Debugging Message Passing Programs Using Invisible Message Tags*

Roland Wismüller

Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik der Technischen Universität München
D-80290 München, Germany
Tel.: +49-89-289-28243, Fax: +49-89-289-28232
E-mail: wismuell@informatik.tu-muenchen.de
WWW: http://wwwbode.informatik.tu-muenchen.de/~wismuell

Abstract. Source level debuggers for parallel PVM or MPI programs currently offer good support for debugging multiple processes, however, they still lack adequate mechanisms for debugging message passing errors. In this paper, we present a new concept called *message breakpoints*, which allows to follow the information flow between processes. We also show how these breakpoints can be implemented very efficiently by using invisible message tags and will demonstrate other applications of this technique in the context of distributed event detection.

1 Introduction

One of the major tasks during the process of software debugging is to follow a program's data and information flow in order to find out why it behaves the way it does. This is usually done by stopping the program at interesting points and executing it step by step while watching the changes in program state, e.g. the values of variables. This process is supported by source level debuggers [12], which now are also available for parallel programs. These debuggers offer commands e.g. to stop, inspect and single-step a running program. Some of them allow to work on just one process with a single command (e.g. p2d2 [7] and totalview [1]), while others allow to execute commands for sets of processes (e.g. prism [14] and DETOP [11, 17]). With the High Performance Debugging Forum[2], there is even an effort to standardize the features of parallel debuggers.

However, in parallel programs data flow also occurs between different processes due to message passing. So if the process, you are currently looking at with a debugger, sends a message, you often want to know how this message is processed by its receiver. This means that the debugger somehow has to stop the receiver immediately after it has read the message. Today's parallel debuggers

* This work is partly funded by *Deutsche Forschungsgemeinschaft*, Special Research Grant SFB 342, Subproject A1.

[2] See `http://www.ptools.org/hpdf/`

do not adequately support this task. The only thing you can do is to set breakpoints at the programming library's receive routine(s) in the receiver process. However, this only works properly if:

1. You can determine the process (or the processes in case of a multicast or broadcast) that will receive the message in advance. This may not be possible when the programming model uses indirect communication (e.g. via mailboxes) or offers broadcasts to dynamic groups (as PVM does).
2. The communication is unbuffered (i.e. based on a rendez-vous) and there is no possibility for alternative receives (such as a `select` call or the sender wildcards in the PVM receive routines). Only in this case, the next message read by the receiver process is always the one you want to watch.

Obviously, at least one of these preconditions is not satisfied for virtually every existing message passing library. Thus, watching the information flow between processes is very tedious, since you have to stop the program at *any* receive operation and look whether it actually reads the message you are interested in.

In the following section, we will propose new debugger commands to ease this job and explain how these commands can be implemented efficiently by invisible message tags. The work presented here is somehow related to previous work in distributed event detection [4, 10] (see [13] for an excellent summary). However, our work is motivated by a practical problem, not by theoretical aspects. The proposed solution is also more intuitive for users and much more efficient to implement. Nevertheless, in Section 3 we will show that invisible message tags can to some extent also be used for distributed event detection. Section 4 finally presents the framework for our implementation of the proposed techniques.

2 Message Breakpoints

In the above scenario, the user can be strongly supported, if the debugger offers a *message breakpoint* with the following semantics: If you set a message breakpoint in a process P1, the receiver(s) of the next message sent by P1 will be stopped immediately after they have received that message. As an extension, two very useful single stepping commands can be implemented on top of this breakpoint primitive:

1. A "step over send" command can be provided as a simple combination of a message breakpoint and a traditional step command. If the executed statement is a send call, this command will result in the sender to be stopped right after the send call, and the receivers right after they read the message sent. Thus, you can examine how the message is processed while still having access to the variable values etc. used to compose the message.
2. The combination of a message breakpoint with a traditional breakpoint on the send function(s) of a single process leads to a single stepped execution at the granularity of message passing. With a suitable graphical user interface, you can watch the places in the source code where the sends and

the corresponding receives are located and thus check the correctness of the program's communication protocols.

Message breakpoints can be easily implemented by adding a tag to the message being sent, which is invisible for both the application and the message passing library. Since the tag only indicates whether a process receiving that message should be stopped, a single bit per active breakpoint is sufficient. Thus, message breakpoints can be implemented very efficiently.

For PVM, we were able to implement the basic mechanisms for message breakpoints even without increasing the size of the transfered messages by using otherwise unused bits in the message header. One field of this header specifies the encoding used for the message [3] and needs only three bits, leaving space for 29 different message breakpoints, which is more than enough in practice. Similar situations are expected to occur also in other message passing libraries.

Using these bits requires to restore them to their original value before the corresponding field in the message header is accessed by the programming library. This must be done without modifying the source code of the library, since users are extremely reluctant to use such special versions of programming libraries. For PVM, we have achieved this goal by binary wrapping [2] an internal function of the library with a small routine clearing the bits used for the message breakpoints. This wrapping step is done during the linking of the application code, thus recompiling the program is not necessary. The overhead of this (static) instrumentation is extremely small: it only adds a few machine cycles ($55ns$ on a SparcStation 5) to each receive call. In future implementations we plan to use a dynamic instrumentation interface [6, 5], which allows to patch the library during runtime. This will eliminate the need of a special link step.

The best solution, however, would be to have services for setting and reading these kinds of invisible message tags in the common message passing libraries.

3 Distributed Event Detection

In some situations in parallel debugging it is useful to have the ability to define breakpoints that are triggered not just by a single event, but by a combination of events in different processes. This is especially true when processes access some common, external object. As an example, consider two processes on different hosts that access a common file, either via a parallel file system library or simply via NFS. During debugging, you may be interested in situations where operations performed by these processes occur in an illegal order or may even be unsynchronized. As Schwarz and Mattern point out in [13], the key to this so called *distributed event detection* is the causality relation. Informally, two event occurrences a and b satisfy this relation (denoted as $a \rightarrow b$) if a happened before b due to synchronization constraints (and not just by chance). More formally, the relation $\rightarrow$ is the smallest relation, such that:

1. if a and b occur in the same process and a occurs before b, then $a \rightarrow b$

2. if a is the sending of a message and b is the receipt of that message, then $a \to b$
3. if $a \to c$ and $c \to b$, then $a \to b$

It is called causality relation, since $a \to b$ holds if and only if a can causally affect b, either because a occurs before b in the same process, or because there are messages transferring information from a to b. As an example, consider the time line diagram in Fig 1. Here $a \to b$ and $a \to d$ hold, while $a \to c$ does not.

In order to detect event occurrences in a program execution that satisfy the causality relation, current approaches are based on a virtual time, which requires to add time stamps to all messages [13]. For static systems, these time stamps are vectors of a length equal to the number of processes, for programming models allowing dynamic process creation, even more complex tree structures have to be used [15]. These approaches are very general, since they can detect relations between arbitrary event occurrences, however, their overhead is also very high.

Using invisible message tags, causal relationships with respect to a fixed event occurrence a can be detected with a much lesser overhead. The idea is to have a tag for each process that is set when the process can be causally influenced by a predefined event occurrence a. These tags are computed in the following way:

1. When the occurrence of a is detected, the process where a happened is being tagged.
2. After a process has been tagged, all messages it sends are also tagged.
3. When a tagged message is received, the receiver process is being tagged.

From the definition of $\to$ given above, it is immediately clear that for any event occurrence b $a \to b$ holds, iff the process containing b is tagged when b occurs. Thus, when the debugger (or more precisely: its distributed monitoring system) detects b in a tagged process, the breakpoint is triggered, i.e. the program is stopped. This is depicted in the time line diagram on the left of Fig. 2, where the situation that a process or message is tagged is indicated by a dashed line next to it: At some point where the application was stopped (indicated by the vertical curved line), a breakpoint has been defined and the program has been continued. When a is detected later, P1 is being tagged. Thus, a tag is added to the message sent by P1, and process P2 is being tagged when it receives that message. When the first occurrence of b is detected in P2, the process is not yet tagged, so nothing will happen. At the second occurrence of b, P2 is already tagged, thus this time the breakpoint is triggered.

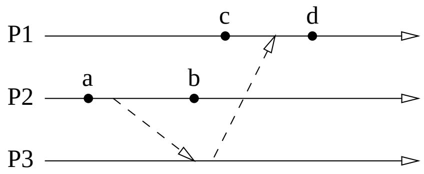

Fig. 1. Examples of causality relations: $a \to b$ and $a \to d$ hold, $a \to c$ does not.

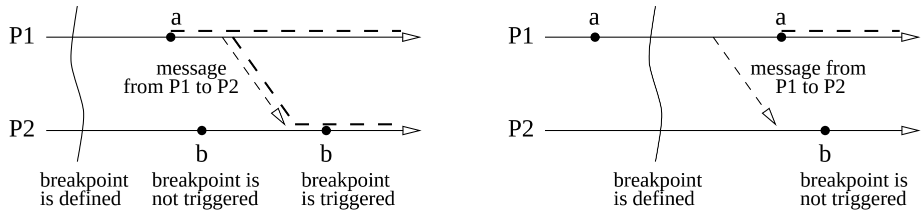

Fig. 2. Detecting a causality relation.

A semantical difference to an implementation using time stamps arises when a already has happened when the breakpoint is being defined as is illustrated on the right of Fig. 2. Time stamp methods (e.g. [4]) are able to take this occurrence of a into account, since they determine and store time stamps for the most recent occurrence of any monitored event. In the example, when a is already monitored due to another active breakpoint, (the monitor attached to) process P1 will generate a time stamp already for the first occurrence of a and will send that information on a to (the monitor attached to) P2. At the occurrence of b in P2, the monitoring system compares the time stamps of a and b, concludes that $a \to b$ holds, and thus will trigger the breakpoint. Using the tagging method, however, the breakpoint will not be triggered, since P2 is not tagged when b occurs. This is consistent with standard breakpoints in on-line debuggers, where at any given time only the next future occurrence of the specified event (e.g. a specific source line is going to be executed) is relevant.

Based on the causality relation, other useful relations can be monitored. The concurrency relation $||$ denotes that two events are not ordered explicitly via synchronization, i.e. they may occur concurrently. This relation is especially useful to detect unsynchronized accesses to common objects which may cause nondeterminism due to race conditions. For two events a and b, $a||b$ is equivalent to $(a \not\to b) \wedge (b \not\to a)$.

Breakpoints having a concurrency predicate $a \,||\, b$ as a trigger condition can be implemented by using two bits per message and process, one indicating that a occurred and $b \not\to a$ holds, the other indicating that b occurred and $a \not\to b$ holds. As with time stamp methods, there is the problem that the knowledge about the state of the two terms $a \not\to b$ and $b \not\to a$ is distributed in the monitoring system, so additional communication is necessary to evaluate $a \,||\, b$ (see Fig. 3). This means that with either technique the concurrency relation can only be detected with a certain delay, so the state of the application may change until the breakpoint is finally triggered.

Again, there is a difference in the semantics of the two methods: If e.g. a is specified as “write access to a common object X by process P1” and b as “write access to the same object X by process P2”, the time stamp method will detect any race between write accesses to X by P1 and P2. On the other hand, the method using tags will again only consider the first occurrences of a and b, respectively, after the definition of the breakpoint, so it will only check

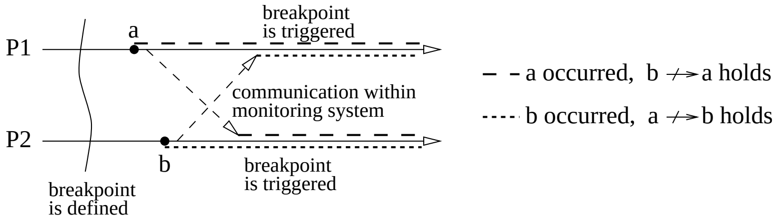

Fig. 3. Detecting a concurrency relation.

whether the next accesses to X by P1 and P2 are unsynchronized. Although this method is obviously more restricted, it can be implemented much more easily and efficiently, and therefore is still a valuable debugger feature when the location of a possible race is known reasonably well.

4 The Parallel Debugger in THE TOOL-SET for PVM

The debugging functionalities presented in the previous sections will be integrated into the parallel debugger of THE TOOL-SET for PVM [16]. THE TOOL-SET project aims to provide an integrated set of interactive and automatic tools for message passing programs on networks of workstations. The basis of THE TOOL-SET is a distributed on-line monitoring system for PVM [8]. The monitoring system is responsible for gathering run-time information about a program's execution, but also for manipulating that execution, if requested. It conforms to the open On-line Monitoring Interface Specification (OMIS)[3] [9] and provides the necessary services for tagging messages and detecting the receipt of a tagged message in a platform-independent way.

Currently, we port the debugger's front end, which connects to the on-line monitoring system, from an existing version called DETOP [11, 17] that has been developed by LRR-TUM within a special research grant (SFB 342) and a cooperation with Parsytec, a German vendor of supercomputers and parallel embedded systems. DETOP is not a simple interface built around a sequential debugger, but a real parallel debugger with special support for parallel programming models. It offers a graphical interface, allowing efficient control and observation of multiple tasks within a single debugger window, but also supports the use of multiple windows for tasks executing different source codes (see Fig. 4). Since the debugger is able to handle different executable files at the same time, support for non-SPMD applications and also for heterogeneous systems is already provided. In order to ease debugging of programs that dynamically create new tasks, tasks can be identified not only by their numerical ID, but also by additional information, e.g. file name, parameters or state. Based on patterns using this information, dynamically spawned tasks can be automatically stopped and added to a debugger window.

[3] see `http://wwwbode.informatik.tu-muenchen.de/~omis`

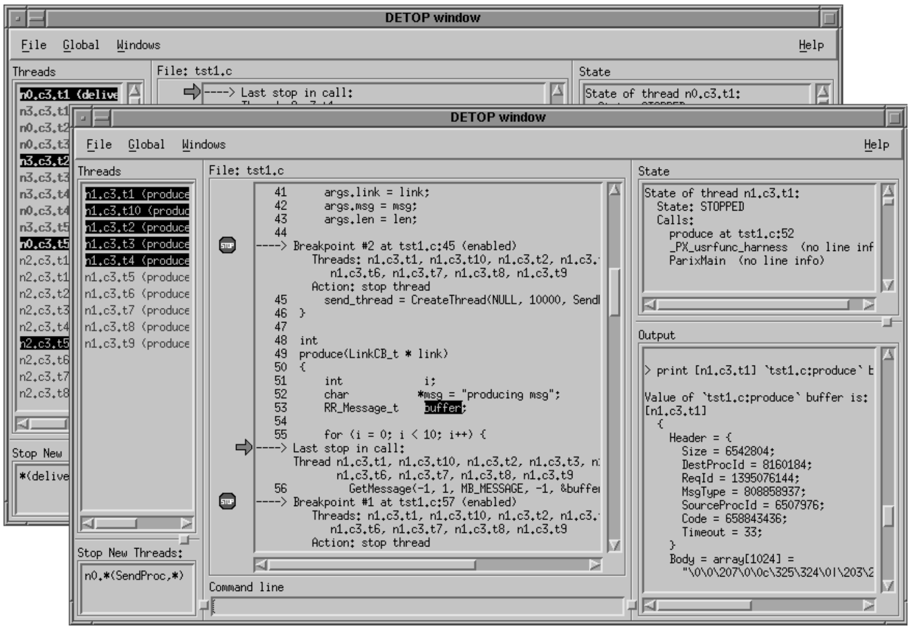

Fig. 4. DETOP graphical user interface.

With the additional functions presented above, the debugger will not only support the observation of all of the processes of a distributed application, but also the monitoring of their interaction.

5 Conclusions

In this article, we have identified a set of valuable debugger commands that support the observation and debugging of the message exchange in parallel and distributed programs. We have shown that these commands can be easily implemented by adding invisible tags to some messages sent by the application program. A prototype implementation for PVM has shown that this is possible without increasing the message size, with a negligible runtime overhead, and without modifying the PVM system.

References

1. BBN Systems and Technologies. *TotalView User's Guide, Version 3.4*, Mar. 1995.
2. J. Cargille and B. P. Miller. Binary Wrapping: A Technique for Instrumenting Object Code. *ACM SIGPLAN Notices*, 27(6):17–18, June 1992.
3. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

4. D. Haban and W. Weigel. Global Events and Global Breakpoints in Distributed Systems. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, volume II, Sortware Track, pages 166–175, 1988.
5. J. K. Hollingsworth and B. P. Miller. Dynamic Instrumentation API (proposed), Revision 0.1. Unpublished Report of Univ. of Maryland and Univ. of Wisconsin, 1996.
`http://www.cs.wisc.edu/~paradyn/dyninstAPI.html`.
6. J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic Program Instrumentaiton for Scalable Performance Tools. In *Proc. Supercomputing'94*, pages 841–850, Washington D.C., Nov. 1994. IEEE.
`ftp://grilled.cs.wisc.edu/technical_papers/dyninst.ps.Z`.
7. R. Hood. The *p2d2* Project: Building a Portable Distributed Debugger. In *Proc. of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 127–136, Philadelphia, Pennsylvania, USA, May 1996. ACM Press.
8. T. Ludwig, R. Wismüller, and M. Oberhuber. OCM – An OMIS Compliant Monitoring System. In A. Bode, J. Dongarra, T. Ludwig, and V. Sunderam, editors, *Parallel Virtual Machine – EuroPVM'96*, volume 1156 of *Lecture Notes in Computer Science*, pages 81–90, Munich, Germany, Oct. 1996. Springer Verlag.
`http://wwwbode.informatik.tu-muenchen.de/~wismuell/pub/europvm96b.ps.gz`.
9. T. Ludwig, R. Wismüller, M. Oberhuber, and A. Bode. An Open Interface for the On-Line Monitoring of Parallel and Distributed Programs. *Intl. Journal of Supercomputer Applications and High Performance Computing*, 11(2), Summer 1997. to appear.
10. B. P. Miller and J. D. Choi. Breakpoints and Halting in Distributed Programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316–323, Los Alamitos, California, 1988. CS Press.
11. M. Oberhuber and R. Wismüller. DETOP - An Interactive Debugger for PowerPC Based Multicomputers. In P. Fritzson and L. Finmo, editors, *Parallel Programming and Applications*, pages 170–183. IOS Press, Amsterdam, May 1995.
`http://wwwbode.informatik.tu-muenchen.de/~wismuell/pub/zeus95.ps.gz`.
12. J. B. Rosenberg. *How Debuggers Work*. John Wiley & Sons, 1996.
13. R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Technical Report SFB124-15/92, Universität Kaiserslautern, 1992.
14. 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*, pages 50–56, Summer 1994.
15. A. Weininger. *Verfahren zur Behandlung von Nichtdeterminismus in parallelen Systemen*. PhD thesis, Munich University of Technology, Munich, Germany, 1994.
16. R. Wismüller, T. Ludwig, A. Bode, R. Borgeest, S. Lamberts, M. Oberhuber, C. Röder, and G. Stellner. The Tool-set Project: Towards an Integrated Tool Environment for Parallel Programming. In *Proceedings of Second Sino-German Workshop on Advanced Parallel Processing Technologies, APPT'97*, Koblenz, Germany, Sept. 1997. to appear.
17. R. Wismüller, M. Oberhuber, J. Krammer, and O. Hansen. Interactive debugging and performance analysis of massively parallel applications. *Parallel Computing*, 22(3):415–442, Mar. 1996.
`http://wwwbode.informatik.tu-muenchen.de/~wismuell/pub/pc95.ps.gz`.