Performance Profiling Overhead Compensation for MPI Programs

Sameer Shende[1], Allen D. Malony[1], Alan Morris[1], and Felix Wolf[2]

[1] Department of Computer and Information Science, University of Oregon
{sameer, malony, amorris}@cs.uoregon.edu
[2] Innovative Computing Laboratory, University of Tennessee
fwolf@cs.utk.edu

Abstract. Performance profiling of MPI programs generates overhead during execution that introduces error in profile measurements. It is possible to track and remove overhead online, but it is necessary to communicate execution delay between processes to correctly adjust their interdependent timing. We demonstrate the first implementation of a onlne measurement overhead compensation system for profiling MPI programs. This is implemented in the TAU performance systems. It requires novel techniques for delay communication in the use of MPI. The ability to reduce measurement error is demonstrated for problematic test cases and real applications.

Keywords: Performance measurement, analysis, parallel computing, profiling, message passing, overhead compensation.

1 Introduction

When a parallel program is profiled, measurement operations generate overhead that affects the performance observed. We call this *performance intrusion* [2]. Performance profiling tools typically report intrusion as a percentage slowdown of total execution time, but the intrusion effects themselves will be distributed throughout the profile results. While performance intrusion can alter program execution and, thus, perceived performance (i.e., *performance perturbation*), performance profiling tools rarely attempt to adjust the performance measurements to compensate for the intrusion.

In earlier work [3], we present a technique to measure overhead on each process of a parallel computation and remove its local effects. But these are not the only effects overhead intrusion can cause. Due to inter-process communication, the delays introduced by intrusion will propagate between processes. In more recent work [4], we specify models for parallel overhead compensation and the algorithms that must be used when profiling message passing parallel programs. These models show why it is necessary to communicate intrusion delays with every message communication. However, this is not so easy to accomplish.

This paper presents our results for the implementation of the parallel profile overhead compensation models in an MPI environment. Such overhead compensation techniques have never been implemented before. Here we outline our approach to piggyback intrusion delay on messages. We demonstrate the technique with applications and show that measurement error due to overhead can be removed, locally and globally.

Section §2 provides a brief background on the problem. Our solution approach is presented in Section §3. Section §4 outlines our experimental environment and shows the results of our validation tests. Conclusions and future work are given in Section §5.

2 Problem Background

Events are actions that occur during program execution. Typical events include *interval events* that are characterized by a pair of actions marking *entry* and *exit*, and *atomic events* that occur at a single place or time. Tools insert measurement code to track the performance of a parallel program as made visible by the instrumented events. Executing the measurement code introduces overhead. If an event trace is collected, there are techniques to analyze the trace and compensate the measurement overhead [7,8,9,10], including the correction of perturbation effects.

Unfortunately, this type of analysis is not possible with profile-based measurements where all compensation decisions must be made at the time the event occurs, more or less. This raises the problem of how to track delays between processes. Basically, measurement overhead occurring on one process affects events on other processes that are causally related [12]. Consider a process that executes a measured routine a large number of times (and thus incurs a large overhead associated with the entry and exit instrumentation), and then sends a message to another process that blocks waiting for the message. If the receiving process has accrued little measurement overhead before the receive operation, the message receipt will be delayed. Without the receiver knowing that that delay was due to the measurement overhead in the sender, the receiver's profile will end up accounting for the sender's overhead as *its waiting time*, when in reality, it may not have waited at all.

To accurately re-construct events in all the processes, we must account for the time spent executing instrumentation calls in the sender process and subtract this time from the wait time in the receive, if any. To do so, we propose a scheme where local delays are propagated along with inter-process communication events in the form of *piggyback* messages. The delay value represents how much sooner a process would have executed the given communication operation if there was no measurement overhead in the process. The receiving process extracts this piggyback message and adjusts its local delay. What is interesting is that the adjustment cannot be any greater than the waiting time for the *current* receive.

Consider two cases. In the first case, the remote delay is equal to or exceeds the sum of the waiting time and the local delay. In this case, the waiting time in the absence of instrumentation is zero, as the message would be received as soon as the receive call is executed. In the second case, the remote delay is less than the sum of the local delay and the waiting time. Here, depending on whether the remote delay is less than or greater than the local delay, the uninstrumented waiting time may be more or less than the current waiting time. On receiving the piggyback message, the receiver compares the local delay with the remote delay. The adjustment of wait time is the difference in the remote delay and the local delay: $Adjustment = remote(delay) - local(delay)$.

This adjustment is subtracted from the original wait time to give the new waiting time: $W_{new} = W - Adjustment = W - (remote(delay) - local(delay))$.

If we consider the beginning of the wait time and the receipt of the message as two events, then, if there is no delay in the local receiving process, the point that corresponds to the beginning of the wait routine is shifted to the left by the amount equal to the local delay. And the point where the message is received shifts by an amount equal to the remote delay. The distance between the two points is the new waiting period. The adjustment (or the difference between the remote and local delays) may be positive, negative, or zero corresponding to when the remote process experiences more, less or the same delay as compared to the local process. Correspondingly, the wait time may decrease, increase, or remain the same respectively, but it can never be adjusted to be negative. This adjustment of waiting time is propagated along the callstack of the receiving process, so the inclusive time spent in all ancestor routines is adjusted accordingly. This is a necessary calculation in order to properly compute profiling measurements. When we compute the local delay (at both sender and receiver processes), we assess the measurement overhead and then subtract the waiting time adjustments that have been made in the program. This value is then sent along with a message. Thus, delays from one process reach all processes that have causally related events.

The full details of our parallel profile compensation algorithms are described in our earlier paper [4].

3 Implementation

To test our models of parallel overhead compensation, we built a prototype using the TAU performance system [5] and the Message Passing Interface (MPI). Our goal was to produce a widely portable prototype that could be efficiently implemented and easily applied. We chose MPI as the communication substrate due to its wide acceptance in the parallel computing community as the de-facto message communication standard, as well as due to its portable tool support.

3.1 MPI Profiling Support

MPI supports creation of portable profiling and tracing tools using its profiling interface, PMPI. This interface allows a tool to interpose a library between the application and the MPI substrate and intercept one or more MPI calls. MPI provides a name-shifted interface to all its calls. For example, an MPI call such as `MPI_Send()` is also available as `PMPI_Send()`. Both are guaranteed by the MPI standard to provide the same functionality. Furthermore, if a tool defines an `MPI_Send()` call, it takes precedence over the MPI library's `MPI_Send()` call (this is done by using weak bindings for defining the library's calls). The tool can then define one or more MPI bindings and create measurement timers and start and stop them around the name-shifted version of the corresponding MPI call. Every MPI implementation must implement this profiling interface to conform to the MPI standard. This mechanism allows vendors of parallel systems to optimize the implementation of MPI to their target platforms and at the same time expose the hooks for tracking MPI performance to tool builders without providing them access to their proprietary source code.

3.2 Schemes to Piggyback Delay

To transmit the local delays encountered in a process (due to program instrumentation) to other processes, we examined several alternatives. The first scheme modifies the source code of the underlying MPI implementation by extending the header sent along with a message in the communication substrate (Photon [11] uses this approach). Unfortunately, it is not portable to all MPI implementations and relies on a specially instrumented communication library. The second scheme sends an additional message containing the delay information for every data message. This scheme only requires changes to the portable MPI wrapper interposition library for the tool. While it is portable to all MPI implementations, it has a performance penalty associated with transmitting an additional message, a penalty not incurred by the first scheme. As a result, the overhead caused by the additional message would require further compensation.

The third scheme copies the contents of the original message and creates a new message with our own header that would include the delay information. This scheme has the portability advantage of the second scheme and avoids the second scheme's transmission of an additional message. However, copying contents of a message could prove to be an expensive operation, especially in the context of large messages that are transmitted in point-to- point communication operations.

We implemented a modification of the third scheme, but instead of building a new message and copying buffers in and out of messages (at the sender and the receiver), we create a new datatype. This new datatype is a structure with two members. The first member is a pointer to the original message buffer comprised of n elements of the datatype passed to the MPI call. The second member is a double precision number that contains the local delay value. Once created, the structure is committed as a new user-defined datatype and MPI is instructed to send or receive one element of the new datatype. Internally, MPI may transmit the new message by composing the message from the two members by using vector read and write calls instead of its scalar counterparts. This efficient transmission of the delay value is portable to all MPI implementations, sends only a single message, and avoids expensive copying of data buffers to construct and extract messages.

3.3 TAU Overhead Compensation Prototype

To test the validity of our parallel profile compensation models, we built the portable prototype within the TAU performance system [5]. We previously implemented local overhead compensation, and now included the parallel compensation support. TAU computes parallel profile data during execution for each instrumented event. At runtime, TAU maintains an event callstack for each thread of execution. This callstack has performance information for the currently executing event (e.g., a routine entry) and its ancestors. We compute the delay that a process sees locally by first adding the number of completed calls to half the number of entries along the thread's callstack. We assume that an *enter* profile call takes roughly the same time as an *exit* profile call, which is true is most cases. Once we know the total number of timer calls and the total overhead associated with calling the enter and exit methods (see [3] for details), their product gives the local timer overhead. We keep track of adjusted wait times in a process, as explained earlier and subtract it from the local overhead to compute the local delay. This delay value is then piggybacked with a message.

Mapping MPI Calls. The essence of our parallel overhead compensation scheme is that whenever two processes interact with each other, the receiver is made aware of the sender's delay value, or how much sooner the communication operation would have taken place in the absence of instrumentation. We have discussed above how this scheme operates for synchronous message communication operations using `MPI_Send` and `MPI_Recv`. In this section we explore how other MPI calls can be made aware of remote delays.

Asynchronous Operations. When storing or retrieving the piggyback value, we create an auto variable on the stack in our wrapper routines for `MPI_Send` or `MPI_Recv`. Synchronization operations involve loads or stores to this variable. The logic to process the piggyback value when it is received is incorporated in the `MPI_Recv` wrapper routine. Here, we compare the local and remote delays to arrive at how much adjustment needs to be made to the waiting time. Now let us examine the asynchronous `MPI_Isend` and `MPI_Irecv` calls. When the user issues the `MPI_Isend` call, we compute the local delay and create a global variable where this is stored. The location of this global piggyback variable in the heap memory is used when we create our struct for a new datatype for sending the message.

On the receiving side, a similar arrangement of the piggyback value is used. When the message is finally received, MPI automatically copies the contents of the piggyback value into the heap where this value is to be stored. We also create a map that links the address of the MPI request to the address of this piggyback value. The logic that compares the local and remote delays cannot be incorporated in the `MPI_Irecv` wrapp-per due to the very nature of the asynchronous operation (the values are not received when the routine executes). Hence, we do not adjust the time spent in `MPI_Irecv` as we did for `MPI_Recv`. Instead, an asynchronous message is visible to the program only after executing the `MPI_Wait`, `MPI_Test`, or variants of these calls (`Waitall`, `Waitsome`, `Testall`, `Testsome`) to wait for or test one or more requests. When a request is satisfied, we examine the map and retrieve the value of the piggyback variable where the remote process' overhead is stored. Then, a comparison of local and remote delays and an adjustment of waiting time is made on the receiving side. When more than one message is received by the process, we need to examine all the remote delays to determine how much time the process would have waited in the absence of instrumentation. We discuss this in more detail next with collective operations.

Collective Operations. Consider the class of collective operations supported by MPI. Let us first examine the `MPI_Gather` call where each process in a given communicator provides a single data item to MPI. The process designated with the rank of root gathers all the data in an array. It is important to communicate the local delays from each process to the root process. To do this, we form a message with the piggyback delay value and call a single `MPI_Gather` call. At the receiving end, we receive a single contiguous buffer where the application data and the delay values are put together in a single buffer. We extract the piggyback values out of this buffer and construct the application buffer with the rest. Once we get an array of the delay values from each process we compute the minimum delay value from the group of processes. Since the collective operation cannot complete without the message with the minimum delay, it must adjust its waiting time based on this value. So, the collective operation reduces to the case where the receiver gets a message from one process that has the least delay in the communicator. We can now apply the performance overhead compensation model as described in the previous section.

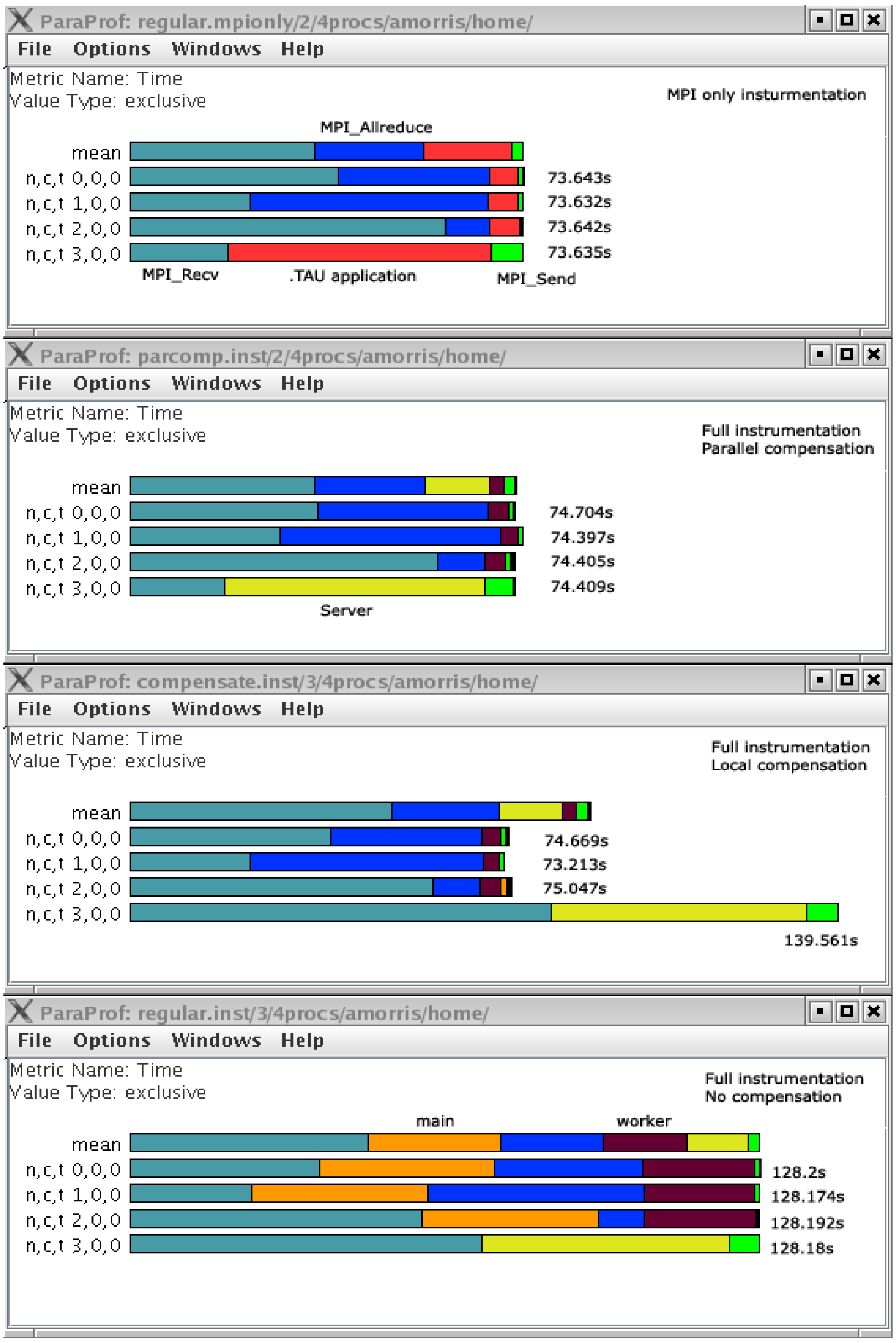
When broadcasting a message from one task to several, `MPI_Bcast` is modelled based on the two process overhead compensation model (see [4]). We create a new datatype, on the root process, that embeds the original message and the local delay value. This message is sent to all other members of the group. Each receiver compares the remote delay with its local delay and makes adjustments to the waiting time and local overhead, as if it had received a single message from the remote task. We use the model described earlier to do this.

To model `MPI_Scatter`, which distributes a distinct message to all members of the group, we create a new datatype that includes the overhead from the root process. This is similar to the `MPI_Gather` operation. After the operation is completed, each receiver examines the remote overhead and treats it as if it had received a single message from the root node, applying our previous scheme for compensating for perturbation.

`MPI_Barrier` requires all tasks to block until all processes invoke this routine. `MPI_Barrier` is implemented as a combination of two operations: `MPI_Gather` and `MPI_Bcast`, sending the local delay from each task to the root task (arbitrarily selected as the process with the least rank in the communicator). This task examines the local delay and compares it with the task with the least delay, adjusts its wait time and then sends the new local delay to all tasks using the MPI_Bcast operation. This mechanism preserves the efficiencies that the underlying MPI substrate may provide in implementing a collective operation. By mapping one MPI routine to another, we exploit those efficiencies.

4 Experimental Results

We validate our parallel performance intrusion compensation model using a prototype implemented within the TAU performance system. To illustrate the problem, we examine a parallel MPI application that computes the value of π using the Monte-carlo integration algorithm. The program calculates the area under the pi function curve from 0 to 1. The program comprises of a master (or server) task that generates work packets with a set of random numbers. The master task waits for a request from any worker and sends the chunk of randomly generated numbers to it. For each pair of numbers that is given to a particular worker, it finds out if the pair of cartesian co-ordinates represented by the numbers is below or above the pi function curve. Then, collectively, the workers estimate the value of pi iteratively until it is within a given error range. This simple example highlights how instrumentation overheads accumulated at the worker tasks are communicated to the master task. We execute the application in four modes: when there is no TAU instrumentation, with instrumentation without any compensation, with local perturbation compensation, and finally, with parallel perturbation compensation. As shown in table 1, these experiments are shown as distinct columns and we show


ParaProf: regular.mpionly/2/4procs/amorris/home/

Metric Name: Time
Value Type: exclusive

ParaProf: parcomp.inst/2/4procs/amorris/home/

Metric Name: Time
Value Type: exclusive

ParaProf: compensate.inst/3/4procs/amorris/home/

Metric Name: Time
Value Type: exclusive

ParaProf: regular.inst/3/4procs/amorris/home/

Metric Name: Time
Value Type: exclusive

Fig. 1. Parallel overhead compensation in TAU

Table 1. A comparison of parallel overhead compensation scheme in Monte-carlo integrator

Task	*No instrumentation*	*No compensation*	*Local compensation*	*Parallel compensation*
Master	73.926	128.179	139.56	73.926
Worker	73.834	128.173	73.212	73.909

the time spent in the worker and master tasks. We show the minimum times spent in the respective tasks. The timer overhead associated with a TAU timer was 480 nanoseconds on an Intel®Itanium2 Linux machine running at 1.5 GHz. The accuracy of compensation improves when we use high resolution timers, such as those provided by PAPI[1].

The results in Figure 1 and Table 1 show that local compensation schemes do manage to reduce the overhead in the worker tasks, but they fail in the master. The parallel compensation scheme reduces the overhead properly in both master and worker tasks.

5 Conclusion

Most parallel performance measurement tools ignore the overhead incurred by their use. Tool developers attempt to build the measurement system as efficiently as possible, but do not attempt to quantify the intrusion other than as a percentage slowdown in execution time. Our earlier work on overhead compensation in parallel profiling showed that the intrusion effects on the performance of events local to a process can be corrected [3] and also modeled how local overheads affected performance delay across the computation [4]. This paper implements those parallel models in the context of MPI message passing and demonstrates that parallel overhead compensation can be effective in practice to improve measurement error. The engineering feats to accomplish the implementation are novel. In particular, the approach to delay piggybacking can be generalized to other problems where additional information must be sent with messages.

It is important to understand that we are not saying that the performance profile we produce with overhead compensation represents the actual performance profile of an uninstrumented execution. The *performance uncertainty principle* [2] implies that the accuracy of performance data is inversely correlated with the degree of performance instrumentation. Our goal is to improve the tradeoff, that is, to improve the accuracy of the performance being measured during profiling. What we are saying in this paper is that the performance profiles produced with our models for performance overhead compensation will be more accurate than performance results produced without compensation.

Acknowledgements

This research is supported by the U.S. Department of Energy, Office of Science contract DE-FG02-05ER25680.

References

1. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, “A Portable Programming Interface for Performance Evaluation on Modern Processors,” *International Journal of High Performance Computing Applications*, **14**(3):189–204, Fall 2000.

2. A. Malony, "Performance Observability," Ph.D. thesis, University of Illinois, Urbana-Champaign, 1991.
3. A. Malony and S. Shende, "Overhead Compensation in Performance Profiling," *Euro-Par Conference*, LNCS 3149, Springer, pp. 119–132, 2004.
4. A. Malony and S. Shende, "Models for On-the-Fly Compensation of Measurement Overhead in Parallel Performance Profiling," (to appear) *Euro-Par Conference*, LNCS, Springer 2005.
5. A. Malony, S. Shende, "Performance Technology for Complex Parallel and Distributed Systems," In G. Kotsis, P. Kacsuk (eds.), *Distributed and Parallel Systems, From Instruction Parallelism to Cluster Computing, Third Workshop on Distributed and Parallel Systems (DAPSYS 2000)*, Kluwer, pp. 37–46, 2000.
6. A. Malony, et al., "Advances in the TAU Performance System," In V. Getov, M. Gerndt, A. Hoisie, A. Malony, B. Miller (eds.), *Performance Analysis and Grid Computing*, Kluwer, Norwell, MA, pp. 129–144, 2003.
7. A. Malony, D. Reed, and H. Wijshoff, "Performance Measurement Intrusion and Perturbation Analysis," *IEEE Transactions on Parallel and Distributed Systems*, **3**(4):433–450, July 1992.
8. A. Malony and D. Reed, "Models for Performance Perturbation Analysis," *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 1–12, May 1991.
9. A. Malony, "Event Based Performance Perturbation: A Case Study," *Principles and Practices of Parallel Programming (PPoPP)*, pp. 201–212, April 1991.
10. S. Sarukkai and A. Malony, "Perturbation Analysis of High-Level Instrumentation for SPMD Programs," *Principles and Practices of Parallel Programming (PPoPP)*, pp. 44–53, May 1993.
11. J. Vetter, "Dynamic Statistical Profiling of Communication Activity in Distributed Applications," *ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ACM, 2002.
12. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, 21(7), 558–565, July 1978.