Catching Idlers with Ease: A Lightweight Wait-State Profiler for MPI Programs

Guoyong Mao
Changzhou Institute of Technology
German Research School for Simulation Sciences
RWTH Aachen University
g.mao@grs-sim.de

David Böhme
German Research School for Simulation Sciences
d.boehme@grs-sim.de

Marc-André Hermanns
JARA-HPC,
RWTH Aachen University
German Research School for Simulation Sciences
m.a.hermanns@grs-sim.de

Markus Geimer
Jülich Supercomputing Centre
m.geimer@fz-juelich.de

Daniel Lorenz
German Research School for Simulation Sciences
d.lorenz@grs-sim.de

Felix Wolf
German Research School for Simulation Sciences
RWTH Aachen University
f.wolf@grs-sim.de

ABSTRACT

Load imbalance usually introduces wait states into the execution of parallel programs. Being able to identify and quantify wait states is therefore essential for the diagnosis and remediation of this phenomenon. An established method of detecting wait states is to generate event traces and compare relevant timestamps across process boundaries. However, large trace volumes usually prevent the analysis of longer execution periods. In this paper, we present an extremely lightweight wait-state profiler which does not rely on traces that can be used to estimate wait states in MPI codes with arbitrarily long runtimes. The profiler combines scalability with portability and low overhead.

Keywords

Performance analysis, wait states, profiling, MPI, Score-P

1. INTRODUCTION

The number of processor cores on modern supercomputers continues to increase at a rapid pace. In 2013, there was not a single system among the top ten of the Top500 list that featured less than a hundred thousand cores. However, exploiting all the available parallelism efficiently presents a major challenge. A common enemy of good performance that especially affects codes with irregular and dynamic domains is load and communication imbalance. Because imbalance delays processes before they can synchronize with other processes, classic symptoms created by this phenomenon are wait states, which are intervals during which a process sits idle without doing useful work.

In message-passing programs, wait states usually materialize in point-to-point or collective communication operations, including barriers. In an all-to-all operation, a single process can delay the progress of thousands of others. Moreover, wait states can easily propagate across process boundaries and proliferate [1]. Overall, their accumulated duration can constitute a substantial fraction of total resource consumption. As a first step towards eliminating the causes, we need a diagnostic tool to locate and quantify them. On the one hand, such a tool must be scalable and it should be sufficiently accurate, which implies that its overhead must be minimal. On the other hand, it should not be overly complex. In the same way as the power envelope of an exascale machine places a cap on its size, the manpower envelope of the HPC community limits the effort it can invest in developing and maintaining production-quality tools.

Because wait states cause temporal displacements between program events occurring in different processes, an established method used to diagnose wait states is event tracing. For example, Scalasca searches for wait states in MPI programs by measuring the time difference between matching communication and synchronization operations that have been previously recorded in event traces [3]. If the overhead of the instrumentation is kept low, for example, through the careful selection of instrumentation points, this method is quite accurate. On the other hand, the huge amount of trace data to be generated limits the scalability of tracing in terms of both the number of processes and the length of the execution interval being traced. The latter makes this approach less suitable for long-running codes with high temporal variability.

In this article, we describe how the extent of typical wait states can be determined without tracing. In our profiling-based approach, we compare the execution time of relevant communication operations with a scaled minimum measured across the entire execution, which provides fairly accurate estimates of typical wait states. Our method is in-

EuroMPI/ASIA '14, September 9-12 2014, Kyoto, Japan

http://dx.doi.org/10.1145/2642769.2642783.

herently scalable, introduces only negligible overhead, and works out-of-the-box regardless of how long the application is running. While the method itself is astonishingly simple, our main contribution is to demonstrate that—in spite of its simplicity—it provides good enough results to decide whether a code suffers from a considerable amount of wait states, and if so, to approximate their accumulated duration.

We start our discussion in Section 2 with a review of related work, followed by a high-level description of our approach in Section 3. In Section 4, we evaluate our approach experimentally with respect to accuracy and overhead. Finally, in Section 5, we summarize our findings and describe possible application scenarios and extensions.

2. RELATED WORK

The richness of the information stored in event traces makes them attractive targets in the pursuit of wait states. Such traces usually record the time when the program enters and leaves communication calls as well as communication parameters such as source or destination of a message and its size. Vetter [12] transforms these traces into a table of message exchange records with send and receive durations. Microbenchmarks with known wait-state patterns are used to train a decision tree that is later employed to classify message exchanges with unknown patterns. As the training is only valid for a specific hardware and software configuration, it has to be repeated for each new platform and runtime environment. Scalasca [3] identifies wait states by measuring temporal displacements between matching send and receive calls through a parallel replay of the communication recorded in the trace. Although trace-based methods offer opportunities for powerful analyses such as finding the delays ultimately responsible for the occurrence of wait states [1], their scalability is limited—especially with regard to the length of execution they can cover.

Trace-less methods like ours avoid these disadvantages, albeit at the expense of convenient access to global information. FPMPI [5] uses PMPI interposition wrappers to replace receive calls with a semantically equivalent sequence of calls. This sequence first probes continuously for incoming messages. Once a message arrives, FPMPI measures the time it spent probing, which corresponds to the extent of the wait state, and eventually invokes the actual receive call to complete the transfer. In a similar manner, FPMPI measures wait states in all-to-all operations by inserting a barrier in front of them. However, as we show in Section 4.2, the overhead of these extra operations is much higher than updating minimum statistics as needed in our approach. Some online tools [4, 9] use piggybacking to transparently send the timestamp of the send operation along with the original message. Comparing the remote with the local timestamp, the receiver can then diagnose wait states at runtime. There are three approaches to piggybacking: re-packing the original message and the piggyback data into a contiguous buffer; using a derived datatype based on the original memory locations; and sending separate messages. Unfortunately, the inability of collectives to support piggyback semantics makes the first two unsuitable for collective operations. In any case, general solutions for piggyback messages layered on top of MPI can significantly harm performance in specific application scenarios [11].

Finally, an MPI implementation may also choose to expose the relevant internal events directly. For example, MPI PERUSE [6] defines a callback interface to notify an application of events such as the start of the actual data transfer, which can serve as a basis for the estimation of wait states. To the best of our knowledge, this interface is currently only implemented by Open MPI and is therefore not a portable solution. Moreover, it only supports point-to-point messages. Although the new MPI tool information interface [10] of MPI-3 is expected to be more widely supported, it is rather unspecific about the type of information an implementation must provide.

In essence, our own approach retains Vetter's idea of classifying communication behavior, but with the difference that we train our classifier with the target application itself and during the same run it is supposed to be valid for. In addition, we classify only process-local summary information, which eliminates the need for trace files.

3. APPROACH

In this section, we introduce our lightweight profiling approach to the identification of wait states in MPI programs. Specifically, we focus on the well-known wait-state patterns *Late Sender* and *Wait at NxN* because our experience suggests that they play the most prominent role in practice.

As a prerequisite, we measure the duration of individual MPI calls as in a classic profiler. For this purpose, we intercept the events of entering and leaving MPI functions using PMPI interposition wrappers, take timestamps, and calculate the difference. However, the measured duration includes not only waiting time but also the time needed for the actual communication. Thus, we need a mechanism to distinguish between the two.

The core assumption of our approach is that the duration of wait-state-free communication calls is minimal. Following this idea, we determine the minimum duration for each (communication) call category across the entire execution. We define this as the actual processing time. Subtracting the minimum from the overall duration yields the waiting time. The call category can be, for example, the MPI function, further distinguished by function parameters such as message size or data type. Since our approach accumulates waiting time as opposed to reporting individual wait-state instances, it does not require event traces. Moreover, the accumulation makes it possible to defer the minimum subtraction until the end of the execution. All we need to do is to count calls, accumulate durations, and maintain minimum statistics in each category, some of which a profiler might do anyway. To minimize overhead and maximize scalability, we refrain from any global communication except for a small number of reductions immediately before finalizing MPI to calculate global minimum values. Otherwise, we consider only process-local information.

The implementation of our method was integrated into the call-path profiler of the performance measurement system Score-P [7]. This means that we report wait states separately for each call path of an MPI function. Although Score-P's current profiler is event-based (as opposed to sampling-based), our method would also work with hybrid profilers that sample user functions but still capture PMPI events. Below, we explain in detail how we recognize and quantify waiting time in different wait-state patterns. Since we assess the accuracy of our method using traces with their richer information base as a yardstick, we also describe how the same wait states can be found there.

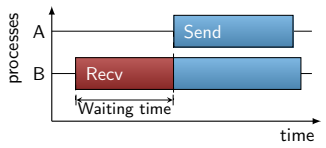


Figure 1: Late Sender. Because process A started the send operation too late, process B must wait in MPI_Recv.

3.1 Late Sender

The Late Sender pattern is depicted in Figure 1. In this point-to-point communication, the receiver is blocked while waiting for a message to arrive which has not yet been sent. If the destination process posts the receive operation before the source process initiates the matching send operation, we can conclude that the receiver has to wait at least until the send starts. This "pessimistic" assumption is the foundation of the trace-based Late Sender quantification in Scalasca, which we use as a reference. Scalasca simply measures the temporal displacement between the start of send and receive, creating a need to compare timestamps across process boundaries. Of course, the actual waiting time can be longer, but determining this extra time is impossible based on the data stored in the trace. In any case, the wait state of this pattern always appears in the receive operation.

In comparison to Scalasca, our method of approximating the true extent of wait states is much simpler. Everything in excess of the minimal duration of a call category is classified as a wait state. However, the key is how we define this minimum. That the time needed for processing a message can vary depending on various communication parameters, including message size, data type, source and destination process, confronts us with a tradeoff decision. The more parameters we consider, the more differentiated our minimum will become. But at the same time, the population from which a meaningful minimum can be chosen will also shrink, reducing the chance that we will see an instance without a wait state among them. After all, the processing time may also be affected by external factors such as OS jitter whose extent is invisible to us. Therefore, considering more parameters will not automatically improve accuracy. In addition, the space and effort needed for maintaining minimum statistics will grow as the number of parameters increases. Following our lightweight philosophy, we currently distinguish only between different receiving processes, MPI functions, and ranges of message sizes, the latter on a logarithmic scale. That is, each process p calculates local minima $M_p(f,s)$ for each combination of MPI functions f and range of message sizes s.

$$M_p(f,s) = \min_{c \in C_f} \min_{i=1}^{n(c,p,s)} d_i(c,p,s)$$

In the above equation, $n(c,p,s)$ denotes the number of times the call path c is visited on a process p with a given message-size class s. C_f is the set of call paths leading to the MPI function f and $d_i(c,p,s)$ denotes the durations of individual visits to these call paths. S represents the set of message size classes. Thus, the Late Sender time of a receive call path c on process p can be written as follows:

$$L(c,p) = \sum_{s \in S} \Big[\sum_{i=1}^{n(c,p,s)} d_i(c,p,s) - n(c,p,s) \cdot M_p(f(c),s) \Big]$$

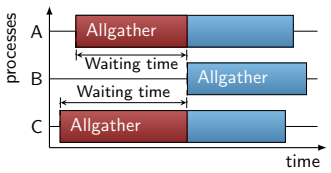


Figure 2: Wait at NxN. Because process B arrives late at a synchronizing all-to-all collective (here allgather), processes A and C must wait.

Here, $f(c)$ denotes the function terminating a call path c, in this case the MPI function for which we calculate the minimum. The way the formula is written also highlights the fact that the minimum calculation can occur in parallel to the accumulation of individual durations, making this a fully self-contained one-pass approach.

3.2 Wait at NxN

The Wait at NxN pattern is depicted in Figure 2. It can occur in all-to-all collective operations such as MPI_Alltoall, MPI_Allgather, or MPI_Allreduce, which exhibit inherent synchronization among all participating processes. Because none of the participants in such an operation can complete the operation before everyone else has entered it, wait states can occur if at least one participant is late. Scalasca models everything in the operation before the entry of the last process as waiting time, as shown in Figure 2. This is, of course, only an approximation because smart algorithms may already process data from a subset of the participants before the last process enters the operation and therefore achieve some progress. On the other hand, the PMPI interface does not provide enough insight to support a more fine-grained definition. Note that our current implementation considers only blocking collectives.

In our lightweight profiling scheme, we detect and quantify Wait at NxN in a similar manner to how we deal with Late Sender. The biggest difference is that we use the global instead of the process-local minimum. At runtime, all processes calculate the local minimum across all visits of the collective function, again differentiated by the size of the data. During finalization, the profiler computes the global minimum in a small set of all-reductions. Thus, for a program with a set of P processes, the Wait at NxN time of a call path c on process p can be written as follows:

$$W(c,p) = \sum_{s \in S} \Big[\sum_{i=1}^{n(c,p,s)} d_i(c,p,s) - n(c,p,s) \cdot M(f(c),s) \Big]$$

where $M(f,s)$ is the minimum for MPI function f and message size class s over all processes:

$$M(f,s) = \min_{p \in P; c \in C_f} \min_{i=1}^{n(c,p,s)} d_i(c,p,s)$$

Although choosing the local minimum for point-to-point and the global minimum for collective communication is more amenable to intuition, we also tried the alternate mode in each case and found it inferior. For point-to-point, the difference was minor, for collectives it was significant. Given that sub-communicators smaller than the world are more the exception than the rule, we refrain from considering anything between the local and the global minimum.

3.3 Further Patterns

Although judging from the experiences gathered with Scalasca the patterns we discussed so far are the most important ones in practice, there are other situations to which our method could be applied. For example, *Late Receiver* denotes a situation in which a sender communicating in synchronous mode is blocked while waiting for the matching receiver. Here, the wait state occurs in the send operation, which is why the local minimum of the send durations has to be used to calculate the waiting time.

Other collective patterns to which our method could be applied are *Late Broadcast* and *Early Reduce*. The first pattern denotes the situation where the root process of a broadcast arrives too late, therefore inducing wait states on those non-root processes that enter the operation earlier. Here, the minimum duration of MPI_Bcast on non-root processes becomes the relevant parameter. However, since the root process does not incur any wait states, the profiler has to distinguish between time spent in a broadcast when called as root and when not called as root. Conversely, the Early Reduce wait state can only occur on the root process, and only when it enters the reduction operation earlier than non-root processes. Therefore, the minimum duration of MPI_Reduce becomes relevant only on the root processes, requiring a similar distinction to that given above. Finally, one-sided operations with their weakly specified blocking semantics also seem natural candidates for our method and will be investigated in the future.

4. EVALUATION

Because our lightweight profiling approach estimates waiting time rather than measuring it directly, we evaluate its accuracy by comparing it to direct measurements. Moreover, we quantify its overhead in comparison to vanilla Score-P and the approach used in FPMPI, which we deem closest in spirit to our own. As test cases, we used the SPEC MPI2007 benchmark suite, the NAS Parallel Benchmarks (NPB), and Sweep3D. We ran Sweep3D and the NAS benchmarks with 1024 processes on both the IBM Blue Gene/Q system JUQUEEN and the Intel Xeon cluster JUROPA, both located at Jülich Supercomputing Centre. The SPEC MPI2007 benchmarks ran with 256 processes and only on JUROPA. To lower the instrumentation overhead, we filtered frequently executed, but otherwise uninteresting user functions.

4.1 Accuracy

To evaluate the accuracy of the minimum-based estimates of wait states, we compare it to the accuracy of trace-based measurements of the same wait states. For this purpose, we performed wait-state profiling runs during which we also generated a trace file. The trace files were analyzed using Scalasca to quantify the amount of wait states in each call path. Since pure minimum-based wait state profiling is naturally prone to jitter, we do not consider call paths whose waiting time amounts to less than 0.5% of the execution time of the whole program. We call this ratio between the amount of waiting time present in a call path and the execution time of the whole program the *wait ratio*, and use it as a comparison metric throughout this section. We devote separate subsections to Late Sender and Wait at NxN.

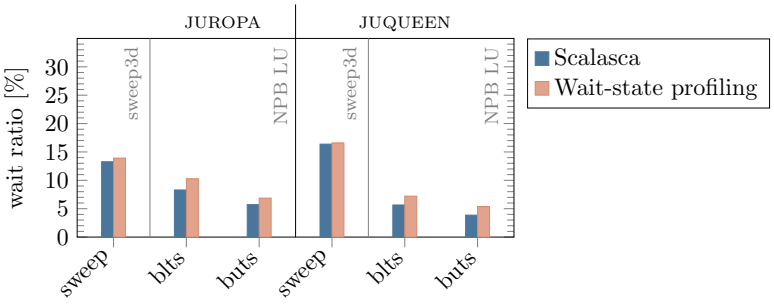
4.1.1 Late Sender

The wait-state quantification results for blocking and non-blocking communication differ. Blocking communication appears in LU from NPB and Sweep3D, all other codes use only non-blocking communication. Figure 3a compares the amount of waiting time we found using lightweight wait-state profiling and Scalasca trace analysis in MPI_Recv call paths. For the `blts` and `buts` call paths in LU, the Scalasca trace analysis reports a wait ratio of 8.2% and 5.7% on JUROPA whereas wait-state profiling reports 10.2% and 6.8%, respectively. On JUQUEEN, Scalasca reports 5.6% and 3.8% and wait-state profiling 7.2% and 5.3%. For Sweep3D, the absolute difference in the detected wait ratios is less than 0.7 percentage points. Although the wait ratios obtained using wait-state profiling are slightly higher, the difference between the two methods in MPI_Recv is small.

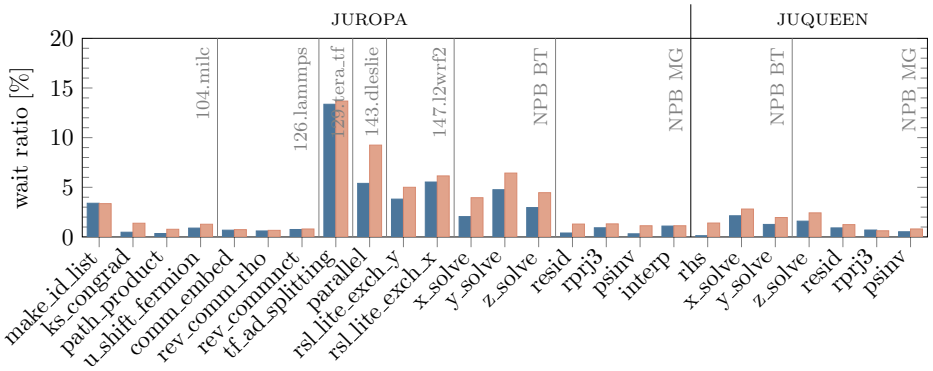
Figures 3b and 3c summarize our results for non-blocking communication, that is for call paths ending in MPI_Wait and MPI_Waitall. Except for SPEC's 143.dleslie, where wait-state profiling reports 3.8 percentage points more waiting time than Scalasca, the wait ratios for MPI_Wait do not differ by more than 2 percentage points. The numbers confirm the trend that the lightweight wait-state profiling slightly overestimates the amount of waiting time.

The accuracy for call paths ending in MPI_Waitall is lower compared with our observations for MPI_Wait on both systems, with differences of more than 2 percentage points in 7 cases. Moreover, SPEC's 145.lGemsFDTD is the only case where wait-state profiling *under*estimates the waiting time reported by Scalasca, likely due to a persistent static load imbalance in this benchmark. For the Waitall call paths in NAS BT and SP, wait-state profiling again significantly overestimates the waiting time on both JUROPA and JUQUEEN. Only 121.pop2 shows little difference between the two methods. While the minumim-based estimates for MPI_Wait are generally quite close to the measured values, especially if the wait ratio is high, the substantial discrepancy observed for MPI_Waitall suggests that our approach should not be used for this particular MPI function.

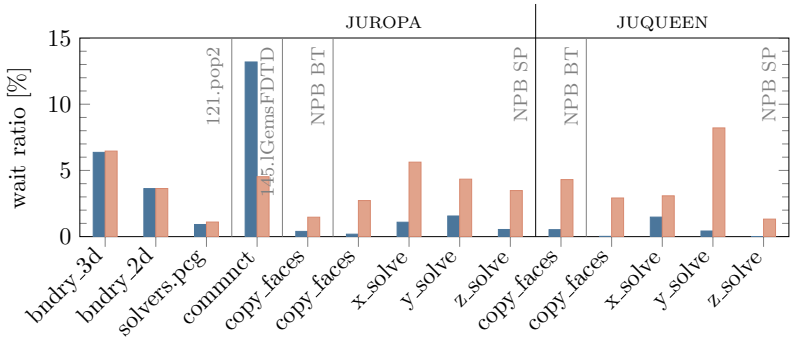
The reason why minimum-based wait-state profiling tends to overestimate the amount of waiting time, especially for non-blocking communication, is that a message may have already been (partially) received at the time MPI_Recv, MPI_Wait, or MPI_Waitall is called. In this case, the MPI function only copies the message in memory (if at all), which is much faster than the full sequence of receive actions. Unfortunately, such abbreviated receive calls are included in the minimum selection, leading to a minimum that is smaller and waiting times that are longer than desired. To alleviate this problem, we tried to exclude MPI_Wait and MPI_Waitall calls from the selection of the minimum and the calculation of waiting times if a preceding call to MPI_Test or MPI_Testall indicates that all requests are already complete. In the case of 143.dleslie, this reduced the difference between wait-state profiling and Scalasca's trace analysis by approximately 40%. In the case of SP on JUQUEEN, the difference was reduced by percentages ranging from 7.7% to 12.2%. However, messages that are nearly complete when the test is performed still render the minimum too small. In conclusion, while this variation seemed promising at first glance, it did not change the accuracy significantly enough to justify the extra overhead. Nevertheless, the minimum-based estimates for MPI_Recv and MPI_Wait still serve as a



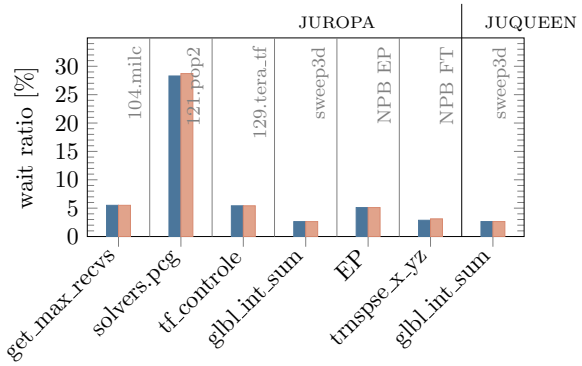
(a) MPI_Recv



(b) MPI_Wait



(c) MPI_Waitall



(d) Wait at NxN

Figure 3: Comparison of waiting times between Scalasca trace analysis and wait-state profiling for all call paths with a wait ratio greater than 0.5%.

good indicator of an enlarged wait ratio—even without the preceding test.

4.1.2 Wait at NxN

The quantitative results for all-to-all call paths are shown in Figure 3d. The absolute wait-ratio difference between the two methods is less than 0.45 percentage points and the relative distance less than 10% in all cases. Overall, we can conclude that minimum-based wait-state profiling estimates this type of wait state very well.

4.2 Runtime Dilation

For all test cases, the overall runtime dilation caused by Score-P instrumentation (frequently executed user functions filtered) with our method enabled did not exceed 15%, and was below 5% in most cases. To examine the additional overhead introduced by wait-state profiling compared to regular profiling in Score-P, we ran a series of microbenchmark experiments for the affected MPI calls (Recv, Wait/Waitall, and all-to-all collectives). On JUQUEEN, we find that the per-call overhead increases by less than 8% when collecting the extra statistics. Put into perspective, collecting traces instead of profiles (as required for the Scalasca wait-state analysis) increases the per-call overhead by 8 (point-to-point) to 20% (all-to-all collectives). In comparison, other on-line wait-state detection techniques lead to much higher intrusion. Modifying the MPI wrappers in Score-P to imitate FPMPI's methods of distinguishing between waiting and communication time increased the overhead for collective MPI calls by 88% and for MPI_Recv by 192%, respectively. Likewise, piggybacking increased the overhead for MPI_Recv in Score-P by 220%.

5. CONCLUSION AND OUTLOOK

In this paper, we presented a lightweight, inherently scalable, and platform-independent profiling technique to diagnose typical wait states in MPI point-to-point and collective operations, which we integrated into Score-P. It neither requires voluminous trace files, nor does it incur any noticeable extra cost in comparison to vanilla Score-P. The enhanced Score-P provides fairly accurate estimates of the fraction of waiting time in both blocking and non-blocking calls relative to the overall execution time. The exception is MPI_Waitall, where our technique often overestimates the waiting time significantly. Future work will aim at closing this gap. Further research will be devoted to the question of whether non-contiguous data types deserve special treatment. Moreover, we want to add support for additional wait-state patterns such as Late Receiver. In combination with phase profiling [8], our approach can help select intervals worth being traced and subjected to a more fine-grained investigation. Finally, it can benefit empirical performance modeling [2] by providing a means to separate waiting time from actual communication progress.

6. ACKNOWLEDGEMENTS

This project was supported in part by the Chinese Government under the Jiangsu Overseas Research and Training Program for University Prominent Young and Middle-aged Teachers and Presidents and by the US Department of Energy, Office of Science under Grants No. DE-FG02-13ER26158 / DE-SC0010668.

7. REFERENCES

[1] D. Böhme, M. Geimer, F. Wolf, and L. Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *Proc. of the 39th International Conference on Parallel Processing (ICPP), San Diego, CA, USA*, pages 90–100. IEEE Computer Society, Sept. 2010.

[2] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*. ACM, November 2013.

[3] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Computing*, 35(7):375–388, July 2009.

[4] M. Gerndt and M. Ott. Automatic performance analysis with Periscope. *Concurrency and Computation: Practice and Experience*, 22(6):736–748, 2010.

[5] W. D. Gropp, D. Gunter, and V. Taylor. FPMPI: A fine-tuning performance library for MPI. Poster presented at SC2001, available at http://www.mcs.anl.gov/research/projects/fpmpi/, Nov. 2001.

[6] T. Jones, editor. *MPI Peruse 2.0: A performance revealing extensions interface to MPI*. Mar. 2006.

[7] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of the 5th Parallel Tools Workshop, Dresden, Germany*, pages 79–91. Springer, Sept. 2012.

[8] A. D. Malony, S. S. Shende, and A. Morris. Phase-based parallel performance profiling. In *Proc. of the Conference on Parallel Computing (ParCo), Malaga, Spain*, volume 33 of *NIC Series*, pages 203–210. John von Neumann Institute for Computing, September 2005.

[9] O. Morajko, A. Morajko, T. Margalef, and E. Luque. On-line performance modeling for MPI applications. In *Proc. of the 14th Euro-Par Conference, Las Palmas de Gran Canaria, Spain*, volume 5168 of *LNCS*, pages 68–77. Springer, Aug–Sep 2008.

[10] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.0*, chapter 14: Tool Support, pages 555–590. MPI Forum, Sept. 2012.

[11] M. Schulz, G. Bronevetsky, and B. R. de Supinski. On the performance of transparent MPI piggyback messages. In *Proc. of the 15th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Dublin, Ireland*, volume 5205 of *LNCS*, pages 194–201. Springer, 2008.

[12] J. Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *Proc. of the 14th International Conference on Supercomputing*, ICS '00, pages 245–254, New York, NY, USA, 2000. ACM.