Decomposing Partial Order Execution Graphs to Improve Message Race Detection

Basile Schaeli Sebastian Gerlach Roger D. Hersch
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
School of Computer and Communication Sciences
{basile.schaeli, sebastian.gerlach, rd.hersch}@epfl.ch

Abstract

In message-passing parallel applications, messages are not delivered in a strict order. In most applications, the computation results and the set of messages produced during the execution should be the same for all distinct orderings of messages delivery. Finding an ordering that produces a different outcome then reveals a message race. Assuming that the Partial Order Execution Graph (POEG) capturing the causality between events is known for a reference execution, the present paper describes techniques for identifying independent sets of messages and within each set equivalent message orderings. Orderings of messages belonging to different sets may then be re-executed independently from each other, thereby reducing the number of orderings that must be tested to detect message races. We integrated the presented techniques into the Dynamic Parallel Schedules parallelization framework, and applied our approach on an image processing, a linear algebra, and a neighborhood-dependent parallel computation. In all cases, the number of possible orderings is reduced by several orders of magnitudes. In order to further reduce this number, we describe an algorithm that generates a subset of orderings that are likely to reveal existing message races.

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 runs with the same input always produce the same result. Obtaining 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 compare the final computation result after each run.

For most scientific applications however, both the number of messages produced during the program execution as well as their content are independent of the order in which messages are delivered. We say that such applications generate a *fixed message set*. For instance, most linear algebra computations and finite elements methods have that property. On the contrary, applications such as divide-and-conquer optimization problems where the current best solution influences the remaining of the computation do not belong to that category, and are therefore not considered in the current study.

We present a method to reduce the number of possible orderings that must be tested to detect message races within message-passing parallel applications producing a fixed set of messages. Given the Partial Order Execution Graph (POEG) [1] of an execution of a parallel application, a static analysis of the graph enables partitioning it into smaller parts. The remaining number of possible orderings is further reduced by using a partial-order reduction technique that leverages knowledge about whether computation steps read or modify the local memory.

Reducing the number of equivalent orderings ensures that only relevant cases are tested, therefore increasing the likelihood that existing message races are revealed. Moreover, the decomposition of the application execution allows developers to focus their effort on specific parts of the application that may be difficult to debug. Once a race is detected, the decomposition isolates its potential sources within the part being tested.

We implemented the proposed techniques within the Dynamic Parallel Schedules (DPS) framework [3]. This framework facilitates the creation of parallel applications by providing high-level constructs. Parallel applications are described as data-driven acyclic graphs of serial operations, allowing POEGs to be easily derived from execution traces. It is sufficient to recompile a parallel application in order to enable the message race detection mechanism. Any modification to the application code or input data is therefore immediately taken into account. Detected erroneous executions can then be replayed [1, 6] for debugging purposes.

21st International Parallel and Distributed Processing Symposium (IPDPS'07)
12th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-ToPMoDRS), March 26-30, Long Beach, USA, 2007

Although the ideas and example applications are presented in the context of DPS applications, they can be generalized to any parallel program that can be modeled using a POEG. This is illustrated in section 7, where we apply the POEG decomposition to a simple MPI application.

2. The Parallel Schedules Model

DPS describes 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.

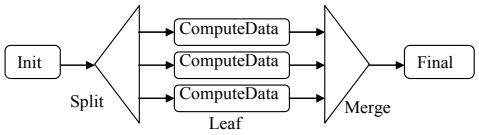

Figure 1. Flow graph describing a high level task divided into subtasks by a custom split operation. Leaf operations perform their tasks in parallel.

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 1). 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.

Split and leaf operations are executed atomically. On the other hand, in order to allow the execution of other operations, merge and stream operations are suspended while waiting for messages to arrive. Given the acyclic nature of the flow graph, a parallel schedule is deadlock-free, provided that no operation terminates without outputting a message.

Operations running in different processes may be running concurrently, but in a given process, only one operation runs at a time. Pending messages are queued until they are delivered to the consuming operation. A message race may therefore occur if the execution ordering of two non-commutative operations is not constrained by the flow graph.

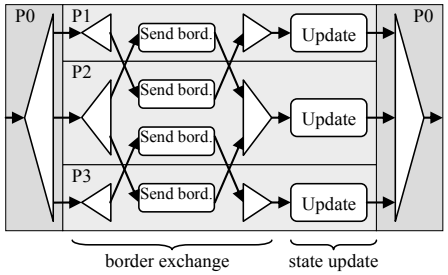

Figure 2. The flow graph of one iteration of a neighborhood dependent parallel computation.

Figure 2 displays the flow graph of one iteration of an iterative neighborhood-dependent parallel computation. Processes P1, P2 and P3 each store 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*) is performed once the requested borders have been received. However, this flow graph enforces no synchronization between the border exchange and state update phases. Therefore, delaying some messages may have unexpected consequences.

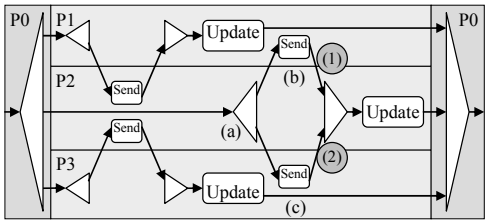

Figure 3. If the processing of the split operation (a) on P2 is delayed, the state of P1 and P3 is read by (b) and (c) after having been updated.

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 P2 and distorting computation results. However, the existence of the race depends on the actual implementation of the operations: here it is inexistent if the borders to be exchanged are stored in double buffers, allowing a copy of the old border to be kept when P1 and P3 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 P2. Detecting the race therefore requires executing the actual application code for different orderings of message delivery. We develop this example throughout the paper to illustrate the concepts of the partial order graph decomposition.

3. Partitioning the POEG

Given a trace of an execution of a DPS application, we can easily determine the causal dependencies between the different messages sent during the application execution (Figure 4a). We then decompose all operations into *atomic steps*, which represent parts of operations that are executed atomically (Figure 4b). An atomic step is triggered by an incoming message. Leaf and split operations consist of a single atomic step, while merge and stream operations are decomposed into one atomic step per input message. Therefore, admissible message orderings are equivalent to admissible orderings of atomic step executions (Figure 4c). The processed message-passing graph is called the Partial Order Execution Graph (POEG) of the application execution [1], where edges represent Lamport's *happened-before* relationship [8]. In our context, a message *a* is delivered before *b*, or, equivalently, the atomic step triggered by *a* is executed before the one triggered by *b*, if *a* is a predecessor of *b* in the POEG.

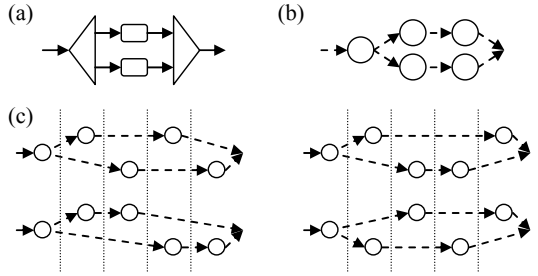

Figure 4. (a) A message-passing graph of a simple application, (b) its corresponding Partial Order Execution Graph, and (c) four orderings allowed by the POEG.

In the general case, the number of admissible orderings grows exponentially with the number of messages sent, making it impossible to test them all. Many orderings can however be prevented by partitioning the POEG into subgraphs representing parts of the application execution, such that each part can be processed independently from the others.

We first separate the contributions of atomic steps running on different processes. Indeed, swapping the delivery order of two messages delivered to different processes will not change the computation outcome. However, for the contributions of each process to be tested independently, we must assume that the set of messages delivered to each process is fixed. Since the input messages of a process are the output messages of other processes, the set of messages produced by each process must be fixed. Finding an ordering of the inputs causing a process to produce a different set of output messages then reveals a message race. If the different outcome is actually intentional, the application being tested does not produce a fixed message set, and the contribution of the different processes may not be tested independently from each other. Such cases are not considered in this paper.

Given the POEG of the whole application, we obtain the POEG of each process by removing all the messages delivered to other processes, while maintaining the causality between messages delivered to the process under consideration. The POEG of each process therefore defines the admissible orderings of all computations performed in the same memory space. Figure 5 illustrates this principle by isolating the contribution of process P2 from Figure 2. When focusing on a single process, it becomes clear that the execution order of the *send border* and *update* operations (S and U in Figure 5) is not constrained, and that a message race might occur.

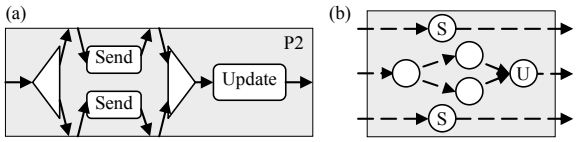

Figure 5. (a) Operations running on process P2 of Figure 2, and (b) the POEG of P2, after decomposition into atomic steps and removal of messages delivered to other processes.

Within a process, the causality between messages can prevent distinct subgroups of messages from being interleaved. Figure 6 illustrates this principle on two iterations of our neighborhood-dependent computation. The synchronization enforced by the *merge-split* construct found between the two iterations is represented within the POEG by the fully interconnected dependences between the atomic steps preceding the *merge* and the ones following the *split* operation (Figure 6b).

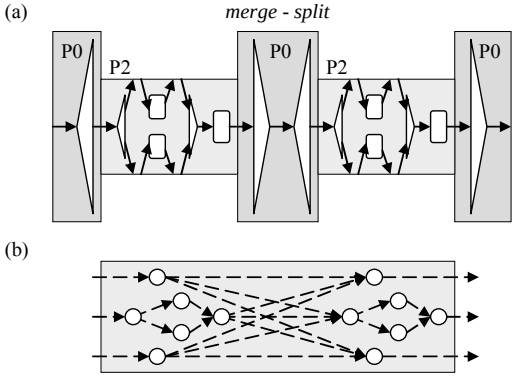

Figure 6. (a) Two iterations of the neighborhood-dependent computation illustrated in Figure 2 (P1, P3 not shown), and (b) equivalent POEG of P2.

In order to identify causally dependent subgroups in the POEG of each process, we first introduce an auxiliary atomic step between every pair of sets of fully interconnected atomic steps (dark grey node in Figure 7a). We then add a source and sink node to the graph, and run a

unit flow through it. The output flow of each atomic step is split equally between each successor, and the contributions of multiple input flows to a single atomic step are added. When the sum of the input flows into an atomic step is one, the number of messages delivered before and respectively after its execution is constant. We are therefore allowed to split the messages sent before and after such an atomic step into two consecutive subgroups, each one with its own POEG.

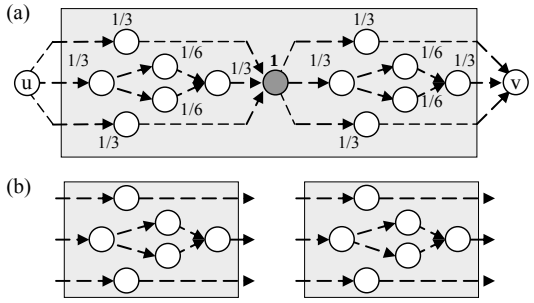

Figure 7. (a) Introduction of an auxiliary atomic step (dark grey), and of source and sink nodes *u* and *v*, and (b) partition of the POEG into subgroups.

4. Partial-Order Reductions

Within a partial order execution graph, orderings can be prevented by inserting additional edges, which force the relative delivery order of specific messages, or equivalently, the relative execution order of the atomic steps they trigger.

Some of the orderings that should be prevented are the ones allowed by the POEG that cannot occur in practice. For instance, message-passing libraries may guarantee a FIFO delivery of messages. Applications that rely on that assumption may send subsequent pieces of data without explicit synchronization between successive messages. In order to account for that assumption within the POEG, we may, when a single delivered message causes multiple messages to be sent, add ordering constraints between the messages that are destined to the same destination process.

Within subgroups, distinct message orderings may also be equivalent. Let us assume that we know for each operation (and by extension for each atomic step and the message that triggers it) if the state of the process is only read or if it is modified. If two atomic steps only read the unmodified state, the order of their execution has no impact on the final process state. If two messages trigger such atomic steps, and if the POEG defines no causality between them, we say that the messages are *exchangeable*. However, we cannot constrain their delivery order without taking their successors and predecessors into account. Indeed, if we constrain a to be delivered before b, we transitively constrain all predecessors of a (noted $Pred_a$) to be delivered before all successors of b (noted $Succ_b$). An edge $a \rightarrow b$ can therefore be added to the POEG only if every message in $\{a\} \cup Pred_a \backslash Pred_b$ is exchangeable with all messages in $\{b\} \cup Succ_b \backslash Succ_a$ ("\" denotes the set difference operation). The partial-order reduction [5] of a POEG is therefore performed by adding all the edges that satisfy this condition.

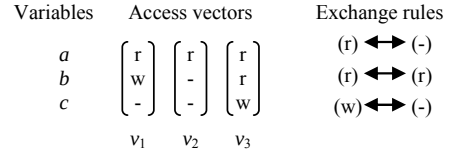

Figure 8. Three possible access vectors for a process with three member variables a, b and c. Member variables can be read (r), modified (w), or ignored (-). According to the exchange rules on the right, the message associated to v_2 can be exchanged with those associated to v_1 and v_3.

Computations that modify different variables in the process state may also be commutative. Detailed memory access information can be represented using *access vectors*, which specify for each message whether each variable of the process state is read, written or ignored by the triggered atomic step. The partial-order reduction can also be applied, using a generalized definition for the exchange of messages: two messages can be exchanged if all the members of their access vector can be exchanged (Figure 8).

An application of the algorithm is illustrated in Figure 9a, where edges are added between read-only atomic steps (in white). The orientation of the new edges is chosen so as not to prevent orderings where the process state is read after the update. This enables us to check that such orderings do not change the computation outcome.

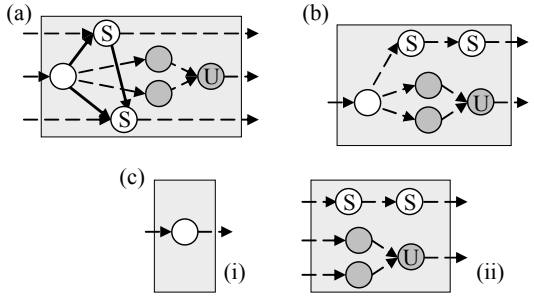

Figure 9. (a) POEG of process P2 after partial-order reduction, (b) transitive reduction and (c) subgroup partitioning.

The adjunction of edges in the POEG may cause some pairs of messages to be connected through several paths. Performing the transitive reduction of the POEG removes the superfluous edges (Figure 9b), and may enable a finer-grain subgroup partitioning. In Figure 9c, the message triggering the atomic step of the split operation now always occupies the first position and can therefore be used

to further partition the subgroup. With this final partition, the number of admissible message orderings is 1 for the subgroup (i), and 60 for the subgroup (ii), versus 168 orderings for the POEG of Figure 5.

Each subgroup is tested for message races by executing multiple different orderings, and by checking that the final process state and the set of generated messages are identical in all cases. The developer may decide for each subgroup whether to test all or only a subset of possible orderings.

5. Generating a subset of possible orderings

A single bug generally causes races in many different orderings. Within the subgroup of Figure 5 for instance, the error is revealed as soon as one of the two borders is sent after the subdomain update, which occurs in 24 orderings out of the admissible 168 orderings. We therefore suspect that most message races can be revealed by testing a small subset of carefully selected orderings, generated as described below.

Since we want to ensure that the final process state is identical for all orderings, and since the final state is determined by the last atomic step that modified it, we generate one ordering for each message such that the message is delivered as late as possible. Formally, given a subgroup of messages S we generate the following orderings (parentheses denote an ordered list):

$$\{(S \setminus \{a, Succ_a\}, a, Succ_a) \mid a \in S\}$$

This expression means that, for every message a in S, we generate an ordering where all messages other than the successors of a in the POEG of S are delivered before a.

The second requirement is that orderings produce a fixed set of messages. We therefore generate orderings such that every message is delivered right after every other message, and both are delivered as early as possible in order to test their influence on the following computations. The rationale here is to check that the output messages sent by triggered atomic steps are the same no matter which other atomic step (which could modify the process state) was executed right before. Formally, if $Pred_a$ is the set of predecessors of a in the POEG of S, and $Pred_{a,b}$ is the union $Pred_a \cup Pred_b$, we generate the following set of orderings:

$$\{(Pred_{a,b}, a, b, S\setminus\{a, b, Pred_{a,b}\}) \mid a, b \in S, a \neq b\}$$

The proposed algorithm guarantees that if the delivery order of two messages is not constrained by the flow graph, we generate at least one ordering containing (…,*a*,*b*,…) and one ordering containing (…,*b*,*a*,…). Note that we only consider orderings that are admissible according to the POEG of the subgroup. In consequence, less than $|S|\cdot(|S|-1)$ distinct orderings are generated in the general case, where $|S|$ is the number of messages in the subgroup S.

Although one may conceive an application whose bugs are not exposed when testing the sets of orderings defined above, such cases should occur very rarely in practice. In order to further reduce the probability that races remain undetected, we may additionally execute randomly generated orderings. Such a technique was successfully used for detecting data races in multithreaded applications [15]. By increasing the number of randomly generated orderings, we can arbitrarily increase our confidence that no message race exists.

6. Implementation and results

We implemented the message race detection within the Dynamic Parallel Schedules (DPS) framework [3]. We obtain the initial execution trace by executing the application once and logging the messages sent and the operations executed. We also use the checkpointing capabilities of DPS [4] to keep a copy of the initial state of each process. When the application completes, operations are decomposed into atomic steps, the partial order execution graph is derived from the logged messages and operations, and the partition into subgroups is performed.

In order to enable the partial order reduction described in section 4, we determine whether variables were modified or not by an operation by comparing checkpoints of the process state taken immediately before and after the operation execution. Within operations, process state variables are accessed through a function, which collects the list of variables accessed during the operation execution. Variables that are accessed but not modified are marked as read. The access vector of each operation is then constructed by combining the collected information for all the variables of the process state. All the messages delivered to the same operation share the same access vector.

Since the FIFO link property is not guaranteed by the DPS runtime, we do not consider this optimization in our study.

When the orderings of a subgroup are being tested, the initial process state is recovered from a checkpoint before the execution of each ordering. Another checkpoint is taken once all the atomic steps have been executed, and serves as a reference for verifying that all orderings lead to the same final process state. It also provides the initial process state for testing the next subgroup. Generated messages are checked against the reference execution as soon as they are produced. If identity is too strict a condition, custom comparison functions may be used to compare states and messages.

The integration within the framework allows the whole procedure to be performed without any modification of the source code. When an ordering leads to a different final process state or message set, the reference input and output messages, the initial process state and the ordering are written to stable storage together with the reference

ordering. The stored information can then be used to replay the ordering that caused the race. Since executions are replayed within a single multithreaded process, a conventional debugger can be used to study the erroneous computations.

6.1 Results

We present practical results for a few parallel applications. In order to measure the number of orderings as well as their length, the metric used for all measurements is the total number of atomic steps that must be executed to test all possible orderings. This number is obtained by summing for all subgroups the number of orderings allowed by their POEG multiplied by the number of messages within the subgroup.

We first quantify the benefits of the partial order reduction and subgroup decomposition using the neighborhood-exchange application illustrated in Figure 2. Table 1 compares the number of executed atomic steps for two iterations of the neighborhood-exchange computation, when no decomposition is performed, when using only the process decomposition (PD, section 3), when adding the partial order reduction (PD+POR, sections 3 and 4), and when performing the full subgroup decomposition (sections 3 and 4). We see that it is impossible to execute all orderings without decomposing the POEG of the application, even when it runs on only two processes. However, the proposed optimizations reduce the number of atomic steps to be executed by a factor of 10^{13}. In practice, testing all orderings for an application run of 6 iterations on 8 processes takes about 8 seconds on a 2.4GHz Pentium 4 processor.

We carry out the same analysis for a parallel implementation of the Floyd-Steinberg halftoning algorithm [9], which converts a grayscale image into a black and white image. 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 2 processes, the full decomposition reduces the number of atomic steps that must be executed by a factor of 10^7 compared to when no decomposition is performed. Testing all orderings for an application run on 8 processes takes about 105 minutes for a grayscale image of size 256 x 256 pixels.

Finally, Table 3 presents the results of our techniques running on a parallel block LU factorization application. The matrix size is given by n, while b specifies the size of a block within the matrix. Since the iterations of the computation are loosely synchronized in order to maximize the pipelining of the computation, subgroups contain many messages with little dependencies between each other, causing the number of atomic steps to be executed to explode: we could only compute it for n/b=3 and using the full subgroup decomposition with partial-order reduction. It remains however possible to test a subset of possible orderings using the algorithm described in section 5, as shown by the last line of Table 3. The partial test for a 160x160 matrix with n/b=5 takes about 30 minutes.

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 compared the results of the partial and full testing in all cases where the latter could be performed. In the applications presented here, testing the orderings produced by the algorithm described in section 5 was sufficient to find every message race. We also discovered a few genuine bugs in previous implementations of the LU factorization application.

Table 1. Total number of atomic steps to be executed in order to test all orderings (neighborhood-exchange application with two iterations).

	2 proc.	4 proc.	6 proc.
No decomposition	$5.6 \cdot 10^{16}$	-	-
Process decomposition (PD)	$2.4 \cdot 10^{5}$	$4.9 \cdot 10^{5}$	$10 \cdot 10^{6}$
PD+partial order reduc. (POR)	28840	65664	$9.4 \cdot 10^{6}$
Subgroup decomp. + POR	860	1932	12616

Table 2. Total number of atomic steps to be executed in order to test all orderings (parallel Floyd-Steinberg halftoning algorithm).

	2 proc.	4 proc.	8 proc.
No decomposition	$6.8 \cdot 10^{8}$	-	-
Process decomposition (PD)	848	$3.5 \cdot 10^{5}$	$4.0 \cdot 10^{12}$
PD+partial order reduc. (POR)	116	18576	$7.5 \cdot 10^{10}$
Subgroup decomp. + POR	42	1280	$6.8 \cdot 10^{6}$

Table 3. Total number of atomic steps to be executed in order to test all orderings or the subset defined in section 5 (pipelined parallel LU factorization).

	n/b =3	n/b =4	n/b =5
Subgroup decomp. + POR	226593	-	-
Subset defined in section 5	35822	$3.0 \cdot 10^{5}$	$1.4 \cdot 10^{6}$

6.2 Limitations

Using a single multithreaded process to test all orderings limits the size of instances that may be tested. In our experiments, storing the whole trace of the parallel Floyd-Steinberg application only requires 4.4MB when processing a 256x256 image on 8 processes, but looking for races

in the LU factorization application of a 160x160 matrix on 10 processes requires 350MB of memory. Since different processes are tested independently, we could test them in parallel, thereby further reducing running times and memory consumption. Since message races generally do not depend on the actual size or content of the processed data, it is generally sufficient to test an application using a small data set, which both reduces the running time and the amount of memory required to store the application trace. Data sets that trigger different execution paths within the application should be tested separately.

7. Application to a simple MPI application

The techniques described in this paper apply to any application that can be modeled using a POEG. We adapt an example and an event model from [16] to derive the POEG of a simple wildcard-free MPI application (note that we do not claim that all MPI applications can be modeled using a POEG). Figure 10 shows the MPI pseudo-code and the corresponding POEG for a neighborhood-dependent application similar to the one used throughout this paper. The model described in [16] distinguishes *send* and *receive* events. The edges of the POEG connect send events to their matching receive, and events from blocking MPI calls to the events of the subsequent call performed by the program. A send from process i to process j at iteration k is denoted as $s^k_{i,j}$, while $r^k_{i,j}$ denotes the matching receive event.

```
/* np: Number of processes, rk: Process rank */
/* maxIter: Number of iterations */
for (i = 0; i < maxIter; ++i) {
  if(rk==0) MPI_Send(lower border to proc. 1);
  else if (rk < np-1)
    MPI_Sendrecv(lower border to proc. rank+1,
                 lower border of proc. rank-1);
  else MPI_Recv(lower border of proc. np-2);
  if(rk==0) MPI_Recv(top border of proc. 1);
  else if (rk < np-1)
    MPI_Sendrecv(top border to proc. rank-1,
                 top border of proc. rank+1);
  else MPI_Send(top border to proc. np-2);
  /* update subdomain */
}
```

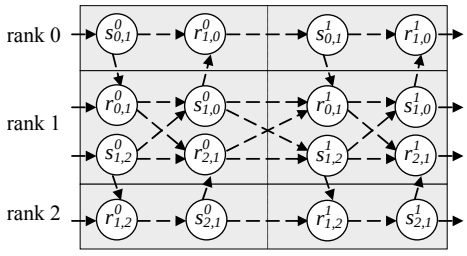

Figure 10. MPI pseudocode for a neighborhood exchange iterative computation, and corresponding POEG for two iterations on three nodes.

The ordering of events on individual processes is fully determined, except for pairs of *send* and *receive* events caused by *MPI_Sendrecv* calls. Nevertheless, the number of event orderings grows exponentially with 60, 6268 and $6.5 \cdot 10^5$ orderings for 1, 2 and 3 iterations on 3 processes respectively. Once the decomposition is performed however (Figure 11), the number of orderings grows linearly with the number of processes and the number of iterations. The number of orderings for each process with rank other than 0 and *np*-1 is $2 \cdot 2 \cdot maxIter$.

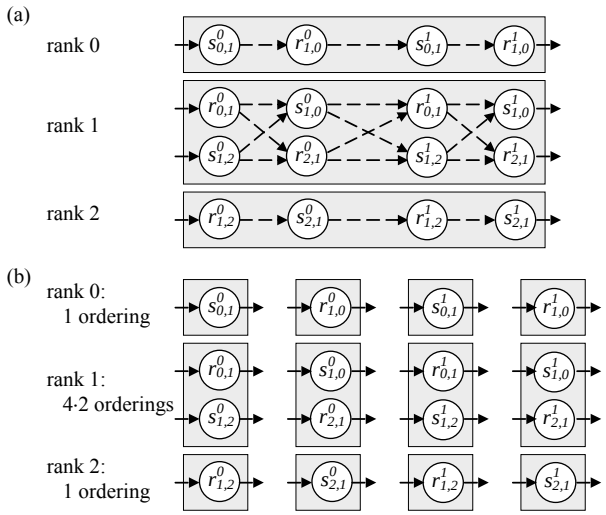

Figure 11. (a) Process partitioning and (b) subgroup partitioning and resulting number of orderings.

8. Related work

Mosbah and Ossamy [11] as well as Otta and Racek [13] detect message races by evaluating predefined predicates that consider both the local and the global state of the application at various points of the execution. Since no control is applied on the program execution, the detection can only work 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 [10] 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. [2] 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 [7] 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 essence, this is similar to our decomposition method, but no results are shown that would allow us to compare the two approaches. Moreover, none of these proposals considered access to local memory in their analysis.

Several authors argue that detecting the first message race is beneficial [12, 14]. Correcting early races not only removes subsequent instances of the same race, but also prevents potential spurious races from being enabled (e.g. a race exists because a prior race invalidates some assumption made on the data). Our method verifies subgroups in chronological order, and output messages are checked against the reference run as soon as they are sent. We can therefore determine the temporal location of every race, thereby allowing the developer to correct early races first.

9. Conclusion and future work

We presented a method to identify and prevent equivalent event orderings within a parallel application modeled as a Partial Order Execution Graph (POEG). We partition the POEG into smaller parts, firstly by distinguishing the sets of messages triggering computations in different memory spaces, and secondly by separating causally dependent message subsets. Leveraging information about how the computations triggered by each message read or modify state variables, we identify equivalent orderings within each subset. Equivalent orderings are prevented by adding edges to the POEG, which force the relative delivery order of the messages. We then showed how to generate a subset of orderings that are still able to reveal many potential errors.

We integrated the POEG decomposition and race detection methods within the DPS parallelization framework, where POEGs are easily derived. For three different parallel applications, we evaluated the influence of the proposed techniques on the total number of messages that must be delivered to the application in order to test all orderings. Since our approach is based on a static graph analysis, it cannot be applied to applications that may produce different sets of messages depending on the evolution of the computation. In the future, we intend to remove this limitation by performing the analysis dynamically during the execution of the application.

The DPS software is available on the Web under the GPL license at http://dps.epfl.ch. The latest version includes the determinacy verification techniques described in this paper.

10. References

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

[2] E. Duesterwald, R. Gupta, M. L. Soffal, Distributed Slicing and Partial Re-execution for Distributed Programs, *Lecture Notes In Computer Science; Vol. 757, Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pp. 497-511, 1992

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

[4] 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

[5] P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, PhD. thesis, University of Liege, Computer Science Department, 1994

[6] 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

[7] 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

[8] L. Lamport, Time, clocks, and the ordering of events in distributed systems, *Communications of the ACM*, 21(7), July 1978

[9] 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.

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] S. F. Siegel, G. S. Avrunin, Modeling wildcard-free MPI programs for verification, *Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*, pp. 95-106, 2005

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