High-Level Parallel Debugging with LGDF2

David C. DiNucci

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 88-007

December, 1988

High-Level Parallel Debugging with LGDF2

David C. DiNucci

Department of Computer Science and Engineering
Oregon Graduate Center
19600 NW Von Neumann Drive
Beaverton, OR 97006 USA
(503) 645-1121

Abstract — **Most approaches to parallel debugging have assumed the worst: that the program to be debugged must be viewed as a black box, with no internal structure useful to the debugger. This document assumes the opposite—that the program was written using Large-Grain Data Flow 2 (LGDF2). Using this as a basis, we explore some high-level debugging techniques that become available, including data flow debugging and instant replay techniques with a minimum of intrusion by the debugger.**

Index Terms — **Parallel Debugging, Instant Replay, Non-determinism, Large-Grain Data Flow, Data Flow Debugging, Program Visualization, High-level Debugging, Parallel Monitoring.**

1. INTRODUCTION

Most approaches to parallel debugging to date have assumed the worst: that the program to be debugged has no structure useful during debugging. Some advocate that (at least) the first step in debugging should be to statically analyze the program to find some inner structure[1] [2]. Simply making this hidden structure visible often brings problems to light, by revealing whether the computer is working with the same structure that the programmer had intended to build. This paper describes the proposed design for a debugger which relies on a programming technique (LGDF2) in which the high-level structure is designed explicitly by the programmer and used as a semantic basis for inter-process communication and control during execution[3]. In addition to having a valuable role in the efficient design and implementation of parallel programs, this intentional and obvious structure is used as a high-level graphic basis for debugging. The user can identify with this structure and use the same reasoning process for debugging as was used in the original design, unlike other parallel programming techniques where a "sequential" program is automatically pre-processed to create a parallel program. The rigid semantics and the simplicity of parallel interactions in the model give the debugger the power to efficiently record, replay, and display execution.

2. The LGDF2 Parallel Processing Model

The program structure, called an LGDF2 dataflow network, is similar to a Petri net, but each transition (called a process) can be a sequential program and each place (called a datapath) can contain data as well as a token-like state. An extra intermediate token state is added to datapaths to account for non-atomic firing of processes. Although the firing condition is very similar to that for Petri nets, a process can selectively remove tokens from inputs and create them on outputs as part of its execution. Extra network notation indicates whether data values can flow to/from a process along an edge, since this data flow is not necessarily in the same direction as "token flow".

A process is typically implemented in a standard sequential language. It cannot maintain any local (hidden) state between executions, and does not use any low-level synchronization or message sending primitives. Instead, only a high-level primitive is required which sets the token state of a datapath and relinquishes the process's access to the datapath's data state. Because the inter-process communication structure is shown declaratively by the network, it can be efficiently implemented on both shared and distributed memory architectures, unifying the way programming and debugging is approached across these architectures.

The model has several properties that aid parallel debugging. While a process is executing, its entire data state is guaranteed to remain unaltered by other concurrent processes. Since a process has a limited amount of input and output during any one execution, its history typically consists of many relatively short executions. Because these processes are deterministic, each execution can be completely described by its input data (i.e., the data present on the datapaths that it can read) and the process code.

The entire scope of effect of any process action is clearly specified in the dataflow network. As a consequence of this and the deterministic nature of the individual processes, the execution of a network can be completely represented by a partial ordering (pomset[4]) of its firing and datapath events. Such a pomset, called an execution graph, efficiently expresses only the essential time/data dependencies of an execution. Equivalent executions, where the same transformations occurred to all input data to create the output data, produce equivalent execution graphs, even though the order of unrelated events may have differed greatly.

3. Non-determinism and Execution Replay

Non-determinism, in the sense that two different execution graphs can result from the same dataflow network and input, can occur only when the network has multiple processes connected to a common datapath in such a way that more than one could fire on a given token state. Though the reachability of a such a conflict token state (in Petri terms) cannot always be ascertained, the connection structure, called a *shared* datapath, is easily recognized syntactically by either a human or a mechanical debugger. If a network contains no shared datapaths, it is deterministic[5] and the execution graph can be constructed (or equivalently, the execution replayed[6]) given only the dataflow network, the input data to the network, and the process code. If the network does contain shared datapaths, an execution graph representing an execution can still be constructed given the above information plus the order in which the processes at shared datapaths fired. This information is easily recorded with a minimum of overhead by an execution scheduler, making possible the replay of any execution during debugging. In fact, this recording can possibly be made efficient enough that it can be always left on, allowing us to bid good riddance to "Heisenbug" parallel debugging effects.

4. A Characterization of Parallel Bugs

Each process in an LGDF2 network, like those in other parallel programs, obtains INPUT data and produces OUTPUT data. The process's computation (COMPUTE) can be thought of both as mapping the INPUT to OUTPUT and as enforcing an ordering on events (dependent on INPUT) by waiting on external events (INEVNT) and posting new events (OUTEVNT). Communication and control paths (datapaths) between processes impose a mapping between OUTPUT/OUTEVNTs of one process and INPUT/INEVNTs of another.

The source of a parallel bug can only be an incorrect COMPUTE or incorrect design of the inter-process communication and control structure, but a process may also malfunction due to incorrect INPUT or a missing, extra, or badly timed INEVNT. The effect of any of these malfunctions can be an incorrect OUTPUT and/or OUTEVNT. These may, in turn be INPUT and/or INEVNT to other processes, thereby propagating, amplifying, and/or masking the error.

5. Proposed Debugging Environment

Because of the amplification effect, it is important to find the error early in the program's execution. Single-stepping and studying each operation until the fault is found is clearly not an efficient method. Our debugger offers a multi-level (hierarchical) approach, where the program is first monitored at the event (INEVNT/OUTEVNT) level, then at the INPUT/OUTPUT level, and finally at the COMPUTE level. When an error appears at any level, the user can descend to the next level and replay the same action there. The program can be viewed from any or all of these levels simultaneously.

Event-level monitoring can locate extra or missing synchronization events in the program. The debugger displays events by showing the high-level dataflow network, animating each firing and change in the datapath "token" state as the execution unfolds. The user has the ability to optionally freeze (i.e., block from performing dataflow events) or release selected processes. With these functions, he/she can coerce the order in which processes will fire or effectively breakpoint and single-step their execution at the event level. If the debugger is in REPLAY mode, it bases the firing order at shared datapaths on the record obtained from the a prior (non-debugged) execution, regardless of the user's use of the freeze/release mechanism.

I/O-level monitoring is used to determine when the data communicated between processes is incorrect. When the user selects datapaths in the above-mentioned network for I/O monitoring, the data therein is displayed in a special window whenever a process capable of modifying its data state finishes with it.

COMPUTE-level monitoring is used to determine where within a process's computation an error occurred. When the user selects a process in the dataflow network, a standard sequential debugger is invoked in a special window when the process fires or attempts to change the token state of a datapath. The user can then set breakpoints, display local values, single-step, etc. One benefit of this approach is that the coupling between this sequential debugger and the parallel debugger is loose enough that any available sequential debugger can be used. The sequential debugger needs no special facilities for debugging non-sequential constructs, since these are taken care of at the higher levels.

Another form of debugging, dataflow debugging, fits logically between the event and I/O levels. It takes two forms - data flagging and data history. Data flagging is for determining the effect of data currently present on a datapath. The data on one or more datapaths can be flagged by selecting them with the debugger. From that point, whenever a process fires which can read the datapath, all datapaths which that process can write also become flagged. A datapath becomes unflagged only when it is overwritten with unflagged data. This type of tracing allows the user to find out why the effects of certain data are/aren't being felt far away in another process. Data history is similar, but in reverse. Here, the user can select a single process and ask where the data for this execution of the process originated from. The debugger ascertains the answer by inspecting the execution graph.

These levels of monitoring and debugging are made more useful by allowing the user to back up and re-execute certain processes while watching from a different "angle" (level). The debugger can accomplish this efficiently by keeping a partial reconstruction of the execution graph. This is then consulted by the debugger to determine when datapath contents need to be recorded to allow the user to re-execute the last N process firings. If the user does indeed re-execute, the reconstructed execution graph is used to simulate a consistent global network state while preventing unneeded re-execution of dependent processes.

6. References

[1] R. N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs", *CACM,* vol. 26, no 5., pp. 362-376, May 1983.

[2] W. F. Appelbe, C.E. McDowell, "Anomaly Reporting - A tool for Debugging and Developing Parallel Numerical Algorithms", *Proc. First Int'l Conf. on Supercomputing Systems,* pp. 386-391, December 1985.

[3] D. C. DiNucci, R. G. Babb II, "Practical Support for Parallel Programming", to appear in *Proc. 1988 Hawaii Int'l Conf. for System Sciences.*

[4] V. Pratt, "Modeling Concurrency with Partial Orders", *Int'l Journal of Parallel Programming,* vol. 15, no. 1, pp. 33-71, Feb. 1986.

[5] G. Kahn, "The Semantics of a Simple Language for Parallel Programming", *Inf. Proc. 74,* pp 471-475, North-Holland, 1974.

[6] T. J. LeBlanc, J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers,* vol. c-36, no. 4, pp. 471-482, Apr. 1987.