message-passing paradigm. Prism addresses scalability by taking advantage of the parallel nature of the debugger itself. It is only available on the CM-5.

X windows Parallel DeBugger (XPDB) [24] can be used to monitor and control the execution of SPPL (Stuttgart Parallel Processing Library which is a message-passing library) programs. It works on the message exchange level and treats the parallel program as objects, which are exchanging messages. To debug processes internally XPDB can call sequential source-level debuggers. It is only available for SPPL programs.

IBM distributed debugger [13] is a client/server application that enables to detect and diagnose errors in programs. This client/server design makes it possible to debug programs running on systems accessible through a network connection. The debugger server (also known as a debug engine) runs on the same system where the program debugged runs. This system can be a workstation or a system accessible through a network. If a program running on the workstation is debugged, local debugging is performed. If a program running on a system accessible through a network connection is debugged, remote debugging is performed. The Distributed Debugger client is a graphical user interface where commands used by the debug engine can be issued to control the execution of a program. The debugger is only available for C/C++, COBOL, Fortran, High Performance Compiled Java, interpreted Java, and PL/I.

For optimized code debugging, Brender, Nelson and Arsenault [1] presented a good survey of the literature and current practice that leads to the identification of three aspects of debugging optimized code that seem to be critical as well as tractable without extraordinary efforts. They are: (1) split lifetime support for variables whose allocation varies within a program combined with definition point reporting for currency determination; (2) stepping and setting breakpoints based on a semantic event characterization of program behavior; (3) treatment of inlined routine calls in a manner that makes inlining largely transparent.

For designing an High Performance Fortran (HPF) debugger, LaFrance-Linden [14] presented several of the challenges involved in designing an HPF debugger and how an experimental debugging technology successfully addresses many of them.

In this paper, we focus on discussing distributed debuggers for cluster computing, design and develop a Java-based scalable distributed debugger CDB to meet three general goals concerning parallel and distributed debuggers.

## 3. Design framework and models of the distributed debugger CDB

In general, cluster systems have good scalability, and each node is a complete computer system. Users are familiar with sequential debuggers of such a complete computer, such as dbx, gdb, which is used in all major 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 CDB is to support any Unix network computing environments, such as Networks of Workstations, Networks of Computers, etc. On the Dawning2000 cluster system [22], we designed and developed the distributed debugger CDB based on sequential debuggers, and used Java to implement other debugging functions and interfaces. Therefore, the distributed debugger CDB can be executed on any Unix platforms with Java.

A sequential debugger is a tool that gives a user visibility into an executing program and control over the target program. A parallel and distributed debugger performs the same function for a parallel and distributed program. In parallel and distributed computing, an executing program consists of one or more processes, each associated with a particular executable. Each process occupies a memory address space,

and has one or more threads, each with its own register set and stack. Therefore, the target program is the complete set of threads and/or communicating processes that make up a given execution of the user's application over the full course of program execution.

To initiate debugging sessions, a debugger can be invoked from the command line and the target program executed within the debugger environment. For a parallel and distributed target program, which consists of many processes, the debugger may need to interact with the run-time system that is responsible for managing those processes (e.g., message-passing systems such as MPI or PVM). Assume that debugging information is only generated when a target executable is compiled with special options "-g" in effect. (Note that when a program is compiled with "-g -O", the optimizer rearranges the source code. Do not be surprised when the execution path does not exactly match the source code.) In this case, the debugger not only controls the executable(s) that constitute the target program, reflected in memory and register values of the execution program, but also utilizes debugging information associated with the source code of the executables.

To control a distributed process, users want to be able to set breakpoints, and start and stop the distributed programs. A breakpoint specifies that execution should stop whenever it reaches a given location relative to the source code. In the CDB, stopping a process means any processes that have not triggered a breakpoint will be unaffected, but all threads in each process that have done so will be stopped together. Starting a process is similar to the stopping the process.

The distributed debugger CDB is designed to receive user input on what actions should be taken to control the target program's execution or to reveal information about it. It provides a portable graphical user interface easy to learn and use. The input is via the graphical interface. In response to user input, a debugger typically issues a variety of debugging messages. Some of them confirm that an operation completed successfully or indicate that a problem occurred. Others provide the detailed information about what the debugger/target program is doing. The CDB displays the debugging information in the corresponding debugging window.

The CDB implements distributed debugging by mainly extending the functions of the sequential debuggers. Its advantages are:

(1) It uses many sequential debugging commands that users know, and the users can use the CDB without taking 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 CDB is divided into four levels from top to bottom as follows, its architecture is shown in figure 1.

Generally, a user can only see the CDB's GUI (Graphical User Interface). The debugging command that the user inputs can be sent to the sequential debugger through Socket Channel. *Master Server* is the Socket's server; *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 CDB is shown in figure 2.

As shown in figures 1 and 2, the *Slave Server* is an implementation of Java objects that translates the request on a *Master Server* object into sequential debugger's commands, sends the commands to the appropriate instance of the sequential debugger, parses the sequential debugger's reply when it arrives, then communicates the result to the code that requested the service. Notice that, when a target program is executing under the control of a sequential debugger, both the debugger and the user's application may be reading or writing output to the same terminal.

The CDB's design is kept simple and scalable. The CDB has the following main features:

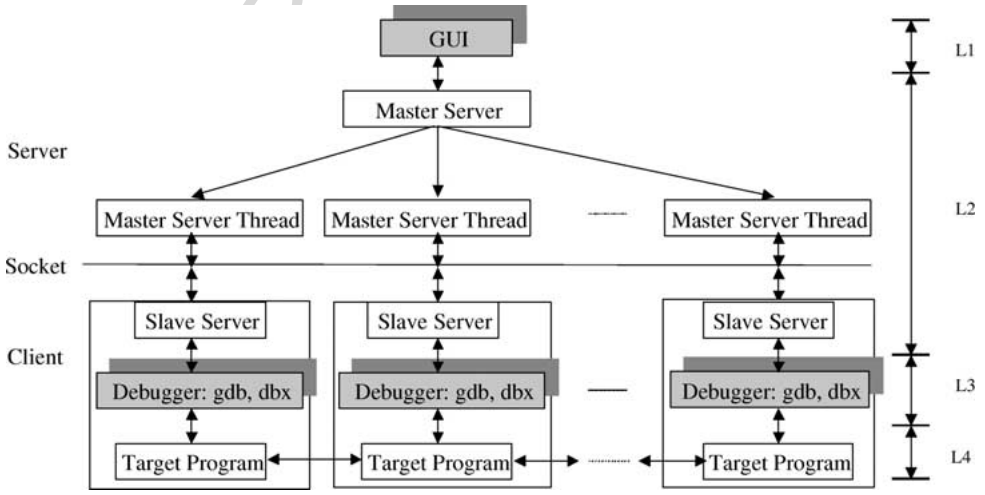


Figure 1. Architecture of the CDB. L1: graphical user interface level, L2: network communication level, L3: sequential debugger level, and L4: parallel and distributed program level.

### 3.1. Portability

Basically, a debugger's job is to provide mapping services between the user's program at the source level and the machine version at the object level. In performing this procedure, a debugger replies on its execution environment for many services [11]. These services may depend on the target architecture for things such as trap instructions used in breakpoint implementation, the operating system for process control and access to the address space, and the compiler for symbol table information that enables mapping between source and object. The CDB isolates them from the user interface through the client–server debugging model shown in figure 1.

In addition, the CDB was implemented by Java, and could support debugging various PVM and MPI programs. Because Java language is independent on platforms, and popular message passing systems: PVM and MPI have good portability, therefore, the distributed debugger CDB is portable, and can support debugging parallel programs in heterogeneous network computing environments.

### 3.2. Scalability

In the CDB, the concept of process group is proposed. A user can set several logical processes into a process group. If the user inputs a debugging command for the group, the CDB can send the command to all processes in the group simultaneously. It is very convenient for the users to use the function to debug their programs. The CDB may efficiently support debugging PVM and MPI programs with various sizes, especially, using the group management function. It has the capability of controlling the execution of hundreds of processes. When the debugged parallel program creates many parallel sub-processes, the CDB can display their source codes in the debugging windows in parallel, and any debugging windows may be activated or closed according to the users' requirements.

Here is the logical breakpoint determination of the CDB in a process group shown in figure 3. Setting a breakpoint in a logical process group sets a breakpoint in each physical process in the group and collects the physical representations into a logical breakpoint. As shown in figure 3, G0, G1, ..., G$n$ are logical process groups defined by the user. "•" means "stopped at some breakpoint". P0G$m$, P1G$m$, ..., P$q$G$m$ are all physical processes of Group G$m$ ($q = i, j, \ldots, k$; $m = 0, 1, \ldots, n$). Setting a breakpoint in a logical process group G$m$ means setting the same breakpoint in all physical processes P0G$m$, P1G$m$, ..., P$q$G$m$.

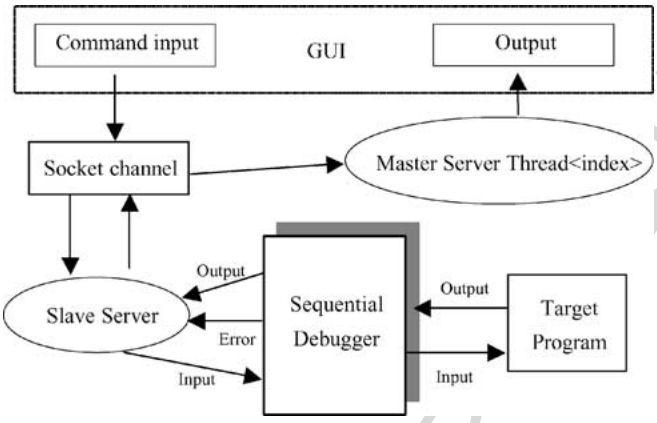


Figure 2. Main dataflow graph of the CDB.

### 3.3. Practicability

Our goal is to keep the CDB simple and scalable. The CDB 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. It provides the same graphical user interface on all platforms. The users can choose to use their favorite sequential debugger such as dbx or gdb. The CDB can greatly reduce the effect of the execution behavior of PVM or MPI programs using the debugger to debug them. Some views of the CDB's GUI will be shown in the following sections.

## 4. Implementation of the distributed debugger CDB

In general, the overall experience of using a tool is often as important as the tool's functionality. For this reason, we step

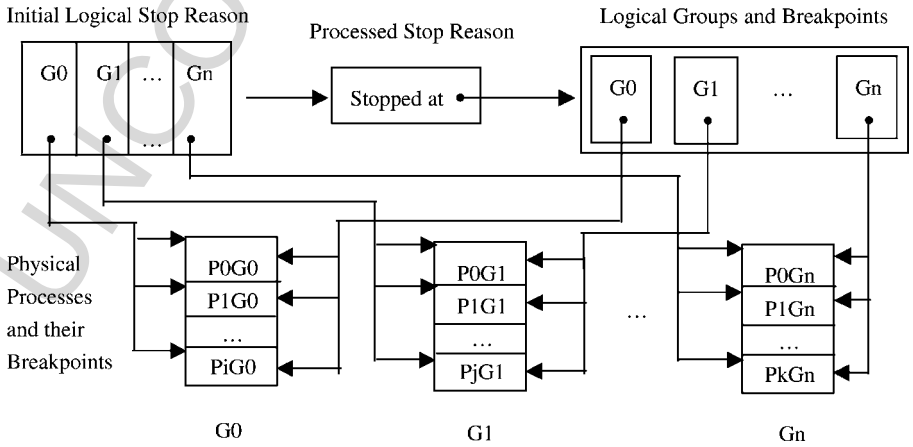


Figure 3. Logical breakpoint determination of the CDB.

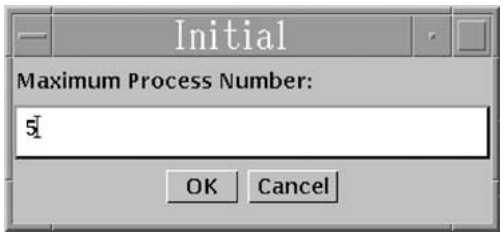


Figure 4. Initial window.

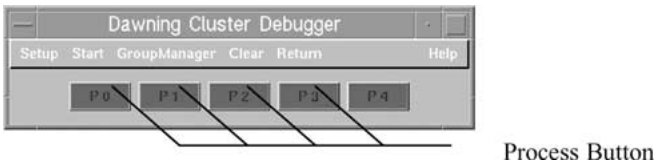


Figure 5. Main window of the CDB.


Figure 6. Startup debugging window.

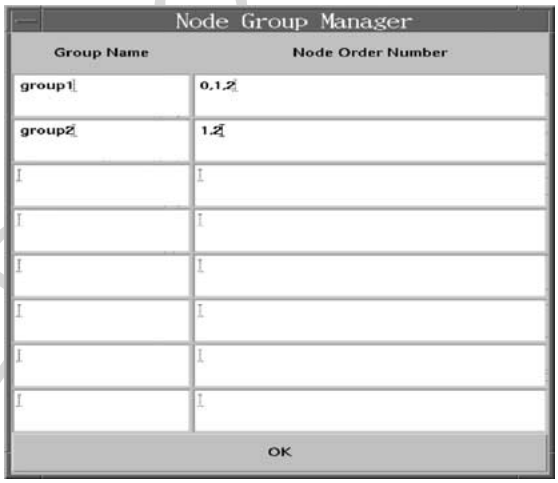


Figure 7. Process group window.

through the process of debugging a parallel and distributed program using the CDB in this section. At the same time, we explain the tool's functionality and how it fits into parallel debugging objectives. 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 CDB, so that the CDB has good portability. In the following subsections, we shall discuss the implementation of the CDB in detail.

The CDB executes a monitoring program *Master Server* 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 Server* 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 CDB mainly includes environment parameter configuration, choosing a target program, creation of process group, starting the debugging, finishing the debugging, and exiting the CDB.

### *4.1. 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 CDB, they need to configure various parameters according to their environments. These parameters are:

(1) the number of processes;

(2) the directory path of message-passing programs such as 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 the chosen sequential debugger.

When the CDB is invoked, an initial window is first appeared. This window is shown in figure 4. The user must input the maximum number of debugging processes, and the number should not be less than that of parallel processes. Although the maximum number of debugging processes in the CDB is not limited, it is often set less than 256 because of some limitations such as operating systems and screen size, etc.

After this step, the main window of the CDB occurs as shown in figure 5. It has the following components:

- Process buttons
  The number of process buttons equals the maximum number of debugging processes that have been given. The button number stands for the logical process number. For example, process button P0, P1, . . . , P*n* stands for the logical process 0, 1, . . . , *n*, respectively. Each process button will be mapped to a process of the parallel program. These buttons are not clickable right now.
- Menu bar
  Setup: Configuring some parameters such as sequential debugger, debugging commands, the path of source code, etc.;

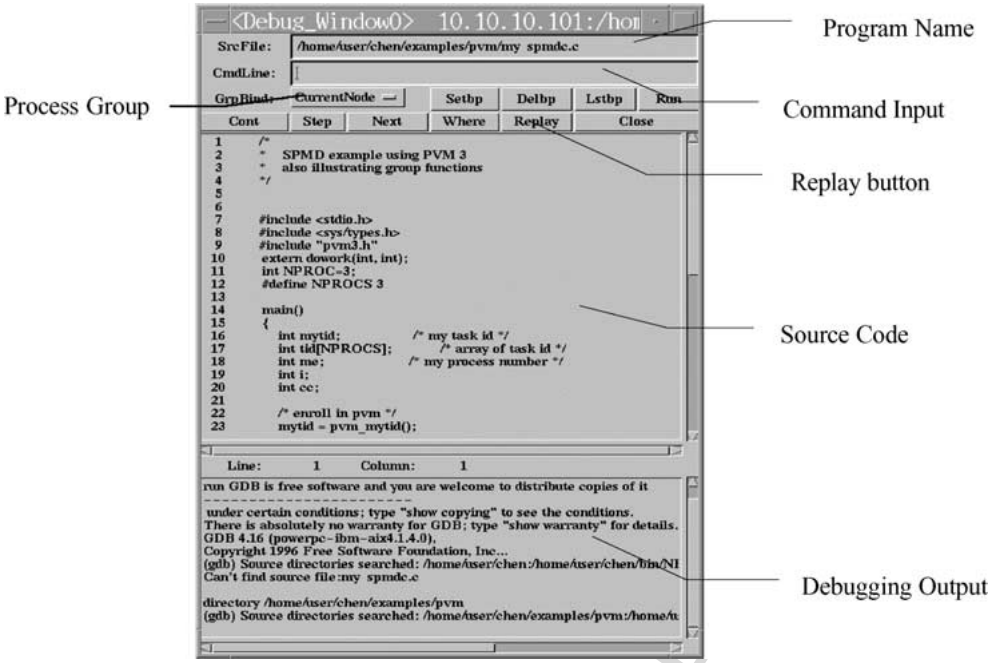


Figure 8. Debugging window.

Start: Choosing a target program and starting the debugging process;
GroupManager: Creating some process groups;
Clear: Clearing all windows and processes of the last debugging;
Return: Exiting the debugging environment.

After debugging a parallel program is finished, the main window of the CDB should not be exited. The window can be reusable.

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 to be configured again unless reconfiguration is needed.

### *4.2. Choosing a target program*

The CDB 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 or Unix) and the types of programming languages (such as C, Fortran) as shown in figure 6. Here, Unix means a general sequential program that can be run on Unix platforms, i.e., the CDB also supports debugging sequential C or Fortran program. The number of processes shown in figure 6 is only valid for executing MPI target program after choosing the program type as MPI + C/Fortran.

### *4.3. Creation of a process group*

The CDB provides the function of process group management. If some processes are set as a group, for example, as shown in figure 7, the group1 includes the logical process 0, 1, and 2. The group2 includes the logical process 1 and 2. Notice that, logical process number 0, 1, and 2 are mapped into


Figure 9. Replay dialog window.

Process Button P0, P1, and P2 shown in figure 5, respectively. Then it is valid for all processes of group1 to send debugging commands to the group1. Therefore, using a debugging command can control all processes of the same group. The CDB can use the global condition control to efficiently monitor the real-time processes of a program execution.

### *4.4. Starting the debugging*

The distributed debugger CDB 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 as shown in figure 8. The user can input or modify the path of his source code, and type a debugging commands or click a debugging command button. The CDB provides the debugging window of each process (but it is not necessary) and sends some processes a debugging command to let them to do it, and displays the execution results of these processes on the debugging windows.

The CDB provides a replay function to treat the nondetermination of parallel execution process based on event records as shown in figure 3 and 9 in order to replay the whole debugging procedure for finding some hidden errors. During the record phase the relevant ordering information about occurring events in a program's execution is stored into a tracefile.

These traces are used to perform trace-driven replay of the program under the constraint of an equivalent execution.

The Process Group Menu shown in figure 8 includes CurrentNode (default option), All Processes, and some user-defined groups. If the option is CurrentNode, all debugging operations are only valid for the current process. If the option is All Processes, then all debugging operations are valid for all processes. If the option is a group named g1 by a user, all debugging operations are valid for all processes of the group g1.

### *4.5. 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 CDB automatically terminates 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 CDB provides a function to automatically clear various debugging "garbages". For example, the "Clear" button shown in figure 5 is for this function.

### *4.6. Exiting the CDB*

This means exiting the whole CDB debugging environment. For example, the "Return" button shown in figure 5 is for this function.

## 5. Case studies: debugging message-passing programs

The CDB is a distributed debugger based on the sequential debugger dbx or gdb for cluster systems. It supports debugging not only SPMD or MPMD (C or Fortran) PVM programs, but also C or Fortran MPI programs. In the following subsections, we shall present how the CDB efficiently supports debugging MPI and PVM programs.

### *5.1. Support for debugging MPI programs*

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

#### *5.1.1. Modifying MPI source codes*

Do not need any modification of the MPI source code.

#### *5.1.2. 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.

#### *5.1.3. Debugging*

When users start to use the CDB to debug a MPI program, they need to input the debugging program name in the main interface of the CDB, next the CDB 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. It 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 from a node and the node's IP address, regarding 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 return it to the local node and displays it in its debugging window.

At this time, the user may use the CDB'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.

#### *5.1.4. An example*

Figure 10 shows a C language 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 figure 11, only the button P1 and P2 are clicked, therefore, only the debugging windows Debug_window1 and Debug_window2 are shown.

### *5.2. Support for debugging PVM programs*

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

#### *5.2.1. 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 CDB, the shell file is modified so that when creating a new process, in fact, it executes the *Slave Server* pro-

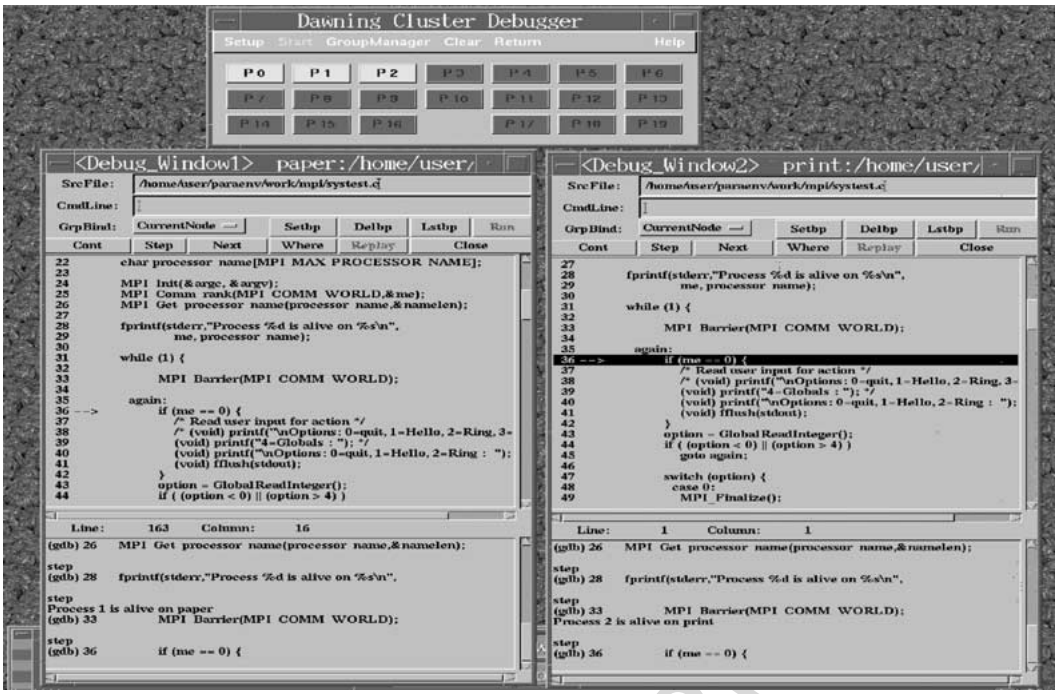


Figure 10. The CDB views for debugging MPI programs.

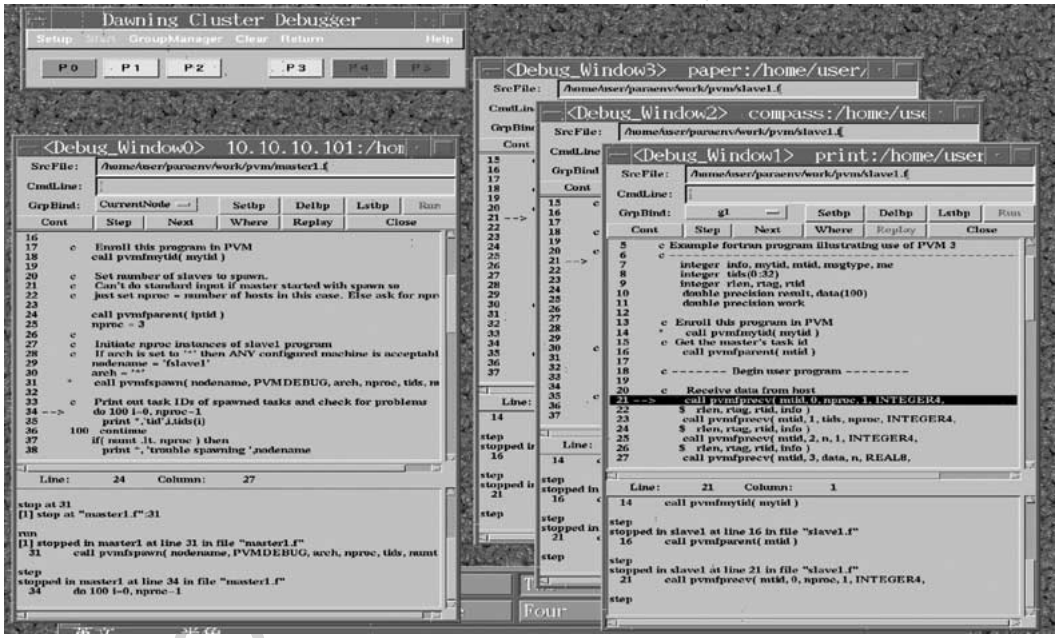


Figure 11. The CDB views for debugging PVM programs.

gram. 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.

*5.2.2. Modifying PVM source codes*

In PVM programs, the users only specify PvmTaskDebug in pvm_spawn(), and 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.

*5.2.3. 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.

*5.2.4. Debugging*

When users start to use the CDB to debug a PVM program, they need to input the debugging program name in the main interface of the CDB, then the CDB 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 from a node and the node's IP address, regarding 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 CDB'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.

*5.2.5. An example*

Figure 11 shows 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 10, these buttons of the CDB's main interface link 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.

## 6. Conclusions

This paper discussed the current debugging techniques and existing tools, and presented the design and implementation of a Java-based scalable distributed debugger CDB for cluster computing, which supports debugging message-passing programs such as PVM and MPI programs. We developed the distributed debugger by extending the functions of current existing sequential debuggers, and used Java to implement the overall interfaces of the CDB so that our distributed debugger achieved the three general goals identified by High Performance Debugging standard concerning parallel and distributed debuggers. It is an attempt to combine Java with sequential debuggers to reach portability and easy to use and learn. Many works will be further done, such as the support for debugging HPF and OpenMP programs, and the support for Microsoft NT cluster system.

## Acknowledgements

The authors would like to acknowledge the comments and suggestions of the reviewers and editor-in-chief Salim Hariri.

## References

[1] R.F. Brender, J.E. Nelson and M.E. Arsenault, Debugging optimized code: Concepts and implementation on DIGITAL Alpha systems, Digital Technical Journal 10 (December 1998).
[2] D. Cheng and R. Hood, A portable debugger for parallel and distributed Programs, in: *Proc. of Supercomputing'94*, November 1994. See also http://science.nas.nasa.gov/Groups/Tools/Projects /P2D2/.
[3] S. Damodaran-Kamal, Xmdb Version 1.0 User Manual 1.2, Los Alamos National Laboratory (1995).
[4] Etnus Inc., TotalView Debugger, http://www.etnus.com/.
[5] J.M. Francioni and C.M. Pancake, High Performance Debugging Standards Effort, http:/www.ptools.org/hpdf/draft/article.htm.
[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine – A Users's Guide and Tutorial for Networked Parallel Computing* (The MIT Press, 1994).
[7] W. Groupp and E. Lusk, User's Guide for MPICH, a Portable Implementation of MPI, Argonne National Laboratory, USA (1994).
[8] M. Hao, HP's distributed debugger, NAS New Technology Seminar, March 1994.
[9] M.T. Heath and J.E. Finger, Paragraph: A Tool for Visualizing Performance of Parallel Programs, ParaGraph User's Guide, December 1994.
[10] High Performance Debugging Forum, HPD (High Performance Debugging) Version 1 Standard: Command Interface for Parallel Debuggers, http:/www.ptools.org/hpdf/draft/, September 1998.
[11] R. Hood, The P2D2 project: Building a portable distributed debugger, in: *Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996.
[12] IBM Corporation, IBM AIX Parallel Environment: Programming Primer, Release 2.0, 1994.
[13] IBM Distributed Debugger for Workstations, http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/olt/index.html. See also the URL http://www.cineca.it/manuali /sp3/idebug/.
[14] D.C.P. LaFrance-Linden, Challenges in designing an HPF debugger, Digital Technical Journal, 29 January 1998.
[15] S.S. Lumetta, Mantis: A debugger for the split-C language, University of California at Berkley, Technical report #CSD-95-865 (1995).
[16] J. May and F. Berman, Designing a parallel debugger for portability, in: *Proc. of IPPS'94*, April 1994.

[17] C.M. Pancake and R.H. Netzer, A bibliography of parallel debuggers, ACM SIGPLAN Notices 28(12), *ACM/ONR Workshop on Parallel and Distributed Debugging* (1993). See also http://www.cs.orst.edu/~pancake/papers/biblio.html.
[18] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons and R. Title, A scalable debugger for massively parallel message-passing programs, *IEEE Parallel and Distributed Technology* 2(2) (Summer 1994).
[19] R. Stallman and C. Support, Debugging with GDB, Cygnus Solutions, Inc. (1994).
[20] Sunsoft, Inc., *Solaris Application Developer's Guide* (1997).
[21] Think Machines Corporation, Prism 2.0 Release Notes, May 1994.
[22] X. Wu, Q. Chen, X. Hu, Y. Hu, M. Zhu and J. 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).
[23] X. Wu, *Performance Evaluation, Prediction and Visualization of Parallel Systems* (Kluwer Academic Publishers, Boston, 1999).
[24] XPDB, http://www.informatik.uni-stuttgart.de/ipvr/as/projekte/grids/xpdb/xpdb-e.html.

**Xingfu Wu** received his Ph.D. degree in computer science from Beijing University of Aeronautics and Astronautics in 1997. He worked in National Research Center for Intelligent Computing Systems and High Performance Computers (NCIC), Institute of Computing Technology, Chinese Academy of Sciences as a postdoctoral researcher during the academic year 1997–1998, and leaded the parallel programming environment group of Dawning2000 project, which was a main project of Chinese 863 Hi-Tech Programme. He worked in Department of Computer Science, Louisiana State University as a visiting Assistant Professor during the academic year 1998–1999. He has worked in Department of Electrical and Computer Engineering at Northwestern University as a postdoctoral researcher since 28 October 1999. He is a member of IEEE Computer Society, ACM (Association for Computing Machinery) and New York Academy of Sciences. His current research interests are database-driven performance analysis systems and web design, parallel and distributed computing, performance evaluation, prediction and visualization of parallel systems, parallel programming environments and tools. Dr. Wu's monograph: *Performance Evaluation, Prediction and Visualization of Parallel Systems* was published by Kluwer Academic Publishers (ISBN 0-7923-8462-8) in 1999.
E-mail: wuxf@ece.northwestern.edu

**Qingping Chen** received her M.S. degree in computer science from University of Science and Technology of China in 1998. Her research interests are parallel programming environments and tools, and Java programming. Currently she works in Anhui province, PR China.

**Xian-He Sun** received his Ph.D. degree in computer science from Michigan State University. He was a staff scientist at ICASE, NASA Langley Research Center and was an Associate Professor in the Computer Science Department at Louisiana State University (LSU). Currently he is an Associate Professor and the Director of the Scalable Computing Software laboratory in the Computer Science Department at Illinois Institute of Technology (IIT), and a guest faculty at the Argonne National Laboratory. Dr. Sun's research interests include parallel and distributed processing, software system, performance evaluation, and scientific computing. He has published intensively in the field and his research has been supported by DoD, DoE, NASA, NSF, and other government agencies. He is a senior member of IEEE, a member of ACM, New York Academy of Science, PHI KAPPA PHI, a partner of the Esprit IV APART (Automatic Performance Analysis: Resources and Tools) working group, and has served and are serving as the chairman or on the program committee for a number of international conferences and workshops. He received the ONR and ASEE Certificate of Recognition award in 1999 and the best paper award from the International Conference on Parallel Processing (ICPP01) in 2001.