A Debugger for Flow Graph Based Parallel Applications

Ali Al-Shabibi Sebastian Gerlach Roger D. Hersch Basile Schaeli

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

ABSTRACT

Flow graphs provide an explicit description of the parallelization of an application by mapping vertices onto serial computations and edges onto message transfers. We present the design and implementation of a debugger for the flow graph based Dynamic Parallel Schedules (DPS) parallelization framework. We use the flow graph to provide both a high level and detailed picture of the current state of the application execution. We describe how reordering incoming messages enables testing for the presence of message races while debugging a parallel application. The knowledge about causal dependencies between messages enables tracking messages and computations along individual branches of the flow graph. In addition to common features such as restricting the analysis to a subset of threads or processes and attaching sequential debuggers to running processes, the proposed debugger also provides support for message alterations and for message content dependent breakpoints.

1. INTRODUCTION

Parallel applications are vulnerable to types of errors to which single-threaded applications are immune. Most of these errors stem from the non-deterministic orderings of events within the application. Common representative of such errors are deadlocks, when conflicts over resources prevent the application from moving forward, and message races, when reordering the order of delivery of messages changes the result of the computation. Parallel application debuggers should therefore provide tools to test and analyze such specific errors. Moreover, the recent advent of large-scale systems requires new solutions that filter, aggregate and preprocess the overwhelming amount of information delivered to the developer.

Several contributions discuss the importance of using multiple abstraction levels for parallel program debugging [13, 14, 16]. They describe interesting parallel debugging concepts, such as process isolation, time-process communication graphs and call graph representations of the underlying parallel program execution.

Other debugging tools explicitly target large-scale systems. One tool [2] focuses on aggregating the textual output of the different processes and another tool [1] aggregates their stack trace to identify processes which, despite being identical, behave differently. In their case study, the authors of [1] use their tool in order to identify a subset of processes that have an erroneous behavior, and then use a distinct full featured debugger to further analyze these processes.

Multiple full-featured interactive parallel debuggers have been described in the literature, e.g. Mantis [17], TotalView [6] and p2d2 [9]. All support the isolation of specific processes, as well as attaching a sequential debugger to remote application instances, which enables breakpoints to be set in individual processes. TotalView also supports the inspection of message queues in MPI programs [4], [21]. Message queue inspection is also available in the debugger for the Charm++ parallel application development framework [11]. In the latter case, the integration with the Charm++ parallel runtime enables higher-level features such as setting breakpoints on remote entry points [18]. While these tools provide the developer with detailed information about the application execution, none of them provides an instantaneous high-level picture of the current state of the application execution.

The detection and debugging of message races received much attention from researchers. However, most work focuses on record and replay techniques to enable reproducing a race once it has been detected [3, 10, 15, 21, 22]. Few proposals explicitly test different message delivery orderings, and none of them has been integrated within a debugger [12, 19].

The debugger described in the present contribution targets applications developed using the Dynamic Parallel Schedules (DPS) parallelization framework [7]. The parallel structure of these applications is described as an acyclic directed graph that specifies the dependencies between messages and computations. By displaying the current state of the graph, the debugger can provide the application developer with much information in a compact form. In order to change the ordering of computations, we allow the reordering of messages that await processing. For this purpose, we provide several types of breakpoints which suspend the execution of specific threads and let messages accumulate in the queues.

Our tool therefore enables controlling the application execution such that execution scenarios that occur only rarely in actual executions can be explicitly tested. In addition to the aforementioned features (isolating processes, attaching sequential debuggers, setting high-level operation breakpoints and inspecting message queues), we describe the following original contributions:

- The content of messages can be modified from within the debugger. In addition, we use knowledge about the structure of the transferred data to create conditional breakpoints relying on specific data values within the messages.
- Our debugger supports the reordering of messages, enabling testing for message races.

International Symposium on Software Testing and Analysis (ISSTA'07), Proc ACM Workdshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD'07), 2007, pp. 14-20

- The graph representation of an application provides an instantaneous feedback of the current state of the execution. While such a representation is natural for data flow languages, it is the first time that this representation is used in the context of parallel application debugging.
- The graph also defines the causality between events occurring during the parallel application execution. We use that knowledge to trace sequences of causally dependent messages, i.e. we study successions of messages and operations along a single branch of the flow graph.

Although the current discussion is carried out in the context of DPS applications, the notions of message content based debugging and of message reordering can be applied to any message-passing programming model.

The rest of the paper is organized as follows. Section 2 presents the Dynamic Parallel Schedules programming model. Section 3 describes the general architecture of the debugger, and Section 4 describes its features. An example of utilization is then provided in Section 5. Issues regarding the scalability of the debugger as well as the future work are mentioned in Section 6. Section 7 draws the conclusions.

2. DYNAMIC PARALLEL SCHEDULES

DPS describes a distributed memory parallel computation as a flow graph composed of serial operations arranged to form an acyclic directed graph. The edges are defined by the message types that transit between operations. The flow graph describes the asynchronous flow of data between operations.

The particular implementation of operations is left to the developer, but each operation must be of one of four fundamental types: *leaf*, *split*, *merge* or *stream*. *Leaf* operations accept a single input and generate a single output message. *Split* operations take one input message and generate one or several output messages. *Merge* operations expect one or several input messages, and generate a single output messages once all expected messages have been received. The fourth operation type, the *stream*, puts no restriction on the number of input and output messages and allows the programmer to refine the synchronization granularity by streaming out new messages as soon as specific groups of incoming messages have been received.

Figure 1 shows the flow graph of a simple parallel merge sort application. It is composed of a custom *split* operation that receives a vector of integers as input and partitions it into parts (SplitVector). A *leaf* operation then sorts the vector parts (Sort), and a *merge* operation aggregates the results into a single sorted vector (MergeVectors). Communications are performed using messages of a single type.

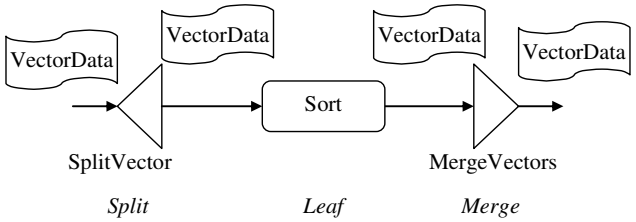

Figure 1. Flow graph describing a basic parallel merge sort computation.

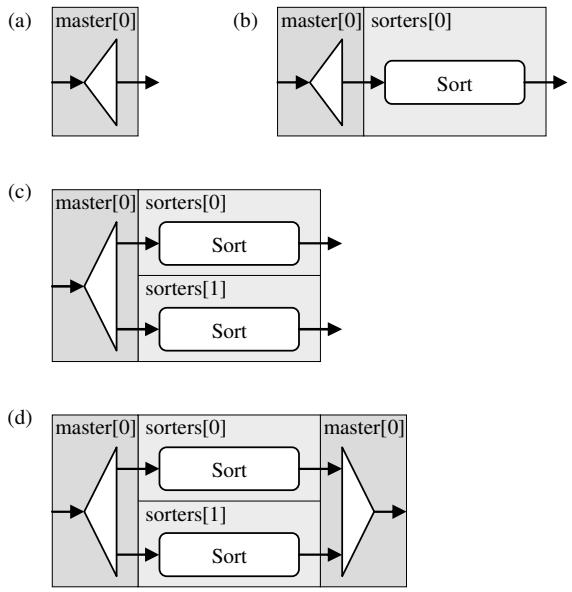

Figure 2. Successive states of the unfolded flow graph of the parallel merge sort application. The *master* thread collection contains a single thread while the *sorters* thread collection contains two threads.

Operations within a flow graph are carried out by threads. Each thread may have an associated state, which is accessible to all the operations it executes. Threads are grouped within thread collections, enabling groups of threads that play distinct roles within the application to be indexed independently. The flow graph together with the thread collections determine the actual deployment of the computations on the compute nodes. This representation, the *unfolded flow graph* of the application, is known only at runtime. Figure 2 illustrates various stages of the application as its flow graph unfolds. The SplitVector and MergeVectors operations run on a thread collection *master* that contains a single thread while the *sorters* thread collection on which Sort operations run contains two threads.

By transferring messages as soon as they are computed, and maintaining queues of arriving messages, execution of DPS applications is fully pipelined and asynchronous. Messages are associated with the thread executing the operations that will consume them. This macro data flow behavior enables automatic overlapping of communications and computations.

DPS applications are written in C++. Messages are C++ objects serialized using an efficient and automatic data serialization mechanism which supports complex data types such as circular linked lists. Each message has a unique identifier [8]. Network transfers are carried out on separate threads, enabling the automatic overlapping of communications and computations. Communications rely on TCP/IP, and we assume that messages are neither lost nor corrupted. However, the DPS framework provides no guarantee about the ordering of delivery of the messages.

3. ARCHITECTURE

The debugging functionality is provided via two independent components. The first, the *debugger*, is a standalone Java program to which the parallel application connects upon startup. It receives and displays information about the current state of the application, which the developer may use to influence future computations.

The second component is the debugging support within the application, which mainly consists in hooks that send notifications to the debugger. These hooks are integrated into the DPS runtime and are enabled at compilation time. All applications may therefore automatically benefit from the debugging functionality without requiring any modification. When the hooks are enabled, an extra parameter added to the application command line specifies the location of the debugger. All instances then open a connection to the debugger upon startup. Figure 3 illustrates a debugging session with an application running on three compute nodes, while the debugger runs on a fourth compute node.

The application communicates several types of information to the debugger. Upon startup, the application first creates the thread collections and the flow graph, and sends both pieces of information to the debugger. During its execution, the application generates a *send* notification for every message it sends. The notification contains a copy of the message. The reception of messages is notified via a *recv* notification containing the message identifier. Additional notifications are sent every time an operation starts or stops processing a message (*opStart* and *opStop* notifications).

Since different operations may run on different threads, each thread is responsible for sending the notifications related to the operations it executes and to the messages that they produce. The *send*, *opStart* and *opStop* notifications must then be acknowledged by the debugger for the thread to continue executing its operations. By holding a specific acknowledgment, the debugger may therefore suspend the execution of the corresponding thread while the rest of the application keeps executing. The debugger also uses acknowledge messages to transmit information back to the application and to influence the future computation steps.

Suspending the execution of threads until the reception of an acknowledgement also guarantees that the debugger receives causally dependent notifications in the correct order. An operation does not send a message over the network before the debugger received and acknowledged the corresponding *send* notification. Messages sent to a given processing thread may be received by different communication threads, which send each a corresponding *recv* notification after adding the message to the thread's pending message queue. Since we use of a single TCP connection between the debugger and each application instance, and since TCP guarantees that data sent over a single connection is not reordered [19], the order of reception of *recv* notifications at the debugger matches the order in which messages are delivered to the thread. Similarly, the debugger cannot receive an *opStart* notification before the corresponding *recv* notification.

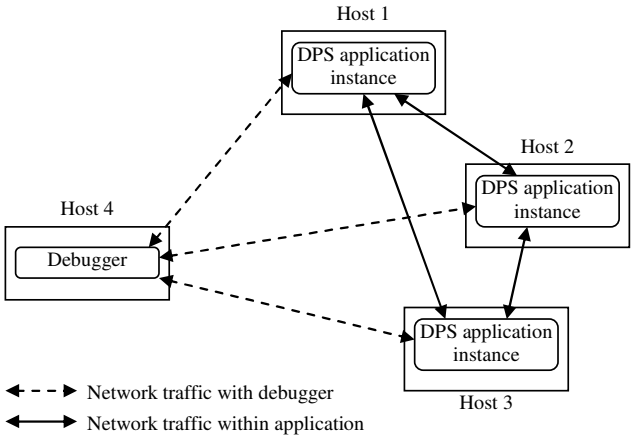

Figure 3. During a debugging session, each application instance opens a connection to the debugger in addition to the ones required for executing the application.

4. FEATURES AND FUNCTIONALITY

The user interface of the debugger is shown in Figure 4. The main area displays the current state of the application in the form of its partially unfolded flow graph as illustrated in Figure 2. Operation names and the thread they run on also appear to identify the different operations. The view is updated every time the debugger receives a notification. Operations are drawn in different colors to indicate their status, such as idle, breakpointed or running.

When the application starts, the debugger holds the acknowledgment for the input message of the first operation in the flow graph. A *Continue* button then enables the developer to resume the application execution. From that point, the default behavior of the debugger is to immediately acknowledge all notifications, allowing the application to continue until completion. Several mechanisms are provided to control the execution. The first is the "Global Step-by-Step" mode. When enabled, the debugger holds all acknowledgments, thereby suspending all threads of the application. Pushing the *Continue* button (Figure 4, top left) then sends one acknowledgment to each thread, which then executes until it sends another notification to the debugger. Steps are therefore taken at the operation level rather than at the instruction level. This mode allows advancing quickly through the execution while still allowing the developer to take action on every notification.

The second execution control mechanism offers a finer grain of control using *operation breakpoints* that break on *opStart* notifications from a particular operation running on a particular thread. These breakpoints are derived from the flow graph and the thread collections sent to the debugger upon application startup: given an operation in the application flow graph and its thread collection, the debugger displays the list of threads within which the operation may run. Each operation breakpoint has an associated box, which may be checked to set the breakpoint and instruct the debugger to hold the acknowledgments of the matching *opStart* notifications. The *Continue* button next to each activated operation breakpoint is enabled when the breakpoint is hit, i.e. when

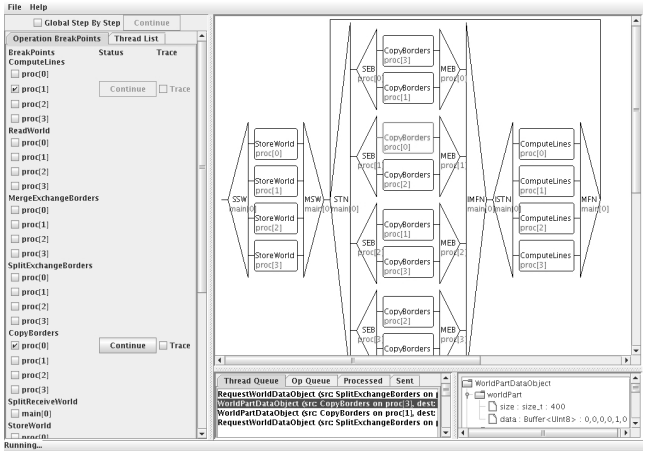

Figure 4. The graphical user inte

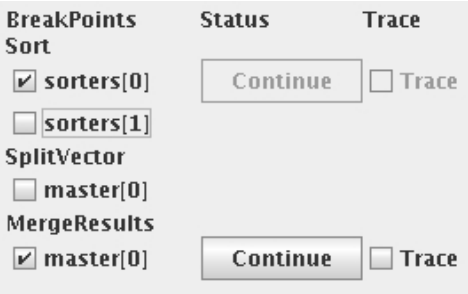

Figure 5. List of operation breakpoints for the merge sort example. Since the thread collection *sorters* contains two threads, the *Sort* operation has two associated operation breakpoints. The *SplitVector* and *MergeResults* operations can be breakpointed on the unique thread of the *master* thread collection. The *Continue* button and *Trace* checkbox are enabled when the breakpoint is hit.

the debugger receives a matching *opStart* notification. Pressing the button then resumes the operation execution. Figure 5 shows the list of operation breakpoints for the flow graph of our merge sort example.

While looking for a bug, it is often useful to study one particular path in the application. That is, when looking at an operation we follow its output message to the successor operation, then to the next, etc., while ignoring what is happening in the rest of the application. Such a behavior is enabled by *tracing* the messages generated by a breakpointed operation. This is done by checking the right-hand side box of an operation breakpoint (Figure 5, right part). When the debugger resumes the execution, the tracing flag is piggybacked on the acknowledgment, causing all the *send* notifications generated by the operation to have a tracing flag set. From this moment onwards, when the debugger receives an *opStart* notification containing the identifier of a traced message, it automatically sets the breakpoint of the triggered operation and holds the acknowledgment. All the successors of an operation in the unfolded flow graph are therefore automatically breakpointed, enabling the developer to navigate through one particular branch of the graph. The trace box can then be unchecked individually for every operation breakpoint, allowing the programmer to focus on the problem he is debugging. This is particularly helpful when a traced message enters a split operation. Since all the outputs of the split will be traced, many operation breakpoints will be set simultaneously.

A second tab displays the list of running application threads, enabling the developer to hide specific threads (or groups of threads such as thread collections) that do not need to be monitored. The operations running on hidden threads are also hidden from the flow graph view, as well as from the list of operation breakpoints. The debugger also ignores (i.e. immediately acknowledges) all notifications from hidden threads.

4.1 Influencing the application execution

Each thread has a FIFO queue that holds the messages awaiting processing (Figure 6, Thread Queue). Such queues may form when a thread is blocked by the debugger, or when the processing time of messages is significant. Reordering the messages contained in the queue therefore changes their processing order on the thread, which allows testing the application for message races. A new modified ordering is transmitted back to the thread along

Figure 6. Message lists. A single message is pending on the *sorters[0]* thread.

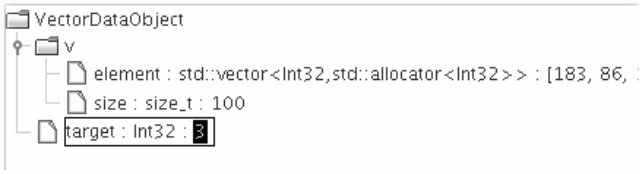

Figure 7. View of a data object used by our parallel merge sort example. The value of *target* is being edited. Integers are stored in an actual C++ standard library vector.

with the acknowledgment, and messages are reordered in the FIFO accordingly. The ability to hide specific threads is a convenient way to use that feature: as the execution of hidden threads is unhindered, all the messages they send to visible threads will accumulate in their pending message queue, providing the developer with an instantaneous view of sets of messages that may race.

When the number of messages is large, the number of possible orderings explodes. However, if communication links are FIFO or if the debugger has additional knowledge about whether operations read or modify the thread state, partial order reduction techniques can be used to provide the developer with a shorter list of orderings [20].

At any moment, the developer may select an operation in the flow graph. This updates the content of the three remaining tabs in Figure 6. The *Op Queue* tab displays the messages from the thread queue that are to be processed by the selected operation. The *Processed* tab lists all the messages which have been received and processed by this operation since the beginning of the application execution. Finally, the *Sent* tab displays all the messages which have been sent by the operation. The queues identify each message by displaying its type, as well as the name of the source and destination operations and threads.

Selecting a message in any one of the lists displays its content in a tree view similar to the ones found in traditional sequential debuggers (Figure 7). This is made possible by using a specialized textual serializer to serialize data objects transferred from the application to the debugger. The textual serialization avoids byte-ordering and internal data representation issues between hosts running the C++ application and the Java debugger. Therefore, the debugger and the parallel application can run on different operating systems and hardware. Since the debugger has no knowledge about the types and structure of the data objects used by the application, the textual serializer adds the necessary typing and variable name information to the data.

Thread states are also stored in serializable data objects identical to the ones transferred between operations [8]. The developer may therefore also retrieve the state of a suspended thread at any time by double-clicking on the corresponding thread in the thread list. The request is piggybacked on the acknowledgment for the pending notification of the selected thread. The thread then sends a

copy of its state to the debugger, which may then display it in a tree view similar to the one displayed in Figure 7.

If the developer selects an operation that is suspended on a *send* notification, the corresponding message is highlighted in the *Sent* list. The *send* notifications are sent before delivering the message to the communication layer. The developer may therefore modify the message from within the debugger before its transfer to the next operation. The modified message is then sent back to the suspended thread together with the acknowledgment. The thread then discards the original message and replaces it with the one received from the debugger. This scheme allows the developer to alter messages so as to modify the behavior of the application. In Figure 7, the *target* field specifies the index of the thread that will sort the partial vector. That *target* value could for instance be set such that it balances the load among the threads. The developer could therefore change that value to test that its application makes no *a priori* assumption about which thread will process the message.

The developer may additionally specify *message breakpoints* by indicating a message type and a particular value in a field of that message. Every time the debugger receives a *send* notification, it tries to match each message breakpoint against the enclosed message. If it succeeds, it simply holds the acknowledgment for the received notification.

4.2 Operation level debugging

While all the features described so far enable controlling the application at the flow graph level, it is often useful to inspect the content of individual operations. For this purpose, double-clicking a breakpointed operation opens a remote connection to the host running the application instance, attaches a user-specified sequential debugger to the running process and sets a breakpoint in the suspended operation (several operations may be inspected simultaneously). Figure 8 illustrates this situation when DDD [5] is used as the sequential debugger.

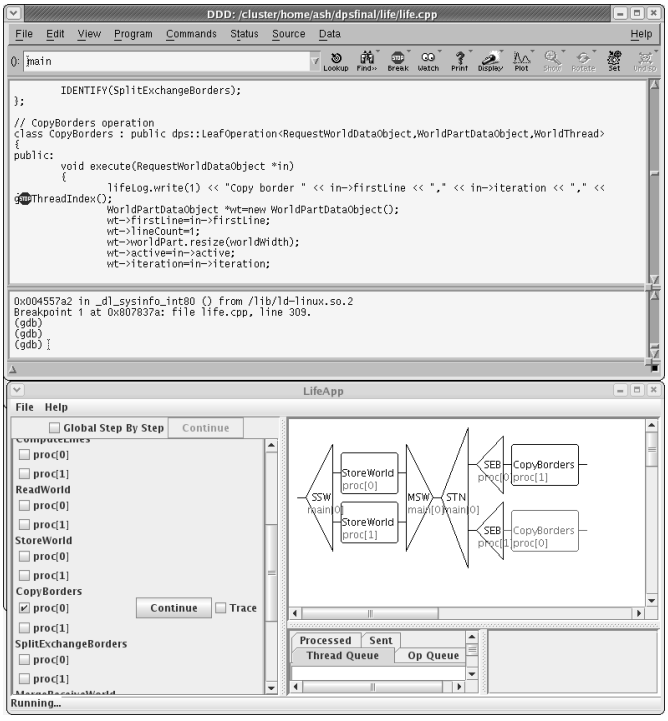

Figure 8. Debugging a single operation with DDD.

5. DEBUGGING EXAMPLE

This section shows how the debugger can be used to discover a message race within a neighborhood-dependent parallel application such as a parallel game of life or a parallel finite element computation. Figure 9 displays the unfolded flow graph of one iteration of the application. The three threads of the *proc* thread collection store each one third of the processed data domain. Each thread sends a "send border" request to its two neighbors. The neighbors send back a copy of their subdomain border (*Send border* operation). The computation of the new state of the subdomain (*Update*) is performed once both borders have been received.

In order to test his application, the developer sets an operation breakpoint on the split operation on thread *proc[1]*. When *proc[1]* is about to start the split operation that sends "send border" requests, it sends an *opStart* notification to the debugger, which hits the breakpoint. As the other threads keep executing, their messages requesting the borders appear in the pending message queue of *proc[1]*. The developer then moves the two requests in front of the input message of the split operation in the queue. Once the new ordering is uploaded to the thread and the messages have been reordered accordingly in the queue, *proc[1]* sends back a new *opStart* notification saying it is about to start executing a *Send border* operation. Since this operation is not breakpointed, and assuming that the Global Step By Step mechanism is not enabled, the notification is immediately acknowledged. The execution of *proc[1]* is suspended once again when both border exchange requests have been processed. At that point, both *proc[0]* and *proc[2]* have received the borders from their neighbors and start updating their part of the domain. From that moment, no matter when *proc[1]* is resumed, its neighbors will process its requests for borders after they have updated their state (Figure 10).

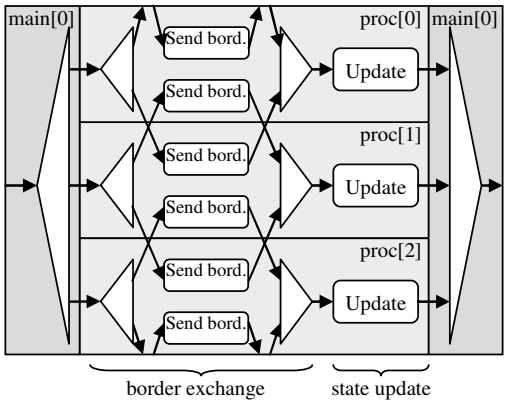

Figure 9. The flow graph of one iteration of a neighborhood dependent parallel computation (the communications between threads *proc[0]* and *proc[2]* wrap around). Each thread of *proc* stores one third of the processed data domain.

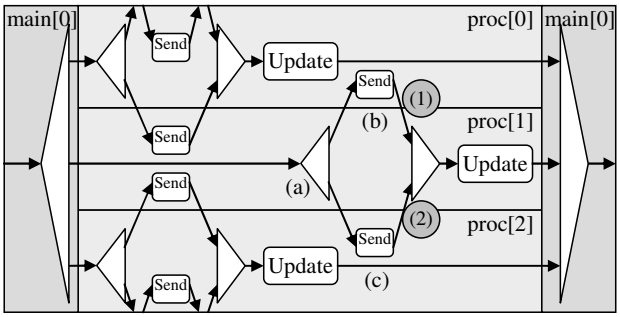

Figure 10. If the processing of the split operation (a) on *proc[1]* is delayed, the state of *proc[0]* and *proc[2]* is read by (b) and (c) after having been updated.

In this example, the flow graph of Figure 9 enforces no synchronization between the border exchange and state update phases, causing a delay in the delivery of a single message to produce an erroneous execution. Since the content of messages (1) and (2) is used to update the state of *proc[1]*, transferring subdomain borders after they have been updated causes an error in the computation. Here the race may be revealed either by looking at the final state of the thread, or by tracing messages starting from the split operation breakpoint and looking at the content of the messages containing the borders received by *proc[1]*.

6. SCALABILITY ISSUES AND FUTURE WORK

Since it must receive, process and acknowledge all the notifications sent by the application threads, the debugger may quickly become a bottleneck when the number of threads grows. Measurements for the flow graph described in section 5 running on 2 compute nodes show that the parallel application runs 170 times slower when the debugger is active and iterations are short (i.e. 22 ms per iteration). The slowdown drops to 2.7 with more intensive computations (i.e. 2 s per iteration). The corresponding slowdown factors are respectively 390 and 9 when the application runs on 8 compute nodes. The overhead is therefore particularly significant for applications performing many short-lived operations.

One possible solution would be to distribute parts of the debugger functionality such as message and operation breakpoint evaluation within the threads, or within debugger servers running on the compute nodes. It would then no longer be necessary to systematically send all messages and all notifications to the debugger, thereby reducing the load of both the debugger and the network. The full functionality would then only be enabled on demand for specific application parts.

Another challenge consists in displaying the flow graph information for a large number of threads. Currently, when hiding specific threads, the programmer also loses the ability to control their execution. This should be overcome by decoupling the processing of notifications from the visibility of the thread. Furthermore, in future releases of the software, we intend to offer the possibility of collapsing a matching split-merge pairs of operations into a single node.

The built-in fault-tolerance mechanism of DPS [8] supports the checkpointing of individual threads. Enabling the developer to take checkpoints of its application may in the future add significant value to the debugger. Taking global snapshots of the current state of the application allows the developer to roll back to it later without reexecuting the whole application. Multiple snapshots could be differentiated using thumbnails of their respective flow graph views. Combined with the reordering or with the modification of messages, this feature would enable interactively testing multiple execution scenarios within a specific part of the whole application. The debugger may then retrieve the thread states after the execution of each ordering and automatically compare and highlight differences between the final states to reveal message races.

Since the flow graph description provide the debugger with a full knowledge about the causality between the executed operations, it would also be possible to undo a specific operation and determine which causally dependent operations must also be undone to maintain a consistent state, thereby providing a finer grain of control while stepping back to previous execution states.

7. CONCLUSION

We have presented a debugger for flow graph based parallel applications. By dynamically drawing the application flow graph as it unfolds, it enables the programmer to easily see the state of the execution as well as the status of every thread and operation. General features like operation and message breakpoints, operation inspection using a sequential debugger and message queue inspection provide the developer with much insight. The message tracing functionality allows focusing on messages and computations on a given branch of the flow graph while ignoring computations occurring in the rest of the application.

The ability to influence the application through the reordering or modification of messages provides the developer with full control over the execution of the application. This control can be used to execute cases that occur only rarely in practice and compare execution outcomes, for example for testing the presence of message races within the parallel application.

The Dynamic Parallel Schedules framework and its debugger are freely available at *http://dps.epfl.ch*.

8. ACKNOWLEDGEMENTS

The authors would like to thank the reviewers for their valuable comments.

9. REFERENCES

[1] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, M. Schulz, Stack Trace Analysis for Large Scale Debugging, *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, p.64, Long Beach, CA, March 2007

[2] S. M. Balle, B. R. Brett, C.-P. Chen, D. LaFrance-Linden, Extending a traditional debugger to debug massively parallel applications, *Journal of Parallel and Distributed Computing*, vol. 64, pp. 617-628, 2004

[3] J.-D. Choi, S. L. Min, Race Frontier: reproducing data races in parallel-program debugging, *Proceedings of the 3rd ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP' 91)*, pp. 145-154, 1991

[4] J. Cownie, W. Gropp, A standard interface for debugger access to message queue information in MPI, *PVM/MPI*, pp. 51-58, 1999

[5] Data Display Debugger, http://www.gnu.org/software/ddd

[6] Etnus, LLC. TotalView, http://www.etnus.com/TotalView

[7] S. Gerlach, R. D. Hersch, DPS - Dynamic Parallel Schedules, *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03), Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, pp. 15-24, Nice, France, April 2003, see also *http://dps.epfl.ch*

[8] S. Gerlach, R.D. Hersch, Fault-tolerant Parallel Applications with Dynamic Parallel Schedules, *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS'05)*, *Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*, p. 278b, 2005

[9] R. Hood, The p2d2 Project: Building a Portable Distributed Debugger, *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, pp. 127-136, Philadelphia, PA, 1996

[10] C.-E. Hong, B.-S. Lee, G.-W. On, D.-H. Chi, Replay for debugging MPI parallel programs, *Proceedings of the MPI Developer's Conference*, pp. 156-160, July 1996

[11] R. Jyothi, O. S. Lawlor, L. V. Kalé, Debugging Support for Charm++, *Proceedings of the 18th International Parallel and Distributed Symposium (IPDPS'04), Parallel and Distributed Systems: Testing and Debugging Workshop (PADTAD)*, p. 294, 2004

[12] R. Kilgore, C. Chase. Re-execution of distributed programs to detect bugs hidden by racing messages. *Proceedings of the 30th Hawaii International Conference on System Sciences (HICCS)*, vol. 1, p. 423, 1997

[13] E. Kraemer, J. T. Stasko, The Visualization of Parallel Systems: An Overview, *Journal of Parallel And Distributed Computing*, vol. 18, pp. 105-117, 1993

[14] D. Kranzlmüller, Scalable Parallel Program Debugging with Process Isolation and Grouping, *Proceedings of the 16th International Parallel and Distributed Symposium (IPDPS'02)*, pp. 109-115, April 2002

[15] T. J. LeBlanc, J. M. Mellor-Crumey, Debugging parallel programs with instant replay, *IEEE Transactions on Computers*, C36 (4), pp. 471-481, April 1987.

[16] T. J. LeBlanc, J. M. Mellor-Crumey, R. J. Fowler, Analyzing Parallel Programs Execution Using Multiple Views, *Journal of Parallel and Distributed Computing*, vol. 9 (2) , pp. 203-217, 1990

[17] S. S. Lumetta, D. E. Culler, The Mantis Parallel Debugger, *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, pp. 118-126, Philadelphia, PA, 1996

[18] Parallel Programming Laboratory, University of Illinois, Urbana-Champaign, *The Charm++ Programming Language, Version 6.0*, Jan 2004

[19] J. Postel, *Transmission Control Protocol*, RFC 793, Sept. 1981

[20] B. Schaeli, S. Gerlach, R.D. Hersch, Decomposing Partial Order Execution Graph to Improve Message Race Detection, *Proceedings of the, 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, *Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-TOPMoDRS)*, p. 187, Long Beach, CA, March 2007

[21] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, *MPI: The Complete Reference (Vol. 1)*, 2nd edition, MIT Press, 1998.

[22] N. Thoai, D. Kranzlmüller, J. Volkert, Shortcut Replay, A Replay Technique for Debugging Long-Running Parallel Programs, *Proceedings of the 7th Asian Computing Science Conference on Advances in Computing Science*, Lecture Notes In Computer Science, vol. 2550, pp. 34-46, 2002

[23] Q. Zheng, G. Cheng, L. Huang, Optimal record and replay for debugging of nondeterministic MPI/PVM programs, *Proceedings of the 4th International Conference on High Performance Computing in the Asia-Pacific Region*, vol. 1, pp. 473-475, May 2000