Monitor Overhead Measurement with SKaMPI

Dieter Kranzlmüller\textsuperscript{1}, Ralf Reussner\textsuperscript{2}, and Christian Schaubschläger\textsuperscript{1}

\textsuperscript{1} GUP Linz, Joh. Kepler University Linz, Austria  
\texttt{[kranzlmueller|schaubschlaeger]@gup.unl-linz.ac.at}  
\textsuperscript{2} Chair Computer Science for Engineering and Science, University Karlsruhe, Germany  
\texttt{reussner@ira.uka.de}

\textbf{Abstract.} The activities testing and tuning of the software lifecycle are concerned with analyzing program executions. Such analysis relies on state information that is generated by monitoring tools during program runs. Unfortunately the monitor overhead imposes intrusion onto the observed program. The resulting influences are manifested as different temporal behavior and possible reordering of nondeterministic events, which is called the probe effect. Consequently correct analysis data requires to keep the perturbation a minimum, which defines the need for monitors with small overhead. Measuring the actual overhead of monitors for MPI programs can be done with the benchmarking suite SKaMPI. It's results serve as a main characteristic for the quality of the applied tool, and additionally increase the user's awareness of the monitoring crisis. Besides that, the measurements of SKaMPI can be used in correction algorithms, that remove the monitoring overhead from perturbed traces.

1 Introduction

Software analysis is concerned with detecting errors and performance bottlenecks in programs. Since program execution is of main interest, most analysis activities rely on process state information that is generated by observation tools during representative program runs. Yet, this observation also introduces problems, which are summarized as the probe effect\textsuperscript{2}. This means that monitoring a program influences its behavior, and the generated analysis data does not represent the same execution, as if monitoring is turned off.

The monitor perturbation takes place in time and space\textsuperscript{7}, because it depends on the time spent in monitoring functionality and the amount of memory required for the monitoring code and the observation data. The memory consumption is not a problem as long as enough memory resources are available. More critical is the timing perturbation, which occurs in two areas: Primarily, the occurrence time of each observed event is delayed by the amount of time spent in the monitoring functions. Additionally, the different event times may sometimes lead to variations in event ordering if nondeterministic programs are considered.
The consequences are manifold. Firstly, the observed execution is different from the same program run without observation. Therefore it is difficult to connect both executions and obtain the required information. Secondly, the observation may even introduce a completely different execution path and therefore different results. Additionally, errors and bottlenecks may be hidden in paths not taken by the monitored execution, while errors may be introduced, that would never occur in the original program.

It is therefore required to raise the user’s awareness about the probe effect and to integrate solutions in corresponding analysis tools. Yet, so far tool developers either neglected the importance of the monitor overhead or, if they included measurements, they did not provide information about how these measurements were performed. With SKaMPI [9], the Special Karlsruhe MPI-Benchmark, we provide an approach to standardize these measurements, which allows to compare different monitoring tools. While SKaMPI was originally intended for benchmarking of MPI implementations, it can easily be extended for other measurements as described in this paper.

The paper is organized as follows. In the next section we define our target of investigation, which are nondeterministic MPI programs. For simplification we will focus only on point-to-point communication. Afterwards we define the amount of monitor overhead for a generally valid and abstract monitoring approach. Section 4 introduces SKaMPI and its extensions for monitor overhead measurements as well as some results for an example implementation.

2 Nondeterministic MPI Programs

The main focus of our research are parallel programs based on the standard Message Passing Interface MPI [8]. In MPI parallelism is achieved with multiple instances of (possibly the same) code that are executed concurrently. Synchronization and communication is performed explicitly via dedicated functions provided by the corresponding MPI library. For example, to exchange messages between two processes one could use the basic functions `MPI_Send` and `MPI_Recv`.

\[
\text{MPI\_Send}(\text{buf, count, datatype, dest, tag, comm})
\]
\[
\text{MPI\_Recv}(\text{buf, count, datatype, source, tag, comm, status})
\]

With these functions, the message at memory location `buf` is transferred between two processes, if the variables are set accordingly: At the sending process `dest` points to the receiving process, while at the receiving process `source` identifies the sender. Additionally the parameter `tag` must fit, which serves as an indicator for the contents of the message. Other parameters are provided for additional functionality.

As extension to the standard functionality the definition of `MPI\_Recv` introduces nondeterminism, because it is possible to accept messages originating from more than one process at a particular receive. This is achieved with wildcards, where the identifier for the message `source` is specified as `MPI\_ANY\_SOURCE`, while the tag may be defined as `MPI\_ANY\_TAG`. In fact, if there is more than one
message racing towards a receive, the order of arrival may be determined by system characteristics like processor speed, scheduling and caching strategy, or contention on the communication network. The behavior of the process after the receive may be affected and is therefore unpredictable, even if the same input is provided for the program. Yet, due to performance improvements such behavior may be requested by the user, e.g. in case of implementing a FIFO queue.

Besides its useful features, nondeterminism also introduces a major drawback for observation tools. Since monitors perturb the target programs, the occurrence of messages at wild card receive events may be influenced. It is possible, that the arrival order of messages in the original program is completely different from the arrival order in the monitored execution. Additionally, the results of the program are possibly changed, if the program code after the wild card receive depends on the arrival order. As a consequence, errors may be hidden in branches of the code, that are not taken if the program is observed. Furthermore, less critical errors may be emphasized, which may possibly never occur with the original program.

3 Monitor Overhead

The problems as described above occur, whenever an observer is added to a system, and are therefore not solely connected to message-passing programs. To react to the probe effect’s implications, it is necessary to identify the amount of overhead that is generated by the observer. Only with known intrusion it is possible to reduce the generated overhead, to provide tools for overhead correction, and to increase the user’s awareness of this problem. To identify the amount of overhead, it is important to determine the places where monitoring occurs. Of main interest is the group of events, that are observed during execution. An event is an action without duration that takes place at some time during the execution and changes the state of a process [6]. The reason for an event is a programming statement, that has been placed during the implementation. Clearly, not every programming statement might be interesting, and thus, the numbers of observed events is limited and defined either by the user or the applied analysis tool.

For the remainder of this paper we will focus on communication events in message-passing programs, especially those generated by the MPI_Send and MPI_Recv function as described in section 2. The important questions are, how monitoring code is added to these function calls during the instrumentation phase, and where monitoring functionality interferes with the application's execution.

Instrumenting a program can be done in various ways. One example is to manually or automatically substitute interesting function calls with corresponding calls from the monitoring library. More sophisticated techniques, like binary wrapping [1], may apply changes by patching object files. Another solution is the profiling interface defined by the MPI standard itself, which requires a simple name-shifting convention from any MPI implementation [8]. It allows to inte-
grate monitoring functionality without the need to access the MPI source code.

![Diagram](image)

**Fig. 1.** Monitor overhead in communication functions

After adding monitoring code to the target application, activities of the monitor are performed whenever an instrumented statement or function call is executed. In principle, the interaction between monitor and application is as described in Figure 1. Instead of calling the MPI library directly from the application, the application calls the monitor function, which on the other hand initiates the original MPI function. After the MPI function is finished, it returns again to the observer which notices the end of the function and performs analysis of the return parameters and error codes. This defines the two areas of monitor activities, the init part and the exit part as displayed in Figure 1. Please note, while it is not necessary to perform monitoring functions before and after the original function call, it is quite common in existing monitors to detect successful function completion.

As soon as the application has been instrumented and an execution has been performed, observation data is available. Additionally, the produced monitor overhead can be evaluated. The amount of the overhead is mainly determined by the functionality that is performed in the init and the exit part, which can be characterized by [3]:

- the amount of data to be processed
- the number of measurements to be performed
- whether source code connection has to be traced or not
- the connection between the monitoring tool and the observer

The amount of data and the number of measurements is clearly defined by the requirements of the analysis task. For example, if users are only interested in a correct logical communication structure, no timing measurements need to be performed. Similarly, the source code connection is only needed in some cases.

Another main characteristic is the connection between the monitor and the analysis tool. While the monitor has to be active during program execution, the analysis task may be performed concurrently or at another time and possibly at another location. This defines the two possible connections: on-line and post-mortem. The advantage of on-line connection is that program behavior and monitor characteristics can be interactively determined by the user. Yet,
this may also mean a bigger overhead, since user interaction may delay the program significantly. On the other hand, post-mortem mechanisms require a lot of storage space and permit only limited program steering. However, since it is not possible to estimate user activities, we focus mainly on post-mortem connections for now.

The monitor overhead can be determined either for each event or for a complete execution. Since one type of event usually requires almost the same monitoring functionality during each execution, the overhead of an event can be measured independently from the target program. To determine the overhead, we define the time of a generic communication event as follows:

\[ t_{\text{event}} = t_{\text{startup}} + d_{\text{msg}} \cdot t_{\text{transfer}} \]

In this case, \( t_{\text{startup}} \) is the necessary time to prepare the message transfer, while \( d_{\text{msg}} \) is the number of elements (usually bytes) in the message and \( t_{\text{transfer}} \) is the amount of time necessary to transfer one element. The main difference between these two times is, that \( t_{\text{startup}} \) will always delay the process, while \( t_{\text{transfer}} \) may occur in the background. Additionally, there may be some blocking time, which is not relevant for determining the overhead.

When executing the program with the monitor, the time \( t_{\text{event}} \) is increased by the functionality performed in the monitor’s init and exit parts. Obviously, most of the monitoring functionality will delay the process and is therefore added to the startup-time. Furthermore, the monitor will affect the transfer-time, for example, if a time-stamp or a vector-clock has to be transferred together with the message. Thus, the time for each event during monitoring can be computed as follows:

\[ t_{\text{event}} = t_{\text{startup}} + t_{\text{mon}} + (d_{\text{msg}} + d_{\text{mon}}) \cdot t_{\text{transfer}} \]

Here \( t_{\text{mon}} \) is the amount of time spent during the monitor’s activities, while \( d_{\text{mon}} \) is the amount of monitoring data added to the transferred message. Thus, the monitor overhead per event can be defined as follows:

\[ t_{\text{overhead}} = t_{\text{mon}} + d_{\text{mon}} \cdot t_{\text{transfer}} \]

Computing the complete overhead of a process is then based on the overhead accumulated from all events occurring on that particular process during the program’s execution. Clearly, this overhead depends on the implementation, the computation path selected during the execution, and the underlying hardware architecture.

Defining the overhead for a complete program is more difficult. Firstly, there is the question about how to combine the overheads achieved on all processes. Adding all process overheads is one possibility, but it neglects the fact that processes (and therefore monitoring functionality) is executed concurrently. On the other hand, the processes’ overheads can be combined by computing minimum, average, and maximum values, which may fit the users’ requirements better.
Secondly, there are some problems, which may invalidate all the overhead computation and prohibits a useful comparison between the program's execution with and without the monitor. One problem is, that the overhead may overlap with waiting times, and may therefore be not critical at all. Another problem is associated with network traffic, where it is possible, that less network contention is achieved if the program is running with monitor. This could even mean, that an instrumented program may execute even faster than it's corresponding original program.

4 Benchmarking with SKaMPI

To provide a standard method for evaluating and comparing different monitor approaches, we applied SKaMPI [9], the Special Karlsruhe MPI-Benchmark. SKaMPI measures the performance of MPI [10] implementations and of the underlying hardware. Its primary goal is to support software developers, since knowledge of MPI function's performance has several benefits, such as: shorter performance tuning after programming, consideration of performance issues during the design stage, and program development independently of the target platform.

MPI performance knowledge is especially important, when developing portable parallel programs. Then the code can be adjusted for all considered target platforms in an optimal manner. As a result performance portability can be achieved, which means that code runs without time consuming tuning after recompilation on a new platform.

Basically, there are two ways to use SKaMPI for new measurements:

(a) **Instrumenting an applications with SKaMPI functionality.** For this purpose we developed the library SKaLib. In this library we packaged the mechanisms of SKaMPI, such as precision adjustable measurement of time, controlled standard error, automatic parameter refinement, and merging results of several benchmarking runs. The advantage of this approach is, that SKaMPI mechanisms can be reused as needed. In addition, SKaLib is usable when developing sequential benchmarks, since it does not require MPI.

(b) **Enhancing SKaMPI.** It is easy to add new functions to be measured to SKaMPI. Choosing this way means, that the whole SKaMPI infrastructure could be reused, which proved useful in our case. In principle, our approach is as follows. First, we measure the function of MPI without applying a monitor. Secondly, we instrument the MPI code to insert the monitor into the program and repeat the measurements. Afterwards the overhead of the monitor can be evaluated by comparing both results.

The remaining question is concerned with the kind of required measurements. Clearly, this depends on the statements that are instrumented. Corresponding to the definitions in section 3, we can identify two different kinds of tests. On the one hand, we have to evaluate the delay occurring on each process, which we
called $t_{mon}$ before. This can easily be achieved by measuring the time required for a function call to a communication function.

On the other hand, we require the increased transfer times, which are defined by $d_{mon}$. Contrary to the delay on one process, there are two processes associated with this problem. Therefore, we have to perform ping-pong measurements, where one process sends a message to its communication partner, which returns the message immediately. The communication time is then computed by removing the on-process delays for a send and receive function and dividing the remaining time by two, because message transfer occurs twice.

5 Example and Results

This section provides first experimental results of determining the results for a particular monitor system. The system under consideration is the record & replay module NOPE of the Monitoring And Debugging environment MAD [4], which is a test set for performance analysis and error detection in message passing parallel programs. Since the measurements of NOPE's overhead also depend on the implementation of MPI on the underlying operating system and hardware architecture, we performed our measurements on the SGI/Cray Origin 2000 with its native MPI implementation and the current version of the Cray Message Passing Toolkit.

![Graph](image)

Fig. 2. SKaMPI measurements of monitor overhead at MPI_Send (left) and monitor overhead for ping-pong communication (right)

The total overhead of a monitor depends on the overhead of the monitor at each of the communication functions. To determine this overhead for an example, we selected the function MPI_Send as displayed in Figure /reffig:ovvsend. The horizontal axis displays the number of bytes that have been sent, while the vertical axis displays the time required for that particular function call. The two curves display the times of MPI_Send, once without any observation, once with the monitor NOPE applied. Consequently, the overhead of each function is the difference between both curves.

Additionally, we determined the overhead during message transfer. As mentioned before, this can be attributed to the additional data being required for
monitoring purposes. The applied measurements are ping-pongs between two corresponding MPI\texttt{Send} and MPI\texttt{Recv} function calls. The results are displayed in Figure 2. Again, the upper curve shows the ping-pong communication with the monitor overhead, while the lower curve represents the original performance.

Please note, that after removing the on-process delays from the curves of Figure 2, we found out, that the monitor overhead established during message transfer can be neglected. This stems from the fact, that the target monitor implementation applies Lamport clocks [5], and only one additional byte has to be transferred.

6 Conclusion and Future Work

The importance of measuring the monitor overhead is well-known in the tool developers domain, but has either been ignored or not been defined as much as it might be required in some cases. With SKaMPI it is possible to perform standardized benchmarking of MPI monitors and compare those measurements to obtain a characteristic criterion for the quality of different monitor implementations. As a side-effect, these measurements increase the user’s awareness of the monitor overhead and it’s implications for program analysis.

The future work in this project covers two main issues: On the one hand, the presented features of SKaMPI will be integrated and made available for other developers to measure their tools. Only by measuring several tools a comparison between different approaches is possible. Another interesting idea would be to measure one monitor approach available for different platforms, like for example tools based on the On-line Monitoring Interface Standard OMIS [11].

On the other hand, we will use the results of SKaMPI in a utility for monitor overhead removal, which covers timing delays of event occurrences and event reordering. So far, our work in this area has been performed with stand-alone measurements, which were rather difficult to adapt to the variety of available systems. Therefore it would be a big improvement in terms of standardization, if SKaMPI and it’s monitor overhead measurements can be used in the scope of this project.

Remarks Public versions and further details of SKaMPI are available at:

\url{http://wwwipd.ira.uka.de/skampi}

References