# Dynamic Testing of Flow Graph Based Parallel Applications

Basile Schaeli
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015, Lausanne, Switzerland
basile.schaeli@epfl.ch

Roger D. Hersch
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015, Lausanne, Switzerland
rd.hersch@epfl.ch

## ABSTRACT
In message-passing parallel applications, messages are not delivered in a strict order. The number of messages, their content and their destination may depend on the ordering of their delivery. Nevertheless, for most applications, the computation results should be the same for all possible orderings. Finding an ordering that produces a different outcome or that prevents the execution from terminating reveals a message race or a deadlock. Starting from the initial application state, we dynamically build an acyclic message-passing state graph such that each path within the graph represents one possible message ordering. All paths lead to the same final state if no deadlock or message race exists. If multiple final states are reached, we reveal message orderings that produce the different outcomes. The corresponding executions may then be replayed for debugging purposes. We reduce the number of states to be explored by using previously acquired knowledge about communication patterns and about how operations read and modify local process variables. We also describe a heuristic that tests a subset of orderings that are likely to reveal existing message races or deadlocks. We applied our approach on several applications developed using the Dynamic Parallel Schedules (DPS) parallelization framework. Compared to the naive execution of all message orderings, the use of a message-passing state graph reduces the cost of testing all orderings by several orders of magnitude. The use of prior information further reduces the number of visited states by a factor of up to fifty in our tests. The heuristic relying on a subset of orderings was able to reveal race conditions in all tested cases. We finally present a first step in generalizing the approach to MPI applications.

## Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming – *distributed programming*. D.2.5 [**Software Engineering**]: Testing and Debugging – *debugging aids, testing tools.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*PADTAD'08*, July 20–21, 2008, Seattle, Washington, USA.
Copyright 2008 ACM 978-1-60558-052-4/08/07…$5.00.

## General Terms
Reliability, Experimentation, Verification.

## 1. INTRODUCTION
One of the major difficulties when developing a parallel program is to simultaneously ensure that an application has good performance and that different executions with the same input always produce the same result. Achieving good performance generally requires removing synchronizations within the parallel program, with the risk that the correctness of the computation is no longer guaranteed. Unfortunately, the exponential number of possible message orderings makes it impossible to execute them all and to compare the final computation result after each run.

We describe a dynamic *message-passing state graph* construction and exploration technique that greatly reduces the cost of testing possible orderings. We identify application states common to multiple orderings dynamically by comparing checkpoints taken after the delivery of every message. This ensures that each state appears only once in the state graph, such that sequences of computations common to multiple orderings are executed only once. We then use information about communication patterns and read-write accesses to local process variables to reduce the number of explored states.

This approach greatly reduces the replay time at the expense of the memory or disk space needed to store intermediate application checkpoints. In order to handle cases where the space and time requirements are too large, we also describe an algorithm that tests a subset of orderings that has a high probability of revealing commonly found message races.

We implemented the proposed techniques within the Dynamic Parallel Schedules (DPS) framework [5]. This framework facilitates the creation of parallel applications by providing high-level constructs as well as checkpointing and fault-tolerance capabilities [6]. It is sufficient to recompile a DPS application in order to activate the message race and deadlock detection mechanism. Any modification to the application code or input data can therefore be immediately tested. Detected erroneous executions can then be replayed [8] for debugging purposes.

Although we present the ideas and example applications in the context of DPS applications, message-passing state graphs can be adapted to other message-passing models. Section 7 sketches such a generalization for MPI applications using a subset of MPI calls.

## 2. THE PARALLEL SCHEDULES MODEL

We now briefly describe the Dynamic Parallel Schedules framework [5]. DPS expresses a distributed memory parallel computation as a *flow graph* composed of serial operations arranged to form an acyclic directed graph, whose edges are defined by the messages 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 message once all expected messages have been received. Split operations are typically used to subdivide a high-level task into several subtasks that can be performed in parallel. Computation results are then collected and aggregated by the matching merge operation (Figure 1a). 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.

The processes involved in the computation are grouped into process collections, enabling groups of processes that play distinct roles within the application to be indexed independently. Each operation of the flow graph is attached to a process collection. The destination process of every message, and consequently of the triggered operation is computed at runtime via a user-defined function. The resulting message-passing graph of the application is thus known only at runtime (Figure 1b).

Operations running in different processes may be running concurrently, but in a given process, only one operation runs at a time. In order to allow the execution of other operations, merge and stream operations are suspended while waiting for messages to arrive. A networking layer abstracts the underlying communications, which are performed by MPI or by TCP sockets. The execution is fully asynchronous, and received messages are queued until they are delivered to the consuming operation. Given the acyclic nature of the flow graph, an associated message-passing graph is deadlock-free, provided that no operation terminates without outputting a message. However, message races may occur if the execution ordering of two non-commutative operations is not constrained by the flow graph.

Figure 2 displays the message-passing graph of one iteration of an iterative neighborhood-dependent parallel computation. Processes *P*[0], *P*[1] and *P*[2] belong to the same collection and each one stores one third of the processed data domain. At each iteration, every process sends a request to its neighbors, which send back a copy of their subdomain border (*Send border* operation). The computation of the new state of the subdomain (*Update* operation) is performed once the requested borders have been received.

However, this message-passing graph enforces no synchronization on a given process between the “border exchange” and “state update” phases. Therefore, delaying some messages may have unexpected consequences. In the execution depicted in Figure 3, the borders sent in messages (1) and (2) have already been updated, causing incorrect values to be used to update the subdomain stored on *P*[1] and distorting the results of the computation. The existence of the race depends on the actual implementation of the operations: in the present case, it is nonexistent if the borders to be exchanged are stored in double buffers, allowing a copy of the old border to be kept when *P*[0] and *P*[2] perform the update. Sending the copy of the old subdomain borders in messages (1) and (2) then allows the correct computation to be performed on *P*[1]. Detecting the race therefore requires executing the actual application code in both orderings.

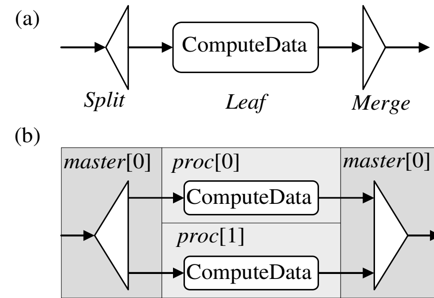


**Figure 1. (a) Flow graph describing a high level task divided into subtasks by a custom split operation and (b) a possible deployment onto three processes. The *master* process collection contains a single process and the *proc* process collection contains two processes.**

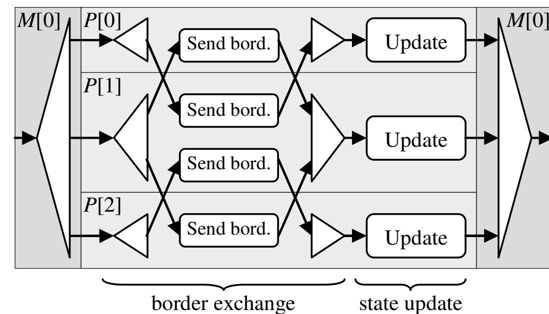


**Figure 2. The message-passing graph of one iteration of a neighborhood dependent parallel computation.**

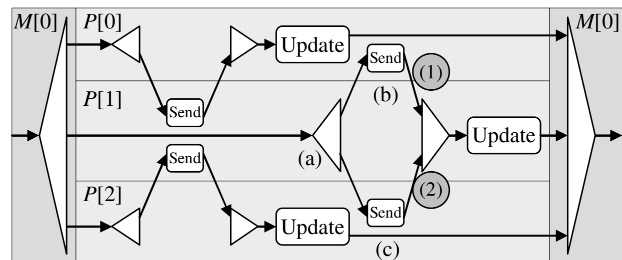


**Figure 3. If the split operation (a) on *P*[1] is delayed, the state of *P*[0] and *P*[2] is read by (b) and (c) after having been updated.**

## 3. BUILDING THE STATE GRAPH

We assume that computations are deterministic, and that processes exchange information only via messages. The delivery order of prior messages may influence the number, content and destination of subsequent messages. However, we assume that two executions with the same delivery order will produce the same messages. The only non-determinism lies thus in the ordering in which messages are delivered. Under these assumptions, each parallel execution of an application has at least one equivalent serial execution, defined by a specific ordering of message delivery. We therefore want to test that all message orderings and their associated serialized computations yield the same results.

We represent an execution as a sequence of states, where the transition from a state to the next is triggered by the delivery of a message. The transition ends upon completion of all the computations triggered by the delivered message. In our context, the state of the application is defined by the set of messages that have been sent but not yet received (i.e., the messages in transit), and by the value of the local variables of every process participating in the computation.

We may combine sequences corresponding to different orderings into a *message-passing state graph* by merging states common to different executions. Combining all possible sequences produces the *full* message-passing state graph of an application. Each path within the graph defines a different ordering of messages. A single state has multiple outgoing edges when more than one message is in transit, and has multiple incoming edges when it can be reached via several message orderings. Since in our execution model all computations are triggered by the delivery of a message, reaching a non-final state with no message in transit reveals a deadlock.

Figure 4 displays a simple example. Given the message-passing graph shown in (a), delivering the initial message 1 triggers the execution of the split operation, which sends messages 2 and 3 during its execution. These two messages are therefore in transit when the operation terminates and the next state is reached. We may then deliver either message 2 or message 3. If the two leaf operations triggered by messages 2 and 3 execute on different processes, they run within distinct memory spaces and cannot interfere with each other. Delivering message 2 before message 3 or message 3 before message 2 therefore leads to the same state with messages 4 and 5 in transit.

If we reach a single final state, we ensure that no message race or deadlock can occur for the given application input data. If a bug in the merge operation causes the content of the output message or the value of the local process variables to depend on the ordering of the delivery of messages 4 and 5, the final state will be different (Figure 5). When several final states are reached, we reveal the paths (i.e. the message orderings) leading to these states to enable their replay and study the erroneous execution. Although any set of paths will do, we choose the ones with the longest common prefix in order to help the developer focus on the ordering variation that caused the divergence in the executions, e.g., in Figure 5, paths 1-2-3-4-5 and 1-2-3-5-4.

Message-passing state graphs have the benefit of taking local and global synchronizations into account. Figure 6a displays the message-passing graph of a two-phase computation. For a single phase, the message-passing graph accepts 6 orderings of length 6, i.e. testing all orderings requires delivering 6·6 messages. For two phases, there are 36 orderings of length 11, which imply the delivery of 396 messages to execute all orderings. In contrast, the number of messages delivered while building the message-passing state graph is given by the number of edges in the graph and grows linearly (from 13 to 26) with the number of phases.

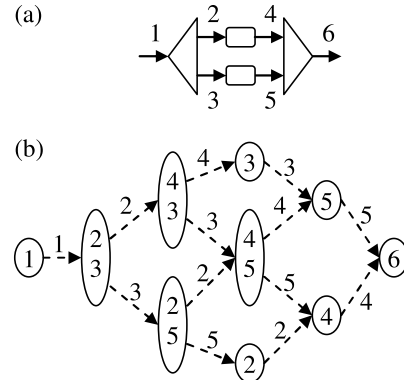


**Figure 4. (a) A message-passing graph and (b) its corresponding state graph. Edge labels identify the delivered message triggering the transition, and node labels indicate which messages are in transit.**

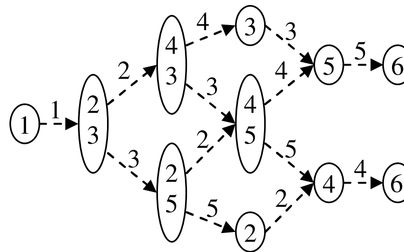

**Figure 5. Resulting state graph if the output of the merge operation is dependent on the ordering of its inputs (4 received before 5, vs. 5 received before 4).**

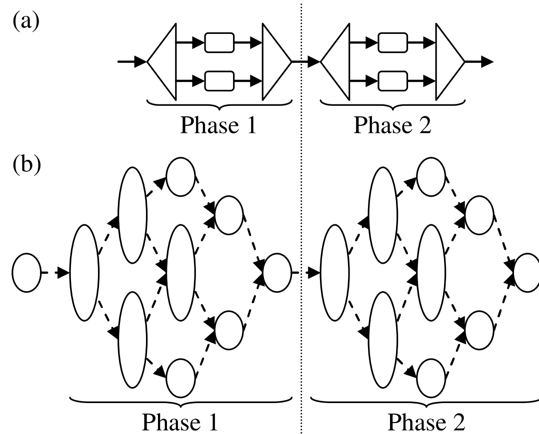


**Figure 6. The barrier synchronization caused by the merge-split sequence in the original flow graph (a) is reflected in the state graph (b).**

## 4. REDUCING THE NUMBER OF VISITED STATES

In the general case, many of the orderings contained in the full message-passing state graph are equivalent. Indeed, for a given ordering we may for instance exchange two consecutive messages that trigger operations running on different processes without modifying the computation results. If we can determine *a priori* that different subpaths in the message-passing state graph will produce identical results, we may cut redundant branches by not sending all the messages that are in transit at a given state. Looking back at Figure 4b for example, sending only message 2 after the delivery of message 1 avoids testing all orderings where message 3 is delivered before message 2, and removes two states from the graph.

Detecting equivalent orderings and determining which messages we may avoid delivering at every state therefore requires *a priori* knowledge about future computations. The computations triggered by two messages *a* and *b* delivered to distinct processes do not directly interfere, i.e., one computation cannot modify the process variables used by the other computation. However, future computations triggered by a successor of *b* may interfere with the computations triggered by *a*. If they do not, we may avoid delivering *b*; if they do, we have to deliver both *a* and *b*.

In our context, the DPS flow graph of the application provides this information: it specifies which operations may be triggered by a message and by its successors, as well as the process collection on which these operations execute. Figure 7 displays an example based on the application described in Section 2. Message 1 triggers operation C1, and one of its successors will eventually trigger an instance of operation E, which is a successor of C in the flow graph. However, since messages 1 and 2 are synchronized by operation D1, we do not need to consider E while determining the operations potentially interfering with messages 1 and 2. On the other hand, the first common successor of messages 1 and 3 is operation F1. Since the destination of each message is computed at runtime, the operation E1 triggered by a successor of message 1 may potentially be executed on the same process as operation C3. If this is the case, a race may appear if E1 modifies local process variables read by C3.

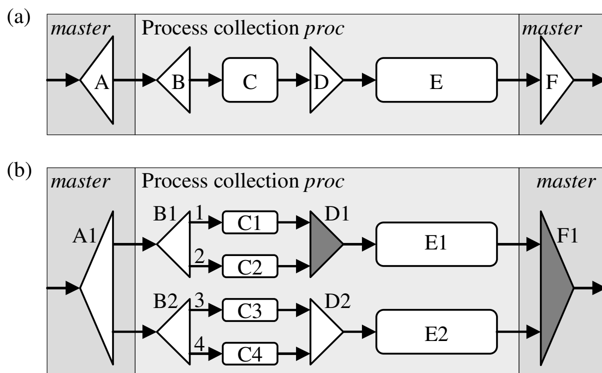


**Figure 7. (a) The flow graph of the application illustrated in Figure 2 and (b) its message-passing graph when deployed on two nodes. The first common successor of messages 1 and 2 is operation D in the original flow graph, while the first common successor of messages 1 and 3 is operation F.**

DPS messages carry a unique identifier [6]. Identifiers are built hierarchically by keeping the list of pending split operations that determines the flow graph branch to which the message belongs, e.g., in Figure 7, A1.B1 for message 1 and A1.B2 for message 3. The first common successor of two messages is therefore the merge operation that matches the split operation identified by the innermost split in the common prefix of their identifiers.

We may now establish rules for identifying sets of messages that may potentially interfere with each other. Let *S* be a state in the message-passing state graph with a set of messages in transit *M*. Let $C_m$ and $F_m$ be the set of *current*, respectively *future* interferers of a message $m \in M$.

1. $C_m$ contains *m* and all messages $m' \in M$ such that *m* and *m*' are delivered to the same process.
2. Given $m' \in M$, let $Succ_{m'}$ be the set of operations triggered by successors of *m*'. Then $F_m$ contains all messages $m' \neq m$ such that the first common successor of *m* and *m*', or at least one operation of $Succ_{m'} \setminus Succ_m$ runs on the same process collection as the operation triggered by *m*.

Both sets are computed dynamically for every message in transit of every state of the message-passing state graph. If the sets $F_m$ and $C_m$ associated to a message *m* are empty, the only message from *S* that we deliver is *m*. Otherwise, for each message $m \in M$ we augment its set $F_m$ by recursively computing the union of $F_m$ with the sets $C_n$ and $F_n$ for all $n \in F_m$. We then compare the sets $F_m$ associated to every message $m \in M$, and pick the set $F_{m^*}$ with the smallest cardinality. We then deliver all the messages contained in $F_{m^*}$. The rationale for selecting the smallest set is that delivering fewer messages per state creates fewer branches in the message-passing state graph, which tends to reduce the number of states to be explored.

Messages that trigger read-only operations can be reordered freely without any impact on the computation result. The number of interfering messages in the sets *C* and *F*, and thus the number of messages to be sent from each state can thereby be further reduced if we know whether an operation only reads or modifies the local variables of the underlying process. Given such information and the sets $C_m$ and $F_m$ as defined above, we may remove from $C_m$ and from $F_m$ every message *m*' such that the operation triggered by *m* does not write or read a variable modified by the operation triggered by *m*'.

## 5. IMPLEMENTATION AND RESULTS

We implemented the proposed mechanisms within the Dynamic Parallel Schedules (DPS) framework [5], and use its built-in checkpointing and restart capabilities [6] to store and recover intermediate application states.

The testing procedure starts right before delivering the input message of the flow graph. We build the initial application state by taking a checkpoint of each process and by making a copy of the input message. Unprocessed message-passing graph states are stored in a queue. For each unprocessed state *S* of the message-passing state graph, we determine the set of messages in transit that must be delivered. We then deliver one message from the set by pushing it into the incoming message queue of its destination

process, thereby triggering the associated operation. After the transition, we log the newly generated messages and checkpoint the process to which the message was delivered. Together with the checkpoints of the other processes, this forms the new state of the application and a successor of $S$ in the message-passing state graph. If the new state of the application has not been reached before, we add it to the queue of unprocessed states. We then roll back the application to its former state $S$ and deliver the next message. $S$ is removed from the queue when all required messages have been delivered.

## 5.1 Results

Let us present practical results for a few parallel applications. The metric used for all measurements is the total number of messages that must be delivered to test all considered orderings. In case of a naive test of all possible executions, we compute the number of messages that must be delivered by multiplying the number of permutations by the number of messages sent during one execution. When using a message-passing state graph, the number of delivered messages corresponds to the number of edges in the graph.

We first quantify the benefits of the state graph approach and of the proposed optimizations using the neighborhood-exchange (NE) application illustrated in Figure 2. Table 1 compares the number of messages delivered for exhaustively testing two iterations of the neighborhood-exchange computation when naively executing all orderings, when using the full message-passing state graph, and when applying the optimizations described in section 4. Table 1 shows that it is impossible to naively execute all orderings without building the state graph, even when the application runs on only two processes. For two processes the optimized message-passing state graph reduces the number of messages that must be delivered by a factor of $10^{13}$ compared to the naive execution of all possible orderings.

We carry out the same analysis for a parallel implementation of the Floyd-Steinberg halftoning algorithm (FS) which converts a grayscale image into a black and white image [10]. It determines for each grayscale pixel whether it should be black or white. The error, i.e., the difference between the desired grey value and the selected binary value, is then added according to an error-diffusion weight matrix to the grey value of the unprocessed neighboring pixels. Table 2 summarizes the results. For 4 processes, the optimized state graph reduces the number of messages that must be delivered by a factor of 50 compared to the full state graph.

Table 3 presents the results of our techniques running on a parallel block LU factorization application [7]. Our implementation requires at least three processes. Since the iterations of the computation are loosely synchronized in order to maximize the pipelining of the computation, messages have little dependencies between each other, causing the size of the message-passing state graph to explode, and almost cancelling the benefits of the optimizations described in section 4.

Finally, Table 4 presents results for a branch-and-bound solver for the Traveling Salesman Problem (no. of cities: 17). Messages distribute the value of the current best solution to processes in order to speed up the search, and a basic load-balancing scheme distributes computations more evenly among processes. Finding a good solution early will therefore impact the remaining computations. This dependence of the content and destination of messages on the ordering of prior computations increases the number of possible message-passing graph states. The running time therefore becomes prohibitive for testing the application exhaustively on more than two processes. All tests produced multiple final states, reflecting the existence of several solutions for our dataset: all final states showed the same minimum length for the total path, but with different orderings of cities.

**Table 1. Number of delivered messages (neighborhood-exchange application, two iterations).**

| | 2 proc. | 4 proc. | 6 proc. |
|---|---|---|---|
| All orderings | $5.6 \cdot 10^{16}$ | - | - |
| Full state graph | 1237 | $3.4 \cdot 10^{6}$ | - |
| Optimized state graph | 843 | $1.6 \cdot 10^{6}$ | $4.2 \cdot 10^{9}$ |

**Table 2. Number of delivered messages (parallel Floyd-Steinberg halftoning algorithm).**

| | 2 proc. | 4 proc. | 6 proc. | 8 proc. |
|---|---|---|---|---|
| All orderings | $6.8 \cdot 10^{8}$ | - | - | - |
| Full state graph | 338 | $3.9 \cdot 10^{4}$ | - | - |
| Optimized state graph | 47 | 765 | $2.7 \cdot 10^{4}$ | $1.0 \cdot 10^{6}$ |

**Table 3. Number of delivered messages (pipelined parallel LU factorization).**

| | 3 proc. | 4 proc. |
|---|---|---|
| All orderings | $>10^{17}$ | - |
| Full state graph | 4841 | $6.2 \cdot 10^{9}$ |
| Optimized state graph | 4780 | $6.2 \cdot 10^{9}$ |

**Table 4. Number of delivered messages (traveling salesman problem).**

| | 2 proc. |
|---|---|
| All orderings | $>10^{10}$ |
| Full state graph | $8.1 \cdot 10^{4}$ |
| Optimized state graph | $2.8 \cdot 10^{4}$ |

In order to test our message race detection software, we artificially introduced races by removing synchronizations or code that reorders messages within merge operations. We also discovered a few genuine potential message races in previous implementations of the LU factorization application.

## 5.2 Scalability issues

While building the state graph, one often encounters the same messages and process checkpoints many times. We therefore save memory by keeping a single physical copy of every element, and by discarding messages and checkpoints no longer needed. All elements are stored in hash tables to be quickly compared and retrieved.

**Table 5. Running time [s] and memory consumption [MB] for regular executions and for tests relying on the message-passing state graph.**

| | Regular execution on 4 proc. | | 4 proc. | | 6 proc. | |
|---|---|---|---|---|---|---|
| | [s] | [MB] | [s] | [MB] | [s] | [MB] |
| NE | 0.13 | 2 | 452 | 227 | 117617 | 5689 |
| FS | 0.014 | 0.066 | 0.716 | 1.9 | 48 | 19 |
| LU | 0.014 | 0.115 | 33804 | 302 | - | - |

Nevertheless, the number of delivered messages impacts both the running time and the amount of memory required to build the message-passing state graph. The first column of Table 5 provides the regular application running time with four processes running on a single dual-core computer, and the memory occupied by the input data (e.g. a matrix or an image). When building the message-passing state graph (columns two and three), we indicate the full running time of the testing procedure, and the amount of memory used by the testing process.

Scaling the tests from 2 to 4 and 6 processes leads to strongly increased execution times and memory consumption. These numbers need to be compared with the running time of executing all orderings: on only two processes, we would already need in the order of $10^{14}$ seconds for the neighborhood-exchange (NE) application, and $10^{5}$ seconds for the Floyd-Steinberg (FS) application.

For three (NE, FS, LU) of our four test applications and for many real-world codes, the number of messages sent and the communication patterns are independent of the size or content of the processed data. Testing small data sets is thus often sufficient for revealing message races and programming errors. Nevertheless, the numbers shown in Table 5 clearly point to the fact that only tiny portions of a parallel program can be tested exhaustively. One could for instance test orderings only between two barrier synchronizations.

# 6. TESTING A SUBSET OF ORDERINGS

Despite the orders of magnitude decrease in the number of explored states, using a message-passing state graph does not change the fundamental fact that the number of states and checkpoints grows exponentially with the number of processes. It is therefore crucial that we can generate a subset of orderings that reveals most if not all of the potential deadlocks and message races.

It is often the relative rather than the absolute ordering of two messages that causes a message race. For example, an error may occur for all orderings where message *b* is delivered before message *a*, regardless of the ordering of other messages. Many such errors occur when one message is unexpectedly delayed. We therefore suspect that most message races can be revealed by testing a small subset of carefully selected orderings.

We generate a first reference ordering by delivering messages in the order in which they are sent by the application. This corresponds to a breadth-first traversal of the message-passing graph of the application. We produce the ordering using a single FIFO queue: new messages are pushed at the back of the queue, and at each transition we deliver the message at the head of the queue (Figure 8b).

The second ordering is produced via a depth-first traversal of the application message-passing graph, where we send all messages

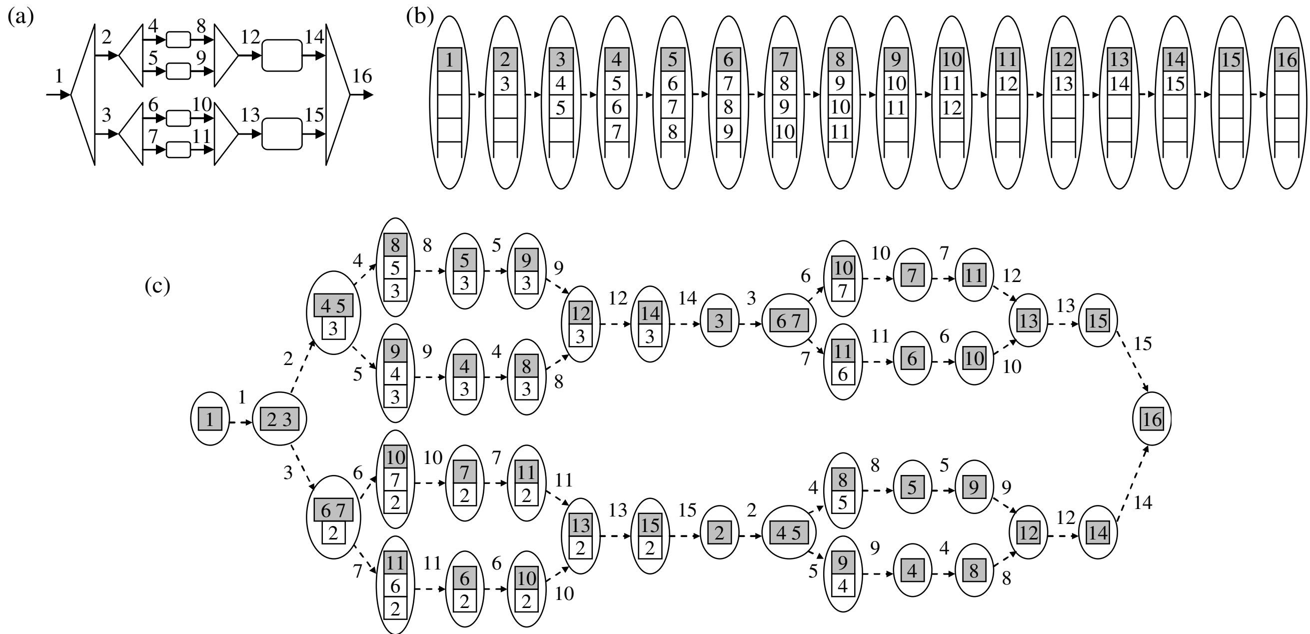


**Figure 8. (a) The message-passing graph of one iteration on two nodes of the neighborhood-dependent computation described in section 2; (b) message-passing state graph for the breadth-first traversal; (c) message-passing state graph of all the possible depth-first traversals (i.e. all branch orderings) of the message-passing graph. Highlighted numbers and edge labels represent the messages triggering the transition from a state to the next.**

from a single branch before sending a message from any other branch. This scheme simulates delays on specific branches, and is implemented as follows. Each state stores its messages in transit as a stack (i.e., a LIFO queue) of sets. A set contains all the messages generated by the delivery of a single message. A transition is triggered by delivering one message from the set at the head of the stack. Once a transition completes, we copy the stack into the new state, remove the message that caused the transition and, if new messages were generated, push them in a new set onto the stack. If no messages are produced, the head of the stack contains a set with unsent messages from a previous transition. When the head set contains more than one message, the choice of the message to deliver defines the order in which we execute the branches of the message-passing graph.

Multiple branch orderings may be tested by delivering more than one message from the head set. Testing all branch orderings for instance ensures that we test executions that maximize and minimize the delays that can be experienced by each branch. Figure 8c displays a message-passing state graph representing all the possible orderings of branches of the message-passing graph in Figure 8a, created by always sending all the messages in the head set. Each path in that state graph represents a single depth-first traversal of the message-passing graph.

Table 6 displays the number of explored states for testing all branch orderings of our applications when applying the heuristic on the full message-passing state graph. We can see that for the FS application, the test using the heuristic on the full state graph delivers more messages than using the optimized state graph (Table 1). For less constrained applications such as the pipelined LU and the travelling salesman, testing only permutations of branch orderings strongly reduces the number of delivered messages.

The optimization and the heuristic could obviously be combined. Moreover, in practice, existing symmetries in the computations performed by different processes imply that a single error is revealed by multiple orderings of branches. A few depth-first executions with distinct branch orderings were sufficient to reveal errors in all our applications.

Since one may conceive an application whose message races or deadlocks are not exposed by the orderings defined above, we may additionally execute randomly generated orderings in order to further reduce the probability that errors remain undetected. Such a technique was successfully used for detecting data races in multithreaded applications [16]. By increasing the number of randomly generated orderings, one can arbitrarily increase the confidence that no message race or deadlock exists.

**Table 6. Number of delivered messages [msgs], running time [s], and memory consumption [MB] for testing all message-passing graph branch permutations on 4 processes.**

| | [msgs] | [s] | [MB] |
|---|---|---|---|
| Neighborhood-exchange | 1737 | 8.32 | 10.0 |
| Floyd-Steinberg | 3513 | 1.92 | 4.6 |
| LU factorization | $4.7{\cdot}10^5$ | 92 | 31.0 |
| Travelling salesman | 481 | 240 | 4.4 |

# 7. APPLYING MESSAGE-PASSING STATE GRAPHS TO MPI APPLICATIONS

We now present a conceptual generalization of the message-passing state graph construction for MPI applications. We only consider applications that perform deterministic computations and use non-buffered blocking point-to-point and collective communications. Under these restrictions, non-determinism only stems from the use of wildcard receives that allow a single *recv* call to receive a message from multiple sources. We therefore want to test that the computation result remains the same no matter which message is actually received.

In DPS applications, new computations are triggered by the delivery of a message. In MPI applications that satisfy our restrictions, new computations are triggered when a set of calls from distinct processes are matched. For collective communications, processes block until all participating processes have joined the communication. For blocking point-to-point MPI communications, a *send* call at the source must match a *recv* call at the destination before any of the sending and receiving process may resume its computation.

We represent an execution using an event model inspired by [20], where every MPI call produces one event. We now construct a message-passing state graph as follows. Starting from the initial application state where all processes have called the MPI_Init function, a transition occurs by matching a *send* event with a *recv* event, or by matching a set of collective communication events, thereby allowing the suspended processes to resume their execution until the next MPI function call. In respect to the state graph construction, the state of a single process is defined by the value of its local variables and by the pending MPI call. The state of the whole application is defined as the set of states of the individual processes.

We illustrate the state graph construction of a simple parallel merge sort application. Each process initially stores a local array containing $n/p$ sorted elements, that must be merged into a single sorted array of size $n$. Figure 9 shows the MPI pseudocode of the application. The number of participating processes $p$ is a power of 2. The $p/2$ processes with the largest ranks send their array to one of the $p/2$ processes with the smallest ranks. The $p/2$ receiving processes merge together the received and local arrays. The $p/2$ processes with the largest ranks then exit the loop, and the $p/2$ processes with the smallest ranks enter the next iteration. At every iteration, half of the remaining processes exit the loop. When all the pieces have been merged, the last process exits the loop and the application exits.

Figure 10a displays the message-passing graph of the expected execution on four processes, when all processes move simultaneously from one iteration to the next. The *init* and *finalize* events from process $i$ are denoted $I_i$ and $F_i$, and are assumed to behave as barrier synchronizations. A send from process $i$ to process $j$ at iteration $k$ is denoted as $s^k_{i,j}$, while $r^k_{*,j}$ denotes the matching wildcard receive event.

If the send $s^0_{2,0}$ of process 2 is delayed however, the send event $s^1_{1,0}$ may instead match the $r^0_{*,0}$ receive (Figure 10b). As $r^0_{*,0}$ expects an array containing $n/4$ elements while process 1 sends

```
/* myRank: process rank                          */
/* n: no of elements to be sorted                */
/* p: no of processes (power of 2)               */
/* procsInLoop: no of processes in the loop */
/* Every process stores a sorted subarray
   containing n/p elements                       */

MPI_Init()
procsInLoop = p
do {
  if (myRank ≥ procsInLoop/2)
    MPI_Ssend(array of size (n/procsInLoop)
              to myRank%(procsInLoop/2)

  else {
    MPI_Recv(array of size (n/procsInLoop)
             from any source)
    /* Merge the received and local arrays */
  }
  procsInLoop = procsInLoop/2
}
while(myRank<procsInLoop && procsInLoop>1)

MPI_Finalize()
```

**Figure 9. MPI pseudocode for a parallel merge sort application.**

$n/2$ elements, the MPI implementation may either deliver a truncated message or raise a fatal error.

The corresponding message-passing state graph of the application is displayed in Figure 11a. The match of the *send* event $s^k_{i,j}$ with the *recv* event $r^{k'}_{*,j}$ is indicated as $\{s^k_{i,j}, r^{k'}_{*,j}\}$. The incorrect match $\{s^1_{1,0}, r^0_{*,0}\}$ is easily detected as message sizes are different. The programmer can solve the problem by explicitly specifying the source of the expected message in the receive call. Then $s^1_{1,0}$ can no longer match $r^0_{2,0}$ (Figure 11b).

We can adapt the breadth-first and depth-first executions described in section 6 as follows. We maintain for each process a counter that stores the number of events that have been matched. We then produce a depth-first (resp. breadth-first) execution by always matching events for the process with the largest (resp. smallest) counter value. In our example, the error is revealed by a depth-first execution of the application. We initialize every match counter to 0, and after matching the init events we match the events of processes 3 and 1 $\{s^0_{3,1}, r^0_{*,1}\}$. Their match counter is incremented to 2 and become the ones with the largest value. In order to produce a depth-first execution, we therefore attempt to match events from these processes in priority. The *finalize* event of process 3 ($F_3$) cannot be matched yet, but we may match the send event $s^1_{1,0}$ of process 1 with the receive event $r^0_{*,0}$ of process 0, thereby producing the incorrect execution.

Although one could easily run a single breadth-first or depth-first execution, building the state graph and avoiding restarting the program for each sequence of event matches requires additional checkpointing support. The Berkeley Lab Checkpoint/Restart library has been successfully used with MPI implementations such as LAM and OpenMPI [1, 18]. DéjàVu [17] specifically targets MPI and distributed programs. Integrating one of these libraries would also require means for performing fast process data checkpoint comparisons.

DPS has the advantage of providing a high-level description of dependencies between computations. In MPI, the lack of a flow

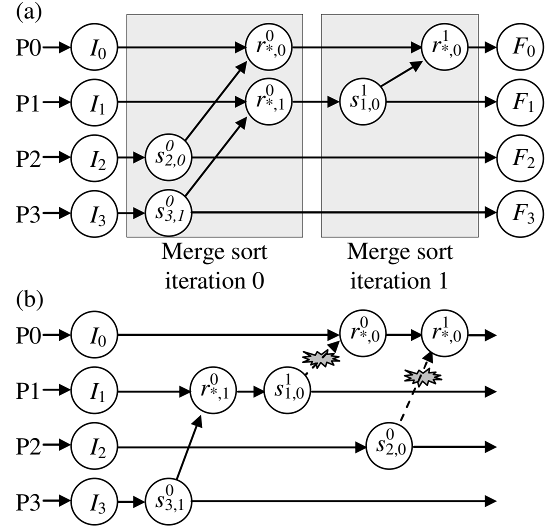


**Figure 10. (a) Message-passing graph of a correct execution on four processes and (b) of an incorrect execution, where send from rank 2 is delayed.**

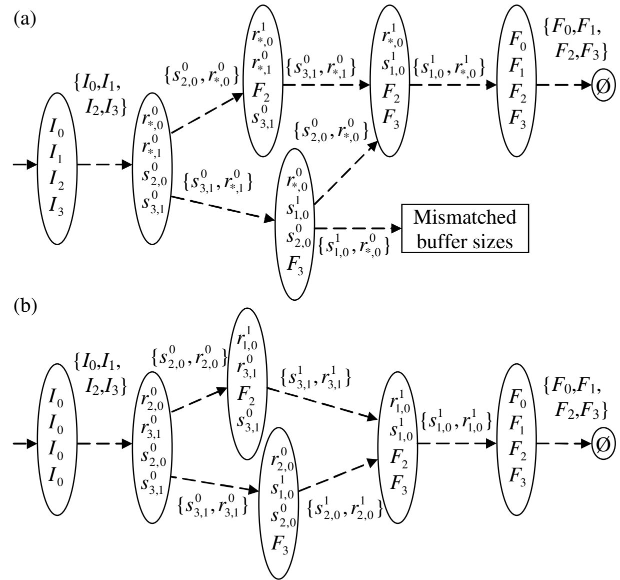


**Figure 11. (a) State graph of the application with wildcard receives and (b) state graph without wildcard receives. Nodes indicate the pending event of each process, while edge labels indicate which event match produces the transition.**

graph or equivalent information about future communication patterns implies that we cannot readily apply the optimizations described in section 4. However, the potential for optimizations does exist. In Figure 11b for instance, all matches are deterministic and both paths in the message-passing state graph are guaranteed to produce the same result. One of the branches could therefore be pruned. Such insight could potentially be provided by a model of the tested application [20, 21]. Another interesting approach consists in dynamically identifying interesting backtracking points by collecting information during the application execution [4].

## 8. RELATED WORK

Mosbah and Ossamy [12] and Otta and Racek [14] detect message races by evaluating predefined predicates that consider both the local and the global application state at various points of the execution. Since no control is applied on the program execution, the detection only works for executions where message races actually occur.

Several variants of controlled re-execution of message-passing applications have been described in the literature. Mittal and Garg [11] determine where to add synchronizations in order to maintain a global predicate, thereby pointing to the location of synchronization bugs. However, they do not allow events to be reordered on a given process. Duesterwald et al. [1] describe a slicing method to isolate only problematic statements when an erroneous result is observed. The slice may then be re-executed for identifying the source of error. Kilgore and Chase [9] identify sets of messages that can be received in any order on a given process, and propose an algorithm that generates a single ordering that maximizes the number of reversed message pairs compared to the original execution. In our previous work [19], we described a static message-passing graph decomposition technique and a partial-order reduction method to reduce the number of tested orderings. In both approaches the destination, content and number of messages sent by the application are assumed to be independent of the ordering of their delivery.

Several authors argue that detecting the first message race is beneficial [13, 15]. Correcting early races not only removes subsequent instances of the same race, but also prevents potential spurious races from being enabled. By identifying the longest common prefix between message orderings yielding different results, we can determine at which point the executions start to diverge, allowing the developer to correct early races first.

Considering work that specifically targets MPI applications, Siegel and Avrunin have been working on the development of formal models of MPI applications [20, 21]. Their modelization inspired the generalization described in the previous section. Very recently, Vakkalanka et al. presented ISP [22], a tool that automatically executes all relevant event orderings within MPI applications. It implements no checkpointing however, and therefore requires reexecuting the whole application for every ordering.

A large body of work exists that focuses on the analysis and verification of multithreaded applications. Since the number of possible thread interleavings is even more intractable than the number of message permutations in distributed memory message-passing programs, a central goal is the detection of equivalent interleavings. In particular, several proposals (e.g. [2, 3, 23]) use information about read/write access to shared variables to determine the commutativity of operations and reduce multiple equivalent executions to the same serialized execution.

## 9. CONCLUSION

We present a method to dynamically test multiple message orderings in a message-passing parallel application. We represent the multiple orderings using a message-passing state graph built at runtime, which ensures that parts common to multiple orderings are executed only once. We then use information about future computations and about how these computations read or write local process variables to identify equivalent orderings within the message-passing state graph. We also show how to generate a subset of orderings that are still able to reveal most errors.

We integrated the dynamic message-passing state graph construction and analysis method within the Dynamic Parallel Schedules parallelization framework, which provides support for checkpointing and restarting individual processes during a computation. For four different parallel applications, we evaluated the influence of the proposed techniques on the total number of messages that must be delivered in order to test all orderings. The message-passing state graph and the described optimizations enable reducing the number of delivered messages by many orders of magnitude compared to the naive approach. The proposed partial tests revealed all errors present in our experiments.

We finally presented a generalization of the approach to MPI applications using a subset of MPI calls.

The DPS software is available on the Web under the GPL license at http://dps.epfl.ch.

## 10. REFERENCES


[1] A. Bouteiller, G. Bosilca, J. Dongarra, Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science (LNCS), Vol. 4757, pp. 297-306, Springer Verlag, 2007, doi:10.1007/978-3-540-75416-9_41

[2] F. Chen, G. Roşu, Parametric and Sliced Causality, *Proceedings of the 19th Conference on Computer Aided Verification (CAV'07)*, Lecture Notes In Computer Science (LNCS), Vol. 4590, Springer, pp 240-253, 2007

[3] C. Flanagan, S.N. Freund, S. Qadeer, Exploiting purity for atomicity, *Software Engineering, IEEE Transactions on*, vol. 31, no. 4, pp. 275-291, April 2005

[4] C. Flanagan, P. Godefroid, Dynamic Partial-Order Reduction for Model Checking Software, *Proceedings of the 32nd ACM symposium on Principles of programming languages (POPL'05)*, pp. 110-121, Long Beach, California, USA, 2005

[5] S. Gerlach, R. D. Hersch, DPS - Dynamic Parallel Schedules, *Proc. 17th Int'l Parallel and Distributed Processing Symposium (IPDPS'03)*, pp. 15-24, Nice, France, April 2003, see also *http://dps.epfl.ch*

[6] S. Gerlach, R.D. Hersch, Fault-tolerant Parallel Applications with Dynamic Parallel Schedules, *Proc. 19th Int'l Parallel and Distributed Processing Symposium (IPDPS'05)*, p. 278b, 2005

[7] G. H. Golub, C. F. van Loan, *Matrix Computations*, The Johns Hopkins University Press, pp. 94-116, 1996

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

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

[10] P. T. Metaxas, Parallel digital halftoning by error-diffusion, *Proc. Paris C. Kanellakis memorial workshop on Principles of computing & knowledge: Paris C. Kanellakis memorial*

*workshop on the occasion of his 50th birthday*, pp. 35-41, 2003.

[11] N. Mittal, V. K. Garg, Debugging distributed programs using controlled re-execution, *Proc. 19th ACM Symposium on Principles of Distributed Computing*, pp. 239-248, 2000

[12] M. Mosbah, R. Ossamy, Checking global properties for local computations in graphs with applications to invariant testing, *Proc. 5th Mexican Conference in Computer Science*, pp. 35-42, 2004

[13] R. H. B. Netzer, T. W. Brennan, S. K. Damodaran-Kamal, Debugging race conditions in message-passing programs, *Proc. SIGMETRICS Symposium on Parallel and Distributed Tools*, pp. 31-40, 1996

[14] M. Otta, S. Racek, A method for testing and debugging distributed applications, *Int'l Conference on Trends in Communications (EUROCON'2001)*, vol. 2, pp. 548-551, July 2001

[15] H.-D. Park, Y.-K. Jun, Detecting the first races in parallel programs with ordered synchronization, *Proc. 1998 Int'l Conference on Parallel and Distributed Systems*, pp.201-208, 1998

[16] S. Qadeer, D. Wu, KISS: keep it simple and sequential, *Proc. ACM SIGPLAN 2004 Conference on Programming language design and implementation*, pp. 14-24, 2004

[17] J.F. Ruscio, M.A. Heffner, S. Varadarajan, DejaVu: Transparent User-Level Checkpointing, Migration, and Recovery for Distributed Systems, *Proc. 21st Int'l Parallel and Distributed Processing Symposium (IPDPS'07)*, pp. 1-8, Long Beach, USA, March 2007.

[18] S. Sankaran, J.M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, E. Roman}, The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing, *Int'l Journal of High Performance Computing Applications*, Vol. 19, No. 4, pp. 479-493, 2005

[19] B. Schaeli, S. Gerlach, R.D. Hersch, Decomposing Partial Order Execution Graphs to Improve Message Race Detection, *Proc. 21st Int'l Parallel and Distributed Processing Symposium (IPDPS'07)*, pp. 1-8, Long Beach, USA, March 2007.

[20] S.F. Siegel, G.S. Avrunin, Modeling wildcard-free MPI programs for verification, *Proc. ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP'05)*, pp. 95-106, 2005.

[21] S.F. Siegel, G.S. Avrunin, Verification of halting properties for MPI programs using nonblocking operations, *Proc. 14th European PVM/MPI Users' Group Conference*, 2007.

[22] S.S. Vakkalanka, S. Sharma, G. Gopalakrishnan, R.M. Kirby, ISP: a tool for model checking MPI programs. *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 258-256, 2008.

[23] L. Wang, S.D. Stoller, Runtime analysis of atomicity for multithreaded programs, *Software Engineering, IEEE Transactions on* , vol. 32, no. 2, pp. 93-110, Feb. 2006