Visual Debugging of MPI Applications

Basile Schaeli [1], Ali Al-Shabibi [1] and Roger D. Hersch [1]

[1] Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
{basile.schaeli, ali.al-shabibi}@epfl.ch

Abstract. We present the design and implementation of a debugging tool that displays a message-passing graph of the execution of an MPI application. Parts of the graph can be hidden or highlighted based on the stack trace, calling process or communicator of MPI calls. The tool incorporates several features enabling developers to explicitly control the ordering of message-passing events during the execution, and test that reordering these events does not compromise the correctness of the computations. In particular, we describe an automated running mode that detects potential races and enables the developer to choose which execution path should be followed by the application.

1 Introduction

Parallel applications are subject to errors that do not occur in single-threaded sequential applications. Such errors include deadlocks, when conflicts over resources prevent the application from moving forward, and message races, when changing the order of reception of messages changes the result of the computation. Parallel application debuggers should therefore enable explicitly testing and analyzing such errors and provide multiple abstraction levels that filter and aggregate the large amount of information displayed to the developer.

Several contributions, e.g. [4], [6], focus on record and replay techniques to enable reproducing a race once it has been detected. For instance, Retrospect [4] enables the deterministic replay of MPI applications, but the lack of control on the application execution may force the developer to run its application many times until an error is revealed. To our knowledge, ISP [11] is the only tool that explicitly tests different orderings of events within MPI applications. While it could produce a suitable trace for a replay tool, being able to replay an erroneous execution deterministically is only a first step in identifying a bug. The ability to visualize and to test slightly different executions may help understanding the origin of an error and correcting it.

Full-featured parallel debuggers such as TotalView [10] and DDT [1] support the isolation of specific processes, the inspection of message queues and are able to attach a sequential debugger to remote application instances. The debugger for the Charm++ framework [7] takes advantage of its integration within the Charm++ parallel runtime to provide higher-level features such as setting breakpoints on remote entry points. While these tools provide the developer with detailed information about the running

Proc. of the 15th European PVM / MPI Users' Group Meeting (EuroPVM / MPI), Lecture Notes in Computer Science (LNCS) vol. 5205, A. Lastovetsky et al. (Eds.) Springer Verlag, 2008, pp. 239-247

processes, none of them provides an instantaneous high-level picture of the current state of the application execution.

In previous work, we described a debugger targeting applications developed using the Dynamic Parallel Schedules (DPS) parallelization framework [2]. The parallel structure of these applications is described as an acyclic directed graph that specifies the dependencies between messages and computations. The debugger may therefore display the current state of the graph very naturally and provides the application developer with much information in a compact form. Different event orderings can be explicitly tested by reordering messages in reception queues or by setting high level breakpoints.

The present contribution applies the concepts presented in [2] to MPI applications, and introduces a few MPI specific features. A graphical user interface displays the message-passing graph of the application and provides a high-level view of its communication patterns. Within the message-passing graph, we can hide or highlight MPI calls based on various criteria such as the originating process, the communicator on which the communication occurred, or the source code file or function that generated the call. We propose various types of high-level breakpoints to control the evolution of the participating processes. Execution scenarios that occur only rarely in actual executions can thereby be explicitly tested. Variants may be executed using an interactive replay functionality. The debugger is able to provoke and detect potential conflicts over *MPI_ANY_SOURCE* receives. Possible matches are drawn on the message-passing graph, enabling the developer to decide which execution path must be followed by the application. The debugger also integrates object visualization support for the *autoserial* library [3], which provides MPI function wrappers that are able to send and receive regular C++ objects.

The paper is organized as follows. Section 2 describes the general architecture of the debugger and Section 3 describes features for controlling the application execution. Scalability issues and performance measurements are presented in Section 4. Section 5 discusses directions for future improvements and draws the conclusions.

2 Architecture

The debugging functionality is provided via two independent components. The first, the interception layer, is a library that intercepts the MPI function calls performed by the application using the MPI Profiling Interface (PMPI [5]). When the MPI initialization function *MPI_Init* is intercepted, every process opens a TCP connection to the debugger, a standalone Java program that receives and displays information about the current state of the application.

Processes first identify themselves to the debugger by sending their rank and their process identifier. During the application execution, the interception layer then sends a notification to the debugger for every point-to-point and collective MPI function called. Notifications are also generated for the various *MPI_Wait* and *MPI_Test* functions, as well as for functions creating new communicators. With the exception of the message content, each notification contains a copy of all the parameters of the called function. These parameters may be MPI defined constants, such as *MPI_COMM_WORLD*, *MPI_INT* or *MPI_ANY_SOURCE*, whose actual value is

Proc. of the 15th European PVM / MPI Users' Group Meeting (EuroPVM / MPI), Lecture Notes in Computer Science (LNCS) vol. 5205, A. Lastovetsky et al. (Eds.) Springer Verlag, 2008, pp. 239-247

specific to MPI implementations. The debugger therefore also receives a copy of these constants when the application starts, so as to be able to translate parameter values into human readable form when displaying information to the developer.

Notifications are sent before calling the actual MPI function. Once it has sent a notification, a process suspends its execution and waits for an acknowledgment from the debugger. By withholding specific acknowledgments, the debugger may thus delay the execution of the associated processes while letting the rest of the application execute. Since a process cannot send more than one notification at a time to the debugger, the order in which the debugger receives notifications from a given process matches the order of occurrence of events within that process.

Receive calls that specify *MPI_ANY_SOURCE* as the source of the expected message may potentially match send calls from multiple sources. In this paper, we refer to such calls as *wildcard receives*. Since in the general case the debugger cannot automatically determine which source is actually matched by a wildcard receive, this information is provided separately by the interception layer via a *matched* notification. If the wildcard receive is blocking, the *matched* notification is sent immediately after the reception of the message by the receive function call. For non-blocking wildcard receives, the *matched* notification is sent when an *MPI_Wait* or *MPI_Test* call successfully queries the status of the non-blocking receive. In both cases, the rank of the matched source is read from the *MPI_Status* parameter of the appropriate call.

The user interface of the debugger consists of a single window that provides control elements to influence the application execution, and displays the current status of the application as a message-passing graph. The vertices of the graph represent the MPI calls performed by the application. Unlike most tracing tools that display time from left to right, our representation matches the one used within the MPI standard, where time flows from top to bottom. Vertices associated to notifications from a same process are therefore displayed one below the other, similarly to successive lines of code within a source file.

The debugger draws edges between successive vertices from a same process. It also draws edges of a different color between vertices associated to matching send and receive calls. For this purpose, the debugger maintains one *unmatched sends* and one *unmatched receives* queue. Upon receiving a notification for a send (resp. receive) call, the debugger looks for a matching receive (resp. send) call within the unmatched receives (resp. unmatched sends) queue. If none is found, the incoming notification is pushed at the end of the corresponding queue. When looking for matches, the queues are explored in a FIFO manner in order to respect the FIFO property of MPI communication channels. New vertices and edges are dynamically added to the graph as the debugger receives new notifications from the application. When the debugger receives a notification for a wildcard receive from a process *p*, it stops matching send calls destined to *p* until the reception of the corresponding *matched* notification. For non-blocking wildcard receives, graph updates are therefore delayed until the application successfully queries the status of the receive call.

On Linux, the interception layer is able to determine the stack trace of every MPI call. A panel in the debugger window displays a tree containing the files, functions and line numbers from which the MPI functions were called. Selecting a node of the tree then highlights all the associated vertices in the message-passing graph,

Proc. of the 15th European PVM / MPI Users' Group Meeting (EuroPVM / MPI), Lecture Notes in Computer Science (LNCS) vol. 5205, A. Lastovetsky et al. (Eds.) Springer Verlag, 2008, pp. 239-247

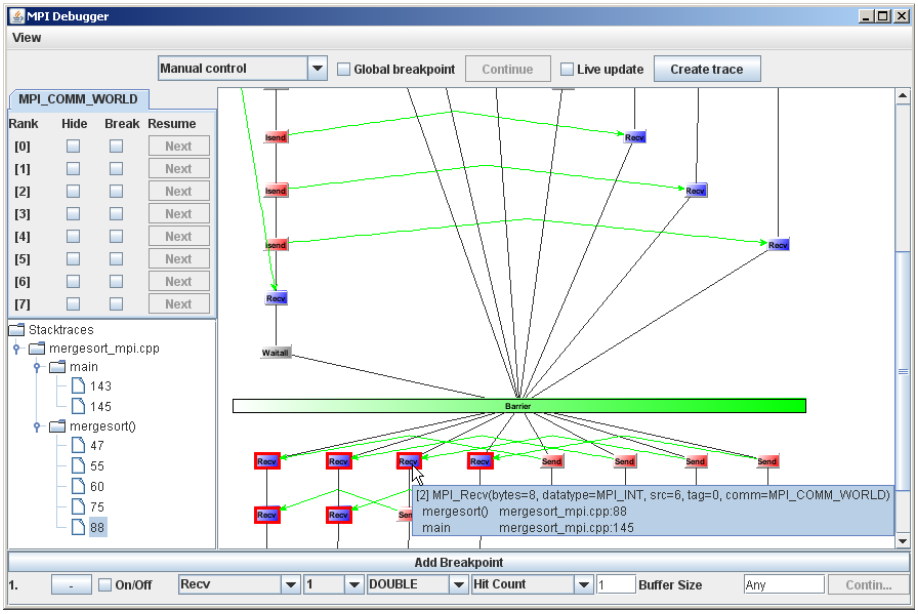

Fig. 1. Debugger window. The left panes contain the list of processes and the stack trace tree. Tooltips display detailed information about MPI calls.

illustrating how and when the selected file or function is used within the application. Another panel displays the list of processes involved in the computation and enables hiding the graph vertices belonging to specific processes. When the application uses multiple communicators, the list of processes belonging to each one of them appears in additional tabs. When switching to a given communicator tab, the developer may choose to display a partial message-passing graph that includes only the vertices associated to MPI calls performing communications on the selected communicator.

We provide the ability to zoom in and out of the graph in order to adapt its level of detail to the needs of the developer. The label and color of every vertex indicates the type of MPI operation executed, and tooltips display detailed information about call parameters, as well as its stack trace if available. Collective operations are grouped into a single vertex and are represented as a rectangle that spans all participating processes. When the developer double-clicks the graph vertex of a suspended MPI call, the debugger attaches a user-specified sequential debugger to the calling application process, and uses the stack trace information to set a breakpoint to the source code line that immediately follows the MPI function call. The debugger then acknowledges the notification, the process is resumed and the new breakpoint is hit, enabling the developer to inspect the application code.

The *autoserial* [3] library provides wrappers around the *MPI_Send* and *MPI_Recv* functions that allow sending and receiving complex C++ objects instead of simple memory buffers. When these functions are used, the interception layer sends the full serialized object to the debugger, which may then display its content using a tree view similar to the ones found in traditional sequential debuggers. For objects to be understood by the debugger, the serialization is performed by a specialized textual