Debugging Point-To-Point Communication in
MPI and PVM

Dieter Kranzlmüller, Jens Volkert

GUP Linz, Johannes Kepler University Linz
Altenbergerstr. 69, A-4040 Linz, Austria/Europe
kranzlmueeller@gup.uni-linz.ac.at
http://www.gup.uni-linz.ac.at:8001/

Abstract. Cyclic debugging of nondeterministic parallel programs requires some kind of record and replay technique, because successive executions may produce different results even if the same input is supplied. The NOndeterministic Program Evaluator NOPE is an implementation of record and replay for message-passing systems. During an initial record phase, ordering information about occurring events is stored in traces, which preserve an equivalent execution during follow-up replay phases. In comparison to other tools, NOPE produces less overhead in time and space by relying on certain properties of MPI and PVM. The key factor is the non-overtaking rule which simplifies not only tracing and replay but also race condition detection. In addition, an automatic approach to event manipulation allows extensive investigation of nondeterministic behavior.

1 Introduction

Debugging of parallel programs is concerned with location and correction of computational bugs and run-time failures in parallel code. The set of possible errors includes all bugs known from sequential debugging as well as additional obstacles introduced by process interaction (e.g. deadlocks [11]). As a consequence, a parallel debugger has to provide special capabilities to cope with communication errors and synchronization bugs.

A critical example are race conditions [9]. These are places in the code, where different event-ordering may occur, leading to different runtime-behavior. Thus, their existence introduces nondeterminism, which manifests itself through variations in program results during successive program runs, even if the same input data are supplied. The main problem of races is the "irreproducibility effect" [11], which means, that program executions cannot be reproduced easily. Furthermore, it is not ad hoc possible to guarantee, that repeated executions of the program reveal all possible results and errors. Both problems are equally important and have to be addressed in order to produce reliable software.

The focus of this paper is the solution of NOPE, the NOndeterministic Program Evaluator. NOPE is integrated in the Monitoring And Debugging environment MAD [4], a toolset for error detection and performance analysis of
message-passing programs. Originally intended for the nCUBE 2 multiprocessor, MAD has now been transferred to the standard Message-Passing Interface MPI [8], with preparations for the Parallel Virtual Machine PVM [3]. During the design of the module NOPE, extensive studies of the communication libraries MPI and PVM were required, which influenced the resulting implementation through certain simplifications. The outcome is a record and replay mechanism, which produces less overhead in time and space than previous approaches.

The paper is organized as follows. The next section describes the relation between nondeterministic program behavior in message-passing programs and the debugging cycle in general. Afterwards an introduction of message-races in MPI and PVM is presented, which leads to the simplifications as implemented in NOPE. Section 4 describes the operation of NOPE and its integration into the debugging cycle. Conclusions and an outlook on future goals of this project summarize the paper.

2 Nondeterminism in Message-Passing Programs

The traditional approach to error detection is cyclic debugging. The program is executed again and again with the same input, allowing the user to gain more insight about program states and intermediate results with the debugger. The basic idea is as follows [12]: determine the results $y$ of the arbitrary function $F$ with a given input $x$. If $x$ is a valid input for $F$, and the input/output pair $(x|y)$ does not represent a correct computation of $F$, the error is located somewhere in function $F$. For that reason, $F$ is split up into subfunctions $f_0, f_1, \ldots, f_n$ and repeated executions of $F$ are used to determine the correct states between these subfunctions by analyzing intermediate results.

A simple sequential program is sketched in figure 1. The function $F$ is split up into $f_0, f_1,$ and $f_2$, which perform the desired computation. The resulting output $y$ is determined by the computation of $F$, the provided input $x$ and the interaction with the system environment. If the input is the same and system-interactions can be reproduced, each iteration of the debugging cycle would yield the same output, and intermediate results can be analyzed with breakpoints and single-stepping.

In parallel programs the situation is more difficult, because subfunctions of $F$ are executed on concurrent processes and all process-interactions are influenced by the computer system. The parallel program of figure 1 is executed on two processes. The function $F$ is split up into $f_0, f_1,$ and $f_2$ on process 0, as well as $g_0, g_1,$ and $g_2$ on process 1. The output is again determined by the computation of $F$, the input $x$, and all system interactions. Since communication and synchronization are carried out by the system, even small changes in the system state (processor speed, scheduling decisions, or cache contents [4]) may trigger different behavior. Unfortunately several of these environmental parameters cannot be controlled by a debugger and thus may lead to changes in the execution path of a program.