Performance Simulation of Non-blocking Communication in Message-Passing Applications

David Böhme[1,2], Marc-André Hermanns[1], Markus Geimer[1], and Felix Wolf[1,2]

[1] Jülich Supercomputing Centre
Forschungszentrum Jülich, Germany
{d.boehme,m.geimer,m.a.hermanns,f.wolf}@fz-juelich.de
[2] Aachen Institute for Advanced Study in Computational Engineering Science
RWTH Aachen University, Germany

Abstract. In our previous work [1], we introduced performance simulation as an instrument to verify hypotheses on causality between locally and spatially distant performance phenomena without altering the application itself. This is accomplished by modifying MPI event traces and using them to simulate hypothetical message-passing behavior. Here, we present enhancements to our approach, which was previously restricted to blocking communication, that now allow us to correctly simulate MPI non-blocking communication. We enhanced the underlying trace data format to record communication requests, and extended the simulator to even retain the inherently non-deterministic behavior of operations such as `MPI_Waitany`.

1 Introduction

As a prerequisite for the productive use of state-of-the-art supercomputers, the HPC community needs powerful and scalable performance-diagnosis tools that make the optimization of parallel applications both more effective and more efficient. One major difficulty application developers are confronting with traditional performance tools is that the tools often diagnose only the symptoms of performance problems but not necessarily their causes. Often, the symptoms appear much later or on a different processor than the event causing it. The temporal or spatial distance between cause and symptom constitutes a substantial challenge in deriving helpful conclusions from a set of performance data.

In our earlier work [1], we have presented a simulator called SILAS (SImulation of LArge-Scale parallel applications) that can be used to verify hypotheses on causal connections between different performance phenomena at very large scales. The verification is accomplished by modifying event traces according to a hypothesis and using them to simulate the hypothetical message-passing behavior. The predicted behavior can then be scanned for wait states to investigate how the modification would influence (and hopefully reduce) their occurrence in various parts of the program. Typical questions the simulation can answer encompass how the performance behavior changes if a specific computation is

H.X. Lin et al. (Eds): Euro-Par 2009 Workshops, LNCS 6043, pp. 208–217, 2010.

more evenly distributed across the machine or if a specific communication operation is replaced or eliminated. The simulator performs a parallel real-time reenactment of the communication to be simulated using the original execution configuration. This eliminates the need for *modeling* communication and, thus, circumvents a major source of prediction inaccuracy.

So far, our simulator was able to replay only MPI blocking point-to-point and collective communication, but not non-blocking communication, as information on communication requests was not yet recorded in the trace data. In this paper, we outline extensions to the trace format and the simulator itself that allow us to correctly simulate all aspects of MPI non-blocking communication, thus making our simulation approach applicable to a much broader range of MPI applications. Special emphasis is given on the feature of retaining the inherent non-determinism exhibited by operations such as `MPI_Waitany` or `MPI_Test`, which may yield different results during simulation than they did during trace recording.

After discussing related work and briefly recapitulating the working principle of the simulator in the remainder of this section, we describe the required extensions to the trace format in Section 2. In Section 3, we present the basic approach for simulating non-blocking communication and mechanisms to retain non-deterministic behavior in the simulation. Finally, an experimental evaluation to demonstrate the scalability and accuracy of our approach is given in Section 4, before concluding in Section 5.

1.1 Related Work

The principle of trace-driven performance prediction has already been intensively studied. An early performance-analysis toolkit offering trace-based simulation capabilities as one element of a comprehensive feature catalog is AIMS [2], which estimates the scalability of parallel applications by extrapolating previously generated execution traces to higher numbers of processors and larger problem sizes. DIMEMAS [3] provides the ability to simulate the execution behavior of parallel programs based on previously generated event traces. The underlying prediction model allows the adjustment of relative processor speeds, network bandwidth and latency within and across nodes, the number of input and output links, and the processor scheduling policy. Predicting application performance for emerging architectures larger than those at one's disposal is the focus of BigSim [4]. BigSim combines an emulator that is capable of running larger numbers of virtual processes on a smaller number of physical processors with a post-mortem simulator that uses traces generated during an emulated run.

Compared to the approaches described above, our work clearly concentrates on the effects of fine-grained alterations of application-level behavior with respect to the performance under an identical execution configuration. The most important methodological difference is the use of a parallel real-time replay of the simulated communication at the original scale, which offers scalability advantages and relieves us of the burden of modeling the extremely complex communication infrastructures found on today's large-scale machines.

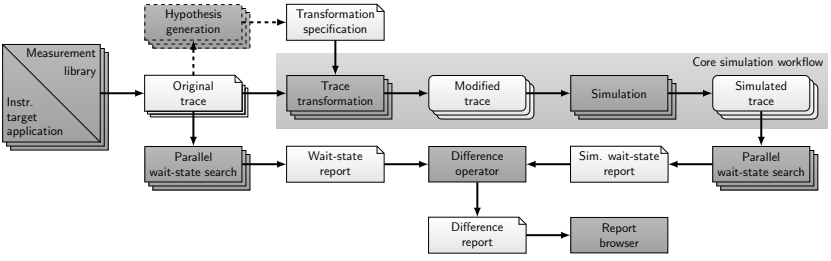


Fig. 1. Workflow for verifying optimization hypotheses. Dark rectangles denote programs, light rectangles with the upper right corner turned down denote files, and light rectangles with rounded corners denote data objects residing in memory. Stacked symbols indicate multiple instances of programs, files, or data objects running or being processed in parallel. The target application generating the event trace is the entry stage of the workflow. Judging the difference between normal execution and the predicted outcome of the optimization displayed in the report browser is the final stage.

1.2 Hypothesis Verification

Here, we briefly review the intended usage scenario for our simulator in the context of the Scalasca toolset [5]. Figure 1 illustrates the role of the simulator in the procedure of verifying hypotheses on causality between temporally or spatially distant performance phenomena. The general objective of the process is to generate wait-state analyses from both the measured and the predicted behavior and compare the results to allow conclusions on the effects of hypothetical program modifications with respect to wait states and other performance metrics. The workflow starts with running the instrumented target application in the execution configuration we want to make predictions for and generating an event trace consisting of one trace file per application process. During all subsequent steps, access to the event trace occurs through a parallel object-oriented high-level API [6]. The primary usage model of the API assumes a one-to-one mapping between application and tool processes, that is, for every process of the target application, one tool process is created which loads the corresponding trace data into main memory and offers random access to individual events. Data exchange among tool processes is accomplished via MPI communication.

A hypothesis includes the specification of a trace transformation, which may prescribe the adjustment of event timestamps, the deletion of existing events, or the insertion of new events to model changes in the application's source code. Currently, a set of parametrized standard transformations including the scaling of functions or the elimination of messages can be specified. After the transformation has been applied, the simulator performs a parallel real-time replay of the events stored in the trace. Computation intervals are simulated simply by elapsing the time in between using busy wait, whereas communications are simulated by reenacting the communication operations recorded in the trace. Thus, the time of a communication is determined by the time needed to execute the

corresponding MPI call under modified conditions. As the simulation progresses, event timestamps are adjusted to reflect the time elapsed since simulation start.

2 Trace Format Extensions

The trace format used by our performance analysis and simulation tools stores data in event records. There is a number of fixed event record types, for example for entering or exiting source code regions, or sending and receiving messages, respectively. Each event record contains a timestamp and, according to its type, other data such as the receiving location for send events or a source code region identifier for region-enter events. Event records are written consecutively in the order of the timestamps, with each process writing its own trace file.

In its previous form, our trace format did not provide explicit support for MPI non-blocking communication. Only send start events for `MPI_Isend` and receive completion events for `MPI_Wait*` or `MPI_Test*` regions were recorded using generic send/receive records. While this is sufficient to detect some communication inefficiencies (e.g., Late Sender) in our parallel performance analyzer, it does not allow accurate replay of communication as it is required for the simulation. In particular, neither information on the send completions and receive starts associated with the respective non-blocking send starts and receive completions, nor on failed tests for completion in `MPI_Test` is available in the trace.

2.1 Attribute Records

In order to enhance application traces with additional information at minimal impact on our current code base, we introduced the notion of *attribute records* to store additional, optional data for events. An event can have an arbitrary number of attributes, which are written as attribute records immediately before the corresponding event record in the trace. Unlike event records, attribute records do not contain a timestamp field, which keeps the record size as small as possible. Compared to the alternative approach of adding more fixed-size special-purpose event records, using attribute records to augment a particular event with additional information offers far more flexibility and better extensibility.

2.2 Non-blocking Event Record Types

For a full representation of non-blocking communication, we introduced *request IDs* to identify individual communication requests and to associate a request start with its completion. During trace recording, the opaque `MPI_Request` objects are mapped onto unique request IDs, which are stored in the trace for every non-blocking communication request start and completion.

While we added new event record types to store the request ID for receive starts and send completions, we continue to use the generic point-to-point send and receive record types for send start and receive completion events. Here, the request ID is stored in an attribute to the generic event. Using an attribute

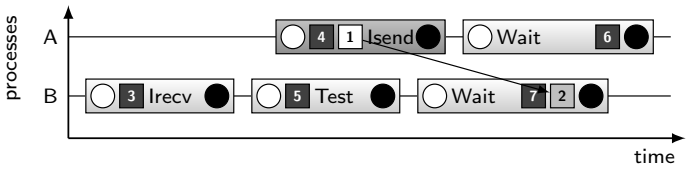


Fig. 2. Trace format extensions. Previous format: (1) send event; (2) receive event. Extensions: (3) receive start; (4), (7) request attribute; (5) test event; (6) send completion. White circles denote region enter events, black circles denote region exit events.

instead of new special-purpose event records keeps backward compatibility and allows us to reuse large portions of existing code in our analysis tools.

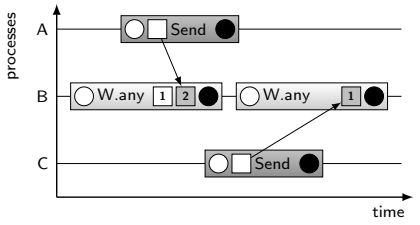
In addition to events for send completion and receive request, we also added a *tested* event, which indicates that a request has been unsuccessfully probed for completion in a call to `MPI_Test` or `MPI_Waitany/some`, and a *cancel* event which indicates a request that has been canceled using `MPI_Cancel`. Figure 2 shows the use of the new event records.

3 Simulation of Non-blocking Communication

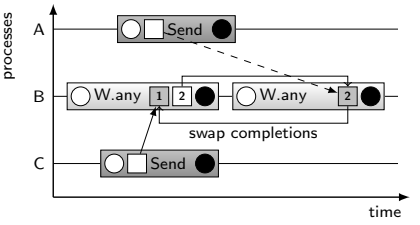
Given a trace enhanced with additional data as described in Section 2.2, replay of deterministic non-blocking communication in the performance simulator is now straightforward. When a non-blocking request start operation is encountered during trace replay, a corresponding `MPI_Isend` or `MPI_Irecv` operation is invoked, and the `MPI_Request` object obtained from MPI is saved in a (request ID, `MPI_Request`) map. Upon request completion, the requests corresponding to the request IDs found in the trace are completed using `MPI_Wait` or `MPI_Waitall`.

3.1 Retaining Non-deterministic Behavior in the Simulation

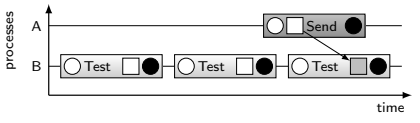
Our basic non-blocking communication simulation approach sketched above allows accurate simulation of non-blocking communication if only `MPI_Wait` or `MPI_Waitall` are used for request completion, but some difficulties arise for inherently non-deterministic operations like `MPI_Test` or `MPI_Waitany/some`. For example, a call to `MPI_Waitany` may yield a different result in the altered, simulated scenario than it did during trace acquisition. Likewise, a test for completion using `MPI_Test` which failed in the original run could succeed in the replay, or vice versa. Essentially, in an application scenario modified according to a performance hypothesis, the order and source code location of request completions can change compared to the original application's behavior, which may have a significant effect on the observed performance characteristics. Restricting request processing in the simulation to the order and locations found in the trace would therefore not accurately predict the application's communication behavior for non-deterministic operations. Hence, our non-blocking communication model needs to take reordering and relocating of request completions into account.



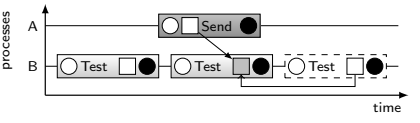
(a) Waitany in original trace: request 2 completes first



(b) In simulation, former request 1 completes first: Completion events are swapped, request ID 1 is remapped



(c) Test in original trace: request completes in second Test



(d) In simulation, test completes earlier. Completion is pre-drawn and additional test call removed.

Fig. 3. Retaining non-deterministic behavior of Waitany and Test

Simulating Waitany. For `MPI_Waitany` regions, the trace contains a completion event record with the request ID that completed in the original run, and *tested* event records with the request IDs that were also passed to the original call to `MPI_Waitany`. In the simulation, the request objects for all given request IDs are passed to `MPI_Waitany`. If the request that completed in the simulation is not the same as in the original run, we swap the completion events and *remap* the request, that is, the remaining events with the request ID that completed in the simulated run are mapped to the ID of the request that completed in the original run (Figure 3a and 3b). As a result, these test or completion events will now be handled for the request that completed originally. Since the positions of subsequent events in the trace pertaining to a certain request ID are known from a preprocessing step, the extra effort for request remapping is negligible. By allowing the simulation in `MPI_Waitany` to complete a request different from the one completed in the original run, we can accurately model the application's intended communication behavior ("return the first request which completes").

Simulating Tests. For calls to `MPI_Test` that are unsuccessful (i.e., do not complete a request), a *tested* event with the corresponding request ID is stored in the trace (Figure 3c). In this case, the simulator calls `MPI_Test` with the associated request. If the request does complete in the simulation, the completion is *pre-drawn*: the test event will be replaced with the requests' completion event, and all remaining test events with this request ID are deleted from the trace. If the last test or completion event remaining in a region is deleted, that region will be removed from the simulation altogether (Figure 3d).

More difficulties arise for `MPI_Test` calls which are successful in the original run, but fail in the simulation. In this case, the application would try to complete the request again later on. However, the simulation is bound to the trace that was recorded in the original run, which does not contain any information on how the application would have handled the request. Since there is no useful strategy for the simulator to process the request later if the `MPI_Test` call was unsuccessful, we explicitly complete a request using `MPI_Wait` if an MPI test operation succeeded in the original application run. This approach may, however, introduce some waiting time which would not have occurred in the modified application.

3.2 Limitations

While our simulator handles non-blocking communication well for most cases, there are a few noteworthy restrictions.

Non-deterministic operations pose a fundamental limitation on our simulator. Applications can take entirely arbitrary actions depending on the outcome of a non-deterministic operation, whereas our simulator is bound to the trace recorded in the original run. In some cases, our heuristics for retaining non-determinism by reordering and relocating request completions may fail to reproduce the application's behavior, or in extreme cases even deadlock. The user can therefore enable a deterministic simulation mode, which restricts request processing to the exact order found in the trace, at the cost of losing some simulation precision. It should be noted, though, that severe problems occur only for pathological cases exhibiting a highly unusual communication behavior. Typical communication patterns, such as looping `MPI_Waitany` on a fixed list of requests, work as expected. Genuinely reproducing the application behavior for different outcomes of `MPI_Test` operations is not possible without knowledge of the application semantics, which currently exceeds the scope of our replay approach. As such, our heuristic represents a best-effort approach which at least allows conclusions, e.g., on the number of tests needed to complete a request.

Also, our model currently does not handle persistent communication requests explicitly. Instead, they are handled as ordinary non-blocking communication, which may slightly overestimate the processing time for those requests in the simulation.

4 Results

We conducted a number of experiments to demonstrate accuracy and scalability of our approach, using small synthetic benchmarks and more complex real-world benchmark codes. All experiments were performed on the 72-rack Blue Gene/P supercomputer Jugene and the 448-core Power6 cluster Jump at the Jülich Supercomputing Centre.

4.1 Simulation Accuracy

One effective way of validating the simulation accuracy is an *identity simulation*, where a simulation run without any performance hypothesis applied is compared to the original program behavior. We conducted identity simulation experiments with `bt` from the NAS parallel benchmark suite [7] with 256 and 1024 processes on Jugene. The runtime of the benchmark kernel in the original, uninstrumented benchmark executable is compared with the runtime during trace recording and the simulated runtime. The results are shown in the following table.

Table 1. NAS `bt` measurement and simulation results

	Comm.	**Runtime (sec)**			**Deviation from Original**	
No. procs	fraction	Original	Traced	Simulated	Traced	Simulation
256	16 %	46.56	47.23	46.82	1.44 %	0.56 %
1024	30 %	14.93	16.57	15.67	10.98 %	4.96 %

Note that the deviation between original and traced runtime is about 2.5 times (256 procs) and 2.2 times (1024 procs, respectively) higher than the deviation between original and simulated runtime. Synthetic experiments confirm that the overhead of tracing is indeed higher than the overhead created by the replay of communication in the simulator. Especially small, short-running functions like `MPI_Irecv` can have a high relative tracing overhead.

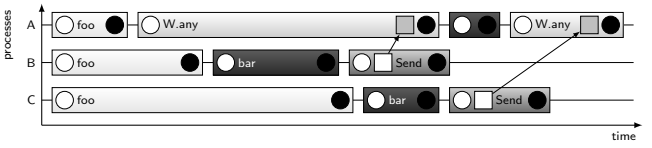
In general, inaccuracies introduced by tracing may also negatively influence the simulation, which is based on the trace. While the overall simulation accuracy for non-blocking communication in the presented case is good, the relation between simulation accuracy and measurement overhead during trace recording still requires further investigation.

4.2 Non-deterministic Behavior Simulation

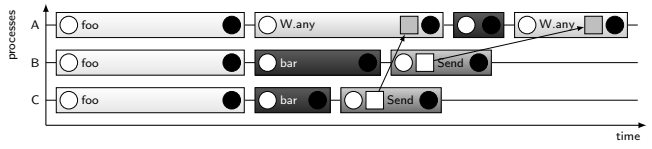
A simple synthetic benchmark demonstrates the necessity of retaining non-deterministic behavior in the simulation. Figure 4a outlines the working principle. The master process is waiting for messages from the remaining processes in a loop using `MPI_Waitany`. Due to a load imbalance in code region `foo`, the messages arrive in the order of process ranks.

We recorded an example trace of the program with four processes on our Power6 cluster and performed simulation runs both with request relocation and reordering enabled (non-deterministic mode) and disabled (deterministic mode). First, we performed an identity simulation, then another simulation with a performance hypothesis to balance code region `foo` applied. The result of the latter was compared to a modified version of the original program with `foo` balanced.

Table 2 shows the results of the experiments. While for the identity simulation, both deterministic and non-deterministic mode yield accurate results, only the non-deterministic mode is able to predict the program's behavior with region `foo`



(a) Load imbalance in `foo`: messages to arrive in order of ranks



(b) Balancing `foo`: message order is reversed on destination

Fig. 4. Waitany Benchmark: Original (a) and modified version (b)

Table 2. Non-deterministic behavior simulation: Deviation of simulation result from original (identity simulation) and modified program behavior (balance experiments)

	Runtime		**Identity simul. Δ**		**Balanced simul. Δ**	
Metric	original	modified	non-det.	det.	non-det.	det.
Total time	34.82 s	31.65 s	0.0003 %	0.0003 %	0.0689 %	8.14 %
Point-to-Point	5.60 s	2.40 s	0.0013 %	0.0009 %	0.0029 %	33.34 %
Late Sender	5.59 s	2.39 s	0.0041 %	0.0036 %	0.0083 %	33.36 %
Synchronization	2.70 s	2.70 s	0.0048 %	0.0011 %	0.0022 %	66.67 %

balanced correctly. This is because by balancing region `foo`, the order of message arrival in the `MPI_Waitany` call is reversed due to another load imbalance in code region `bar` (Figure 4b). By retaining non-deterministic behavior in our simulator, we can predict this effect correctly. In deterministic mode, however, the simulator is restricted to the original order of message arrival in the program, and therefore introduces larger waiting times.

5 Conclusion

We have presented enhancements to our performance simulator and underlying trace data format which allow us to accurately simulate the message-passing behavior of applications that utilize MPI non-blocking communication. Using both new event records and attribute records, we could amend application traces with communication request tracking capabilities requiring only minimal changes to the existing code base of our analysis tools. Moreover, attribute records may provide a generic and flexible approach to enhance application traces with additional, optional information. By reordering and relocating communication requests, our simulator can accurately predict even non-deterministic communication behavior for most typical communication patterns.

Further enhancements we plan to incorporate into our simulator are explicit support for persistent communication requests and support for MPI-2 one-sided communication. We are also investigating more detailed performance analysis procedures for non-blocking communication using the new request tracking capabilities in our parallel performance analyzer.

Acknowledgment

Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 and from the Helmholtz Association of German Research Centers under Grant No. VH-NG-118 is gratefully acknowledged.

References

1. Hermanns, M.A., Geimer, M., Wolf, F., Wylie, B.J.N.: Verifying causality between distant performance phenomena in large-scale MPI applications. In: Proceedings of the 17th International Conference on Parallel, Distributed, and Network-Based Processing (February 2009)
2. Yan, J., Sarukkai, S., Mehra, P.: Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. Software – Practice and Experience 25(4), 429–461 (1995)
3. Rodriguez, G., Badia, R.M., Labarta, J.: Generation of simple analytical models for message passing applications. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 183–188. Springer, Heidelberg (2004)
4. Zheng, G., Wilmarth, T., Jagadishprasad, P., Kalé, L.V.: Simulation-based performance prediction for large parallel machines. International Journal of Parallel Programming 33(2-3) (2005)
5. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable parallel trace-based performance analysis. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 303–312. Springer, Heidelberg (2006)
6. Geimer, M., Wolf, F., Knüpfer, A., Mohr, B., Wylie, B.J.N.: A parallel trace-data interface for scalable performance analysis. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 398–408. Springer,
Heidelberg (2007)
7. Bailey, D.H., Barzcz, E., Dagum, L., Simon, H.D.: NAS parallel benchmark results. IEEE Parallel Distrib. Technol. 1(1), 43–51 (1993)