# Using Software-Based Performance Counters to Expose Low-Level Open MPI Performance Information

David Eberius, Thananon Patinyasakdikul, George Bosilca
Innovative Computing Laboratory, University of Tennessee, Knoxville
1122 Volunteer Blvd
Knoxville, Tennessee 37996

## ABSTRACT

This paper details the implementation and usage of software-based performance counters to understand the performance of a particular implementation of the MPI standard, Open MPI. Such counters can expose intrinsic features of the software stack that are not available otherwise in a generic and portable way. The PMPI-interface is useful for instrumenting MPI applications at a user level, however it is insufficient for providing meaningful internal MPI performance details. While the *Peruse* interface provides more detailed information on state changes within Open MPI, it has not seen widespread adoption. We introduce a simple low-level approach that instruments the Open MPI code at key locations to provide fine-grained MPI performance metrics. We evaluate the overhead associated with adding these counters to Open MPI as well as their use in determining bottlenecks and areas for improvement both in user code and the MPI implementation itself.

## CCS CONCEPTS

• **General and reference** → **Metrics**; **Evaluation**; • **Networks** → **Network measurement**; • **Computing methodologies** → *Massively parallel and high-performance simulations*;

## KEYWORDS

MPI, Tools, Performance Counters, Profiling

**ACM Reference format:**
David Eberius, Thananon Patinyasakdikul, George Bosilca. 2017. Using Software-Based Performance Counters to Expose Low-Level Open MPI Performance Information. In *Proceedings of EuroMPI/USA '17, Chicago, IL, USA, September 25–28, 2017,* 8 pages.
https://doi.org/10.1145/3127024.3127039

## 1 INTRODUCTION

The standard paradigm for distributed memory parallelization is the Message Passing Interface (MPI) [3]. MPI has been used extensively for achieving high performance on distributed systems within academia and industry alike. Parallelizing an application is the first step toward a potential shorter time to solution, and it is usually supplemented by a performance analysis stage where performance bottlenecks (either in the algorithm itself or in the communication pattern) are identified and addressed. The MPI standard defines a profiling interface (PMPI) that allows tools to preempt all MPI function calls and add instrumentation or other functionality. Many tools like Vampir [2], Paraver [8], and TAU [12] use the PMPI interface to profile MPI applications, mainly through inserting timing functionality to track when MPI functions start and complete. This information is generally stored into a binary trace file, available to the tools post-mortem for thorough analysis. This method provides an overview of how the application progressed overall, but cannot expose low-level details and therefore provides little insight into what was happening within MPI. The MPI performance revealing extension interface (*Peruse*) [9] was developed as a means to complement the lack of fine grained details in the PMPI interface, and to provide more insight into MPI implementation performance. As an example, using the Peruse interface a tool could have exposed detailed MPI state change information such as when a send request enters the queue of posted messages or when a communication request is completed. This interface had great potential for performing in-depth analysis of the state changes experienced by each communication, however it does not provide information about what is happening within those states.

In this paper we seek to provide low-level MPI performance metrics to help identify the root cause of performance bottlenecks as well as a simple overview of MPI application properties. Our metrics are modeled after the hardware performance counters that are exposed through PAPI [11]. Unlike PAPI, however, our counters are based entirely in software.

This paper introduces an implementation of *Software-Based Performance Counters* (SPCs) within the Open MPI [4] implementation. We will start off our discussion with Section 2 in which we introduce related work, then we will move on to Section 3 where we will elaborate on our motivations behind implementing SPCs. In Section 4 we explain our design and implementation of SPCs within Open MPI and addresses the extent of the details we decided to expose through our counters. Section 5 shows the overhead associated with adding our counters to Open MPI, and Section 6 shows examples of our counters being used within both synthetic and real world applications. The final section provides a conclusion to our work and illuminates prospects for future work.

## 2 RELATED WORK

There are a variety of different performance analysis tools available to parallel application developers that offer different views of application performance. Each of these tools has its own strategy


*EuroMPI/USA '17, September 25–28, 2017, Chicago, IL, USA*

ACM ISBN 978-1-4503-4849-2/17/09…$15.00
https://doi.org/10.1145/3127024.3127039

for meeting the challenge of scalable performance analysis. Performance analysis tools in a High Performance Computing (HPC) environment tend to either collect data during runtime or perform postmortem analysis. Within data collection tools, there are two approaches to data collection: instrumentation-based and interrupt-based. Our SPC approach uses manual code instrumentation to collect performance information during runtime.

Before analyzing the performance of an application, one must first collect detailed profiling data for analysis. The performance application programming interface (*PAPI*) provides an interface for accessing hardware performance counters available from many modern microprocessors. *PAPI* is highly portable with support for most hardware vendors and operating systems, and provides a wealth of information with its counters such as the number of cache misses and the total number of instructions issued. The Tuning and Analysis Utilities (*TAU*) tool is another option for code instrumentation, and has support for both compiler-based and manual instrumentation of applications [12]. *TAU* keeps track of the information gathered and can generate binary trace files supported by visualization tools like *Vampir* and *Paraver*. There are many other HPC profiling tools, and often there are redundant measurement functionalities that these tools provide. The *Score-P* tool was created to provide a common infrastructure for performing these redundant performance measurement capabilities, and provides support for several other tools, including *Scalasca* and *TAU* [10].

Once data has been collected, tools are needed for analyzing and visualizing the data. Trace files are a simple way to represent what happened during a parallel execution, often including a series of time stamps for when certain events started and stopped along with which processing unit they executed on. The *Paraver* and *Vampir* tools are designed to parse such trace files and display their data in a meaningful layout showing what was happening on each computational unit at any given time. The *Vampir* toolset also provides a number of other visualization tools and works closely with *Score-P* for producing data. In addition to providing instrumentation capability, *TAU* also provides a profile visualization tool called *paraprof* that allows for detailed analysis of data gathered from TAU-enabled applications. The *Scalasca* performance toolset architecture supports measurement and analysis of several parallel programming paradigms in C/C++ and Fortran such as MPI and OpenMP [5]. *Scalasca's* postmortem analysis automatically identifies potential bottlenecks and critical regions within the code.

When it comes to profiling MPI applications, the PMPI interface has been one of the major methods of doing so since it is a part of the MPI standard. The *Peruse* tool was proposed as an addition to the MPI standard, and exposes detailed internal MPI state information through instrumentation using function callbacks [9]. *Peruse* exposes similar information to our SPCs, though it does so from a different perspective. Some of our counters, such as matching time, could be determined through postmortem analysis of a *Peruse* trace. Our SPCs offer some information that is not exposed through *Peruse* such as the number of out of sequence messages and fine-grained bytes sent/received information. The *Peruse* interface was not accepted into the MPI standard, but as of the MPI 3.1 standard, a new MPI tool information interface (*MPI_T*) was added [3].

The *MPI_T* interface is divided into two parts. The first part provides information and editing capability for control variables used to tune the MPI implementation's configuration [3]. The second part provides users access to internal performance variables in the MPI implementation. Our SPCs are similar to these internal performance variables and could be integrated into the *MPI_T* standard as performance variables.

## 3 MOTIVATION

When profiling MPI applications, it can often be difficult to tell what is causing performance issues, particularly when the problem lies within MPI itself. There are many factors that can affect the performance of an MPI implementation, such as handling unexpected or out-of-sequence messages. An unexpected message is one that arrived before the corresponding receive was posted. It is well known that searching the unexpected message queue can quickly become a bottleneck, particularly when there are a large number of messages in the queue [14]. Out-of-sequence messages are messages that were delivered out of the MPI-imposed order (MPI is expected to deliver the messages in FIFO order between each pair of processes within a communicator), due to multiple network paths between the processes. These messages block the matching queue on the target process, as all matching must be delayed until the FIFO order can be guaranteed. We wanted to have a tool that would be able to report such internal MPI information to the user/tool, to provide a more precise picture of what particular conditions could have affected the application performance.

Such a tool has the potential to be generic enough to be of use not only to users, but also be particularly useful to those who are developing an MPI implementation. Having metrics on the internal MPI behavior can help identify bugs and inefficiencies in the implementation, and correct performance critical bottlenecks before they impact production-level scientific applications. One active area of MPI development is in implementing efficient multi-threaded MPI communication, which requires extra care to enforce thread safety and ensure that messages are received in the order in which they were sent[1]. Being able to easily access internal metrics like when data transfers are initiated or when a message arrives out of sequence can help decrease the burden on multi-threaded MPI developers by giving an explanation for the performance they are seeing.

Another crucial benefit to having these metrics internal to the MPI implementation is that they can be exposed without using the PMPI interface. The PMPI interface works by preempting MPI functions, while our SPCs work through instrumentation of Open MPI code and don't need to interfere with this preemption. Many existing MPI tools use the PMPI interface to perform their profiling, so keeping this interface free allows those tools to be used concurrently with our SPCs. Users can also leverage the PMPI interface for supplementing MPI functions with their own code.

We modeled our counters after *PAPI*'s hardware counters due to their simplicity and familiarity within the HPC community. We wanted to have a similar system for exposing low level information as *PAPI*, but for software-based events rather than hardware events.

## 4 DESIGN AND IMPLEMENTATION

In order to allow for a wide range of different functionalities, Open MPI utilizes the *modular component architecture* (MCA) to determine which components and options are used both at compile-time and runtime [15]. The standard stack of components used for communication starts with MPI at the top level; the Point-to-Point management layer (PML) below that; the BML management layer (BML) below that; and the byte transfer layer (BTL) at the lowest level. When a user calls an MPI communication function, Open MPI transfers control to the PML. The PML uses the BML to determine the appropriate BTL implementation for a particular transfer, and then the BTL handles the hardware transfers of data between MPI processes.

Our implementation of SPCs enables users to see useful performance metrics that range from the number of times MPI_Send was called to more detailed internal MPI metrics such as the number of messages that were unexpected. The driver code for our SPCs was implemented in the Open Portable Access Layer (OPAL) within Open MPI which acts as a utility and *glue* layer operating at a low level in the Open MPI stack. The driver code consists of data structures for storing the counter information and functions for managing allocated memory and updating the counters. The counter values are stored as **long long** integer values, so at this time floating point counters are not supported. The instrumentation code for the various counters appears in both the MPI and PML layers, depending on how low level the information is. Some counters are only updated in one location, while others are updated in multiple places to most accurately reflect the metric they represent. Each MPI process has its own SPC data structures, so the counters are updated separately for each process. Since MPI allows for multi-threading within a process, all updates to the SPC data structures are performed using atomic operations.

The counter update functionality is implemented as a macro that gets optimized out if Open MPI is built without SPCs. We also took steps to minimize the impact that instrumentation has on the performance of Open MPI with particular attention to the critical fast path. The actual update consists of an **if** statement that checks whether that particular counter is activated, and an atomic add operation to update the counter. At this time, counters can only be modified through addition. All of the currently implemented counters monotonically increase over time, but there is support for decreasing the counters through adding negative numbers to them. Some counters require timing information, so they utilize Open MPI's low-level timing utilities.

### 4.1 Performance Metrics Exposed

We decided to implement a variety of counters that expose information from two different levels within the Open MPI stack. The first level we added counters to is the MPI layer. These counters can be seen in Table 1. In this layer, the counter value simply denotes how many times each of the user-level communication functions has been called. This information is useful for showing an overview of the types of communications that appear in an MPI application.

In order to expose more detailed information, we added several counters to the PML level as well. The counters for this level are shown in Table 1. We decided to split the counters for bytes received and bytes sent into two subcategories: bytes sent/received by the user and bytes sent/received by MPI. The distinction here is that bytes sent/received by the user are from point to point messaging functions like MPI_Send and MPI_Recv, and bytes sent/received by MPI contains data sent by the MPI library. This includes additional data transmitted for the process of data transfer management, data transmitted for all MPI-internals such as particular implementations of collective messaging functions like MPI_Bcast, topology information detection and exchange, and particular algorithms for communicators, windows, and file creation. These counters are updated at message fragment granularity in that as soon as a fragment is given to or taken from the BTL level, the counters are updated. The aforementioned methodology works well for smaller messages, but the process becomes more complicated with larger messages. Open MPI uses remote direct memory access (RDMA) operations such as *Put* and *Get* operations for large memory transfers. These Put and Get operations are handled by the BTL, so rather than add detailed counters to all of the BTL implementations, we decided to simply add Put and Get counters at the PML level and update them when a Put or Get operation is initiated. It may be some time between initiation and when the data is actually transferred, so these counters are much more coarse-grained in their updates than the bytes sent/received counters.

**Table 1: The names of our counters.**

| MPI Level | PML Level |
|---|---|
| OMPI_SEND | OMPI_BYTES_RECEIVED_USER |
| OMPI_RECV | OMPI_BYTES_RECEIVED_MPI |
| OMPI_ISEND | OMPI_BYTES_SENT_USER |
| OMPI_IRECV | OMPI_BYTES_SENT_MPI |
| OMPI_BCAST | OMPI_BYTES_PUT |
| OMPI_REDUCE | OMPI_BYTES_GET |
| OMPI_ALLREDUCE | OMPI_UNEXPECTED |
| OMPI_SCATTER | OMPI_OUT_OF_SEQUENCE |
| OMPI_GATHER | OMPI_MATCH_TIME |
| OMPI_ALLTOALL | OMPI_OOS_MATCH_TIME |
| OMPI_ALLGATHER | |

Typically, MPI implementations handle unexpected messages by pushing them into a queue that will be checked each time a receive is posted. We decided to provide a counter that keeps track of how many unexpected messages there were in an execution to give users an idea of how often this is happening.

The MPI standard specifies that messages must be received in the order that they are sent [3]. Open MPI enforces this ordering using a *sequence number* for each message between two MPI processes in the same communicator. In Open MPI, order is sometimes enforced by the BTL implementation such as with the TCP BTL, but other BTL implementations such as *openib* for InfiniBand do not necessarily enforce ordering in every use case. The InfiniBand hardware does enforce ordering of the messages, however the *openib* BTL implementation in Open MPI allows for messages to be sent out of sequence when there are messages that failed to send. Essentially, when a message fails to send it is put into a queue for resending later, but the *openib* BTL allows for messages in the fast path to bypass

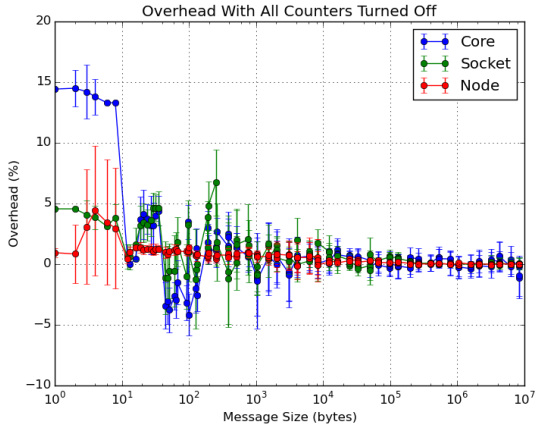


**Figure 1: The overhead of adding SPCs to the code while leaving all of them turned off. Note: the error bars represent the standard deviation across the 10 test runs.**

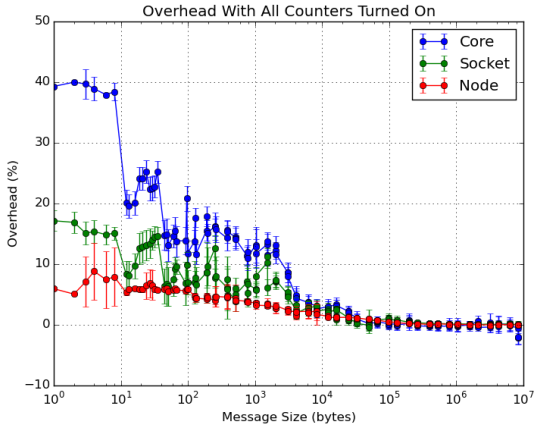


**Figure 2: The overhead of adding SPCs to the code and turning all of them on. Note: the error bars represent the standard deviation across the 10 test runs.**

checking the failed message queue, thus these fast path messages are sent before the failed messages. At the receiver side, posted receives can only be matched once all prior sequence numbered requests have been received. Out of sequence messages can cause a significant bottleneck due to increased memory management and time spent searching through the queue of out of sequence messages. To identify this bottleneck, we increment a counter every time a message is delivered out of sequence.

For both unexpected and out of sequence messages, Open MPI needs to perform a matching process to pair a receive request with its corresponding arrived data. The time it takes to perform this matching process can have a big impact on latency, particularly when there are a large number of messages in the queues. We used a separate counter for matching time of out of sequence messages because they can take particularly long to match and have much more internal overhead associated with them. The number for these matching time counters represents the number of microseconds spent matching messages.

## 5 OVERHEAD

It is critical to ensure that the overhead imposed by any performance gathering mechanism remains minimal, and its impact on the performance of the underlying MPI functionality is unaffected. To test how much overhead is introduced with these counters, we use the NetPIPE benchmark [13]. This benchmark performs a ping-pong throughput test and reports the bandwidth and latency for a variety of message sizes and repetitions for each message size. In this section we focus on the latency numbers from NetPIPE because they provide more insight into the overhead of the different counters.

We tested our implementation on the *Arc* machine, the configuration for which can be found in Table 2. To test different usage cases, we performed the NetPIPE benchmark with three different configurations with varying degrees of expected impact. The first configuration, *Node*, focuses on inter-node communication over

**Table 2: Configuration of the testing system, *Arc*.**

| Property | Arc Configuration |
|---|---|
| Processor | Dual 10-core Intel Xeon E5-2650 v3 @2.3 Ghz |
| Interconnect | InfiniBand EDR (100 Gb) |
| Compiler | gcc 6.3.0 |
| Open MPI | optimized, dynamic build |

InfiniBand using the *openib* BTL. Here, *node* refers to an individual server within a distributed compute cluster. The next two configurations, *Socket* and *Core*, deal with intra-node communication over shared memory using *SysV* shared memory through the Open MPI *vader* BTL. With the *Socket* test, the MPI processes were bound to cores from different CPU sockets within the same node, and for the *Core* test, the MPI processes were bound to cores within the same socket. Here *socket* refers to a CPU slot within a compute server.

For the baseline test, we built Open MPI with the same set of configuration parameters but without our counters enabled which turns all of the code associated with SPCs into no-ops. We then performed several tests with the SPCs compiled in. The first two tests simply have all of the counters turned off or all of the counters turned on. The overhead of all counters being off shows the impact of the **if** statements added to the different paths in the code (including in some cases to the critical path). Having all of the counters turned on is the worst case for overhead, and the impact will be from both the **if** statements and instructions added to handle the counters (including in most cases atomic add operations). All of our overhead data points are the average of ten runs of NetPIPE to help account for noise in the network. We present also the standard deviation of the non-curated data points, to highlight the worst and best case scenario.

Figure 1 shows the overhead incurred when SPCs are built, but all of them are turned off. This effectively shows the difference in performance if SPCs were to be included in the Open MPI build

**Table 3: Results of the pairwise benchmark with 2, 4, and 8 threads. A window size of 256, message size of 64 bytes, and iteration count of 100. Note: The message rate and wall time do not include the warm-up phases, but the other values do. Without warm-up messages there are $256 \times 100 \times N_t$ messages sent where $N_t$ is the number of threads.**

| Threads | Message Rate (msg/sec) | Receives | OOS Messages | % OOS | Wall Time (us) | Match Time (us) | OOS Match Time (us) |
|---|---|---|---|---|---|---|---|
| 2 | 601,773.54 | 56,320 | 16,633 | 29.53% | 85,598 | 9,634 | 9,875 |
| 4 | 476,174.73 | 112,640 | 47,216 | 41.92% | 218,807 | 34,312 | 51,196 |
| 8 | 162,458.93 | 225,280 | 112,813 | 50.08% | 1,260,863 | 96,465 | 729,187 |

**Table 4: Counters used in the NetPIPE benchmark.**

| Counter Name |
|---|
| OMPI_SEND |
| OMPI_RECV |
| OMPI_BYTES_RECEIVED_USER |
| OMPI_BYTES_SENT_USER |
| OMPI_BYTES_GET |
| OMPI_UNEXPECTED |
| OMPI_MATCH_TIME |

by default. The overall trend is that the overhead decreases as the message size increases. As we expected, the inter-node overhead is the lowest with the overhead for most message sizes being around 1%. For messages between 3 and 8 bytes in length, we saw an increase in latency on the *Arc* system. This spike in latency happened infrequently, but was more likely to occur when the counters were turned on resulting in over 4% overhead on average. The maximum overhead for this test was ~14.5% for small messages sent between cores in the same socket. In most cases the overhead was less than 5%, and for the intra-node tests the latency was actually shorter on average for message sizes around 100 bytes when the counters were built. We have not yet determined why the latency improves on average for these messages when the counters are compiled.

In Figure 2, we see the maximum overhead of using SPCs with NetPIPE since all of the counters are turned on. We see similar patterns in the plots for the different test cases, simply with higher magnitudes. Again, the *Core* test shows the highest overhead with ~40.0%. The inter-node overhead remains around or below 5% overhead for most message sizes. This result shows that for the most common case, adding SPCs does not add a large amount of overhead.

To account for the sizable gap between the overheads of having all counters turned on and off, we decided to perform tests with selected counters turned on. For the NetPIPE benchmark, seven different SPCs are encountered during the run. These counters are shown in Table 4. After testing with different counters turned on, we found that the OMPI_MATCH_TIME counter accounts for the majority of the overhead.

Figure 5 provides a comparison between having the counters turned on, turned off, only having OMPI_MATCH_TIME turned on, and only having the six counters needed by NetPIPE minus OMPI_MATCH_TIME, for the *Core* test. This figure shows that nearly all of the overhead increase from all off to all on can be attributed to the OMPI_MATCH_TIME counter. In order to update this counter, we use a timer function to get the start and end times of the matching process. Both starting and stopping the timer require *if* statements to ensure the OMPI_MATCH_TIME counter is turned on in addition to the *if* statement and atomic add operation of the counter update. In total, there are three *if* statements, two timer function calls, one subtraction operation (for calculating elapsed time), and one atomic add operation needed for each match. The other counters used in NetPIPE, by comparison, only require a single *if* statement and an atomic add. The OMPI_MATCH_TIME counter can also happen more often than many other counters because the matching process can happen multiple times for a single message if it is unsuccessful. The OMPI_BYTES_RECEIVED/SENT_USER counters can also happen multiple times per message if the message is broken into fragments, yet these counters do not add a significant amount of additional overhead. This suggests that the additional *if* statements and timer function calls are the cause of this increased overhead.

The timer function used for these test cases simply calculates the time in microseconds by dividing the monotonic number of cycles returned from the RDTSC instruction by the clock frequency in MHz. This division operation can add a large percentage of time when the latency is already low. For example, in the case where the overhead is 40%, the latency without the SPCs built is ~200 nanoseconds and the latency with the SPCs built and turned on is ~280 nanoseconds. This additional overhead of 80 nanoseconds equates to 184 cycles on the *Arc* machine. On this machine, the estimated length of a 32-bit division operation is 35-47 cycles according to the Intel manual, and we need two of them for each time the matching process happens so for each match these division operations add 70-94 cycles of overhead [7]. To verify that removing the OMPI_MATCH_TIME counter reduces the overhead for all cases, we decided to redo the *Core*, *Socket*, and *Node* tests with this counter turned off. Figure 4 shows the overhead of the three tests if the OMPI_MATCH_TIME counter is turned off. For all of the use cases, the performance is nearly the same as having none of the counters turned on.

## 5.1 mpiP Overhead

To compare with a similar tool, we look at the overhead of using the *mpiP* tool which uses the PMPI interface to instrument the code. *mpiP* adds timing information and counters to user-level MPI functions like MPI_Send and MPI_Recv and associates each function call to its calling location in the user code. The idea is to assess the number of times and length of time spent in each function from each location. This tool is similar to our SPCs in that

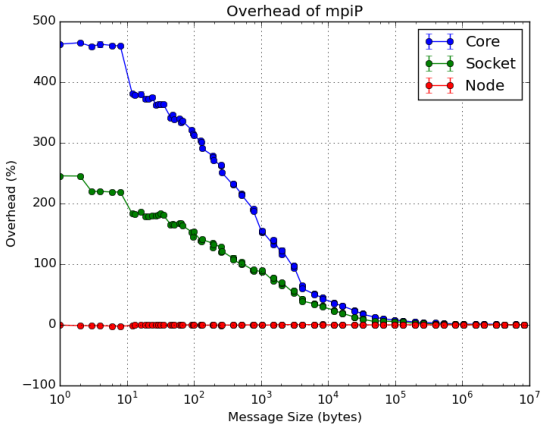


**Figure 3: The overhead of using mpiP with NetPIPE. Note: the error bars represent the standard deviation across the 10 runs, however the deviation was extremely small so they appear nonexistent.**

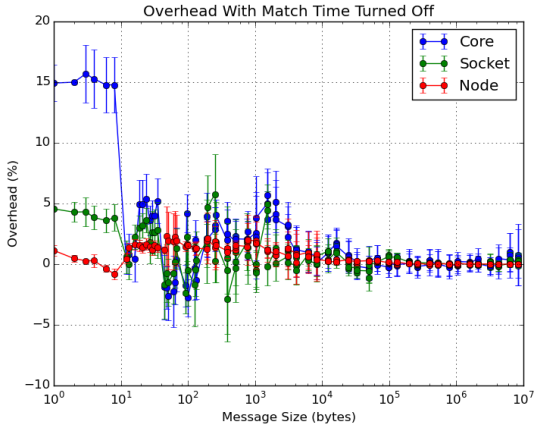


**Figure 4: The overhead of adding SPCs to the code and turning on only the counters from Table 4 minus OMPI_MATCH_TIME. Note: the error bars represent the standard deviation across the 10 runs.**

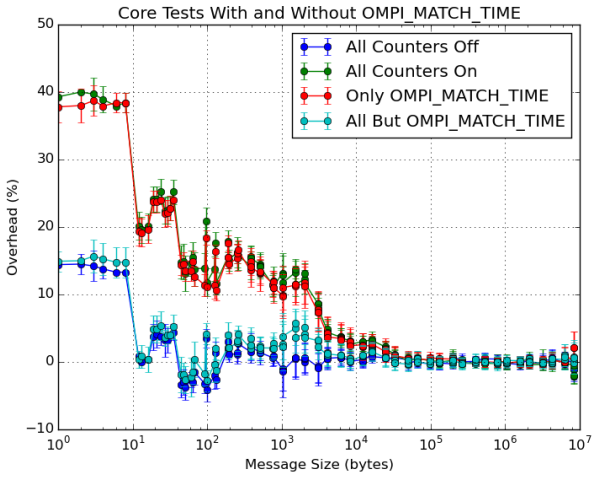


**Figure 5: Comparing the intra-node overhead within a single socket with the counters all on, all off, only OMPI_MATCH_TIME turned on, or only the counters from Table 4 minus OMPI_MATCH_TIME. Note: the error bars represent the standard deviation across the 10 runs.**

it keeps track of the number of times that MPI functions are called, however it provides additional information in the form of timing information for the function calls. Figure 3 shows the overhead of running NetPIPE with *mpiP*, using the default settings for *mpiP*. For the *Node* test, *mpiP* performed similarly to our baseline Open MPI test, but the shared memory tests tell a much different story. For the *Socket* test, the overhead of using *mpiP* was up to ~245.6% and for the *Core* test, the overhead was up to ~465%.

Our *mpiP* build used PMPI_Wtime for its timing measurements, which redirects to the same low level timer that we used for our OMPI_MATCH_TIME counter. Some of the additional overhead comes from determining the call sites of MPI functions from the call stack and from calling *mpiP* functions. This overhead is particularly apparent for the intra-node tests because the latency for these tests is already as low as hundreds of nanoseconds and simply adding timer functions can have a large impact as seen with the OMPI_MATCH_TIME counter. For the inter-node test, this overhead is largely hidden by the network latency since the baseline latency in these tests are in microseconds.

## 6 APPLICATION PERFORMANCE MEASUREMENT

In this section, we will be applying our using our SPCs to analyze the performance of two sample applications. The first application, *pairwise*, is a synthetic benchmark to measure message injection rate in a multi-threaded MPI environment. Our second application is a quantum chemistry application called *moldft* implemented with the Multiresolution, Adaptive Numerical Environment for Scientific Simulation (MADNESS) [6]. This application also runs in a multi-threaded MPI environment. Our analysis will focus on the effect of out of sequence (OOS) messages on MPI application performance.

### 6.1 Pairwise Benchmark

The *pairwise* benchmark runs with two MPI processes and creates $T$ threads per process and connects these threads pairwise within a single communicator. This means that the first thread from the first process is paired with the first thread from the second process and so on. Only paired threads communicate with each other. The benchmark performs a warm-up phase, and then calculates injection rate by posting a window of $W$ asynchronous sends of size $S$ from process 0 to process 1 for each thread and repeats this $I$ times. We decided to use a relatively small message size of 64 bytes, a window size of 256 messages, and an iteration count of 100 for our tests.

We also used the *vader* BTL for shared memory communication and placed the threads from each process in different sockets.

The results of the *pairwise* tests are shown in Table 3. These results show a clear trend that increasing the number of threads decreases the message rate, which is the opposite of what one would expect. The SPC values give us some insight into what is causing this decreasing message rate. We can see that the number of OOS messages is a large percentage of the total number of messages received. For 2 threads, the percentage of OOS messages is 29.53%, and when the thread count gets up to 8, over 50% of the messages were received out of sequence. As the number of OOS messages increase, so does the time spent matching OOS messages. This OOS matching time has become a huge bottleneck for injection rate with the time spent matching reaching over 50% of the total wall time.

We know that OOS messages reduce the performance, but why are these messages arriving out of sequence in the first place? The answer lies in the fact the *pairwise* benchmark was configured to perform all the sends and receives between threads within a single communicator. With all of the threads in one communicator, they compete for acquiring sequence numbers and memory locations to perform the data transfers since both of these operations are atomic. When multiple threads attempt these operations at the same time, the order in which they get what they need is nondeterministic. The OS can also deschedule the threads at any time which can influence the order in which they transfer the data. Effectively, when multiple threads attempt to send data at the same time, their sequence and the order in which they actually acquire a memory location to write to are not necessarily in the same order. The more threads there are, the more likely it is for these collisions to occur, which then increases the number of OOS messages.

## 6.2 MADNESS Benchmark

MADNESS comes packaged with several test benchmarks, one of which is called *moldft*. This benchmark takes a set of molecular geometry as input and performs a molecular dynamics simulation based on density-function theory. MADNESS operates in a multi-threaded MPI environment in which any thread can communicate with any other thread directly. As we have seen with the *pairwise* benchmark, using multi-threaded MPI can potentially cause a large number of OOS messages when the thread counts are high.

To test the impact of OOS messages on simulation performance, we decided to use three different BTLs to run the simulation: *vader*, *openib*, and *TCP* over InfiniBand. The *vader* shared memory BTL and the *openib* InfiniBand BTL can both potentially allow messages to be sent out of sequence, while the *TCP* BTL does not allow OOS messages. To provide a fair comparison between the different BTLs, we held the number of threads constant at 18 with 9 on each node for *openib* and *TCP*. We decided to use a moderate sized problem within *moldft* that performs the simulation on five water molecules.

Table 5 shows the results of the MADNESS experiments. Under normal circumstances, one would expect that using InfiniBand would outperform TCP, but this is not the case for these tests. The TCP over InfiniBand test ran ~20.8% faster than the pure InfiniBand test on average. As we expected, OOS messages have a huge effect on performance here with ~37% of the messages in the *openib* tests being delivered out of sequence. With this many OOS messages, the time spent managing the OOS data structures and matching messages also increases. The shared memory test ran ~17.6% faster than TCP, and ~42% faster than *openib* and had much fewer out of sequence messages than *openib* with only ~4% of the messages arriving out out sequence.

**Table 5: The results of the MADNESS *moldft* tests using five water molecules. The counter values are the average of 10 runs with 18 threads per run of the simulation for each configuration. Note: the total time is the wall time reported by *moldft*.**

| Counter | *openib* | *vader* | *tcp* |
|---|---|---|---|
| **Total Time (sec)** | 626.41 | 440.95 | 518.54 |
| OMPI_RECV | 12,995.6 | 13,024.7 | 12,710.6 |
| OMPI_ISEND | 3,252,957.0 | 3,253,238.0 | 3,143,029.9 |
| OMPI_IRECV | 3,291,284.3 | 3,291,596.8 | 3,180,212.8 |
| OMPI_BCAST | 4.0 | 4.0 | 4.0 |
| OMPI_BYTES_RECEIVED_USER | 1,898,491,237.9 | 879,724,185.9 | 23,428,171,947.7 |
| OMPI_BYTES_RECEIVED_MPI | 168.0 | 168.0 | 168.0 |
| OMPI_BYTES_SENT_USER | 1,980,636,856.5 | 968,525,668.5 | 23,675,080,020.9 |
| OMPI_BYTES_SENT_MPI | 168.0 | 168.0 | 280.0 |
| OMPI_BYTES_PUT | 0.0 | 0.0 | 129,295,502.3 |
| OMPI_BYTES_GET | 23,032,934,218.0 | 24,056,241,711.4 | 0.0 |
| OMPI_UNEXPECTED | 126,339.4 | 21,654.5 | 14,868.5 |
| **OMPI_OUT_OF_SEQUENCE** | 1,222,397.6 | 134,631.0 | 0.0 |
| OMPI_MATCH_TIME | 282,910.2 | 369,343.7 | 251,844.7 |
| OMPI_OOS_MATCH_TIME | 317,157.8 | 32,742.9 | 0.0 |

## 7 CONCLUSIONS

In this paper we have described our implementation of software-based performance counters into the Open MPI library. Incorporating SPCs into Open MPI was a simple process, though the placement of the instrumentation required knowledge of Open MPI's internal code. Adding SPCs into another MPI implementation would not take much effort for a developer familiar with the implementation.

Using these simple performance counters can provide insight into an MPI program's performance characteristics, particularly in a multi-threaded approach. We have introduced several detailed internal counters that provide access to information such as fine-grained bytes sent/received information for non-RDMA transfers, the number of out of sequence messages, and matching time which can provide metrics for how well the MPI implementation is performing and provide context to an MPI application's performance.

With the exception of the matching time counters, using our SPCs introduces less than 5% latency overhead when communicating between server nodes and around 15% for small messages up to 8 bytes in size for shared memory transfers within one CPU socket. When the matching time counters are added, the latency overhead is significantly higher with up to 40% latency overhead in the worst case. It should however be noted that these numbers quickly drop to below 1% as the message size increases, leading to no impact on most types of applications (with the exceptions of applications driven by injection rate of small messages).

In the future, we would like to add additional counters to keep track of more MPI metrics at various levels of the Open MPI stack such as the time spent waiting in MPI_Wait or MPI_Barrier. Of potential interest would also be to allow for counters to be communicator-specific, through incorporating our counters into the MPI Tools Information Interface (*MPI_T*). Merging our counters into the *MPI_T*

interface would add a more robust system for turning counters on and off as well as providing information on each counter to the user. Additionally, we would like to provide an interface for users to be able to easily take *snapshots* of the counter values and use this information as they see fit, possibly by showing the rate of change of a particular counter or calculating instantaneous bandwidth. We are actively working on having our SPCs added to the Open MPI code base as an MCA parameter.

## ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by NSF collaborative award (#1339820).

## REFERENCES

[1] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. 2008. Toward Efficient Support for Multithreaded MPI Communication. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, Berlin, Heidelberg, 120–129. https://doi.org/10.1007/978-3-540-87475-1_20

[2] Holger Brunst, Manuela Winkler, Wolfgang E. Nagel, and Hans-Christian Hoppe. 2001. *Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach*. Springer Berlin Heidelberg, Berlin, Heidelberg, 751–760. https://doi.org/10.1007/3-540-45718-6_80

[3] Message Passing Interface Forum. 2015. *MPI: A Message-Passing Interface Standard Version 3.1*. http://mpi-forum.org/.

[4] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 97–104. https://doi.org/10.1007/978-3-540-30218-6_19

[5] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca Performance Toolset Architecture. *Concurr. Comput. : Pract. Exper.* 22, 6 (April 2010), 702–719. https://doi.org/10.1002/cpe.v22:6

[6] Robert J. Harrison, Gregory Beylkin, Florian A. Bischoff, Justus A. Calvin, George I. Fann, Jacob Fosso-Tande, Diego Galindo, Jeff R. Hammond, Rebecca Hartman-Baker, Judith C. Hill, Jun Jia, Jakob S. Kottmann, M-J. Yvonne Ou, Junchen Pei, Laura E. Ratcliff, Matthew G. Reuter, Adam C. Richie-Halford, Nichols A. Romero, Hideo Sekino, William A. Shelton, Bryan E. Sundahl, W. Scott Thornton, Edward F. Valeev, ÃĄlvaro VÃązquez-Mayagoitia, Nicholas Vence, Takeshi Yanai, and Yukina Yokoi. 2016. MADNESS: A Multiresolution, Adaptive Numerical Environment for Scientific Simulation. *SIAM Journal on Scientific Computing* 38, 5 (2016), S123–S142. https://doi.org/10.1137/15M1026171 arXiv:http://dx.doi.org/10.1137/15M1026171

[7] Intel 2016. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel.

[8] Gabriele Jost, Haoqiang Jin, Jesus Labarta, Judit Gimenez, and Jordi Caubet. 2003. Performance Analysis of Multilevel Parallel Applications on Shared Memory Architectures. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*. IEEE Computer Society, Washington, DC, USA, 80.2–. http://dl.acm.org/citation.cfm?id=838237.838714

[9] Rainer Keller, George Bosilca, Graham Fagg, Michael Resch, and Jack J. Dongarra. 2006. Implementation and Usage of the PERUSE-Interface in Open MPI. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI'06)*. Springer-Verlag, Berlin, Heidelberg, 347–355. https://doi.org/10.1007/11846802_48

[10] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir*. Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91. https://doi.org/10.1007/978-3-642-31476-6_7

[11] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A Portable Interface to Hardware Performance Counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*. 7–10.

[12] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (May 2006), 287–311. https://doi.org/10.1177/1094342006064482

[13] Quinn O. Snell, Armin R. Mikler, John L. Gustafson, and The Pennsylvania State University CiteSeer Archives. 1996. NetPIPE: A Network Protocol Independent Performance Evaluator. (1996). http://citeseer.ist.psu.edu/343003.html

[14] K. D. Underwood and R. Brightwell. 2004. The impact of MPI queue usage on message latency. In *International Conference on Parallel Processing, 2004. ICPP 2004.* 152–160 vol.1. https://doi.org/10.1109/ICPP.2004.1327915

[15] T. S. Woodall, R. L. Graham, R. H. Castain, D. J. Daniel, M. W. Sukalski, G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. 2004. *Open MPI's TEG Point-to-Point Communications Methodology: Comparison to Existing Implementations*. Springer Berlin Heidelberg, Berlin, Heidelberg, 105–111. https://doi.org/10.1007/978-3-540-30218-6_20