Characterization of Computation-Communication Overlap in Message-Passing Systems

Abstract

Effective overlap of computation and communication is a well understood technique for latency hiding and can yield significant performance gains for applications on high-end computers. In this report, we describe an instrumentation framework developed for message-passing systems to characterize the degree of overlap of communication with computation in the execution of parallel applications. The inability to obtain precise time-stamps for pertinent communication events is a significant problem, and is addressed by generation of minimum and maximum bounds on achieved overlap. The overlap measures can aid application developers and system designers in investigating scalability issues. The approach has been used to instrument two MPI implementations as well as the ARMCI system. The implementation resides entirely within the communication library and thus integrates well with existing approaches that operate outside the library. The utility of the framework is demonstrated by analyzing communication-computation overlap for micro-benchmarks and the NAS benchmarks, and the insights obtained are used to modify the NAS SP benchmark, resulting in improved overlap.

1 Introduction

Overlapping computation with communication, or latency hiding, has long been recognized as an effective technique for masking data transfer latency, with the potential for considerable performance gains, enabling applications to scale well on large numbers of processors. Modern user-level networking architectures [1] like InfiniBand [11], Quadrics [26] and Myrinet [2] support OS-bypass communication that reduces the involvement of the host CPU in the actual data transfer path and frees the CPU to do user computation instead. Network interface cards equipped with DMA engines are able to move data between end points without host processor participation. If communication can proceed concurrently while the processor is busy computing, then the system can achieve useful overlap of computation and communication. User-level protocols thus provide an opportunity for overlapping the movement of data with execution of application code. Because improvements in communication performance have not matched improvements in processor speed, latency hiding has become an increasingly important technique for improving application performance.

The degree of actual overlap for an application depends on the overlap potential of both the application and the underlying communication subsystem. Fundamentally, the communication system must provide the means to return control to the calling application while some or all of the communication operation takes place in the background. In turn, the application must be written to exploit this capability, both by using the appropriate part of the communication system's API and by structuring code to maximize overlap.

MPI [17] has been the most popular standard for developing parallel scientific applications.The MPI interface includes non-blocking point-to-point calls (i.e. `MPI_Isend`, `MPI_Irecv`) which allow the separation of initiation and completion of message transfers. Additionally, even with blocking operations, the system can transparently allow for overlap by copying data to internal message buffers and returning from the MPI call to the user program before the data transfer is completed. ARMCI [20] is another communication system which underlies a number of parallel programming models, and is used in a range of high-performance applications. ARMCI focuses on one-sided communication, which does not require explicit coordination of sender and receiver.

However, it is not just the API for the communication subsystem which allows applications to hide latencies – the underlying implementation is also very important. Depending on the communication software and the underlying hardware, it may or may not be possible for communications to make progress when control has returned to the application. For example, previous studies [34, 21] have shown that some message-passing libraries achieve very low overlap for large messages even with non-blocking calls. This is explained by the fact that these libraries have a single-threaded monolithic architecture, a polling-based progress engine and a synchronous message completion and notification mechanism [7]. Polling progress in these libraries requires that communicating processes make frequent calls that invoke the progress engine to ensure continuous transfer progress. Communication software with asynchronous message completion is better positioned in this respect vis-a-vis a system with synchronous completion [9]. Remote Direct Memory Access (RDMA) operations on modern interconnects deliver data directly from source application memory to destination application memory without host processor involvement or intermediate copies of the data. RDMA is a particularly attractive option for implementing zero-copy transfers of large messages and is being increasingly employed in communication libraries. The choice of RDMA Write versus RDMA Read also has an impact on the overlap capability of the communication library [27].

The variety of communication hardware/software, together with a lack of effective tools for measuring the overlap of computation and communication, make it very challenging for application developers to understand the performance of their codes on different platforms and the potential for improvement. In this preport, we present an instrumentation framework to quantitatively assess the extent of overlap in parallel applications for both two-sided (MPI) and one-sided (ARMCI) communication. A fundamental difficulty in instrumenting communication libraries to gauge the extent of achieved overlap is that the initiation of data transfer is done by NICs and often the precise times for NIC-initiated data transfer events is unknown to the host processor. In the absence of precise NIC-level time-stamps, we take the approach of estimating lower and upper bounds on the achieved overlap of communication and computation. We show that these bounds can provide performance insights that may be used to make code changes to improve performance.

The rest of this report is organized as follows. In Sec. 2, we present the conceptual framework for our approach, and describe how this translates into instrumentation of the communication libraries used in this work. In Sec. 3, we validate our approach with synthetic micro-benchmarks, illustrating how different communication protocols within an MPI library can affect performance. In Sec. 4, we apply our approach to characterize and improve the performance of the NAS parallel benchmarks. We provide a discussion of related work in (Sec. 5).

2 Overlap Measurement Framework

Precise measurement of overlap of communication and computation in modern HPC systems is very challenging. A fundamental issue is that data transfer is initiated by NICs, so that the processor is unaware of the exact time of initiation. Our approach to overcome this problem is to develop *bounds* on the degree of overlap based on instrumentation of the communication library to capture time-stamps when the application enters and leaves regions of communication and computation.

2.1 Terminology

The PERUSE specification [25] defines events internal to MPI implementations, primarily for the purposes of facilitating the development of performance monitoring.

Though our approach is not limited to the MPI standard, we follow PERUSE's definition of a *message transfer*

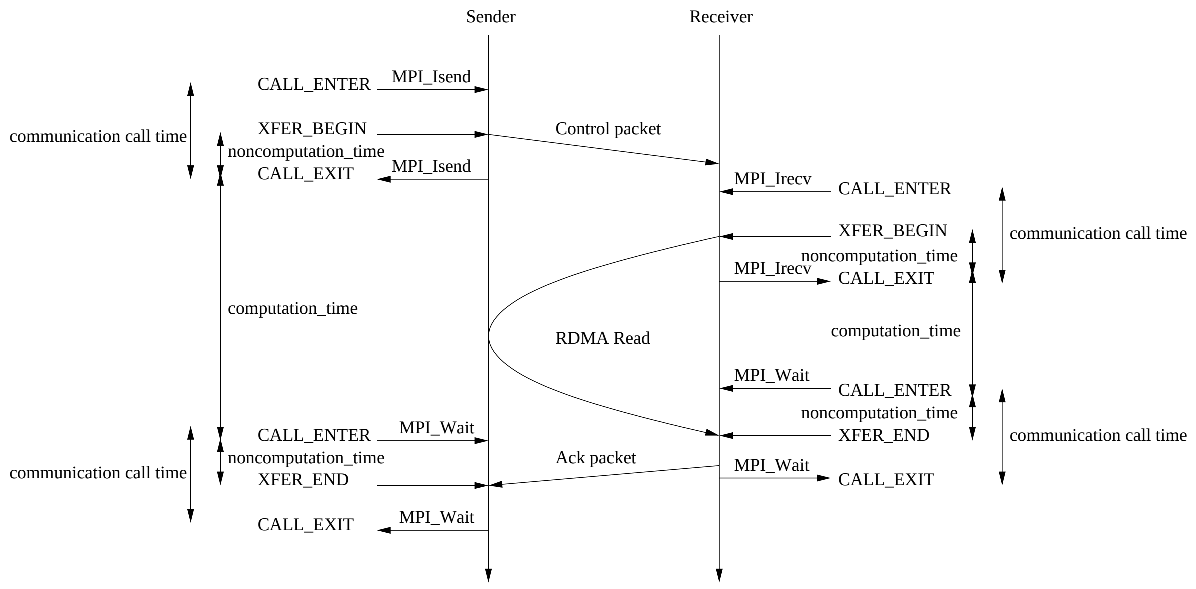


Figure 1. Time-stamped RDMA Read based point-to-point exchange

as the collection of (one or more) *data transfers* that actually perform the physical transfer of the entire user message, not including control packets that might be used by the MPI library for implementing internal protocols and flow control schemes. For example, the control packets associated with rendezvous protocols are not considered part of the message transfer. We also define a number of events which are in the *spirit* of the PERUSE standard, but do not conform precisely to its definitions.

`XFER_BEGIN` and `XFER_END` events mark the start and completion of *data transfer* operations that move user messages. Such events are assumed to be provided by the communication library as approximations to the actual beginning and end of the physical (hardware-level) transfer of the data. For example, with Mellanox VAPI [16] on InfiniBand, `XFER_BEGIN` will correspond to the time at which the library posts a `VAPI_post_sr` request and `XFER_END` will indicate completion of the request detected via return from a `VAPI_poll_cq` call. The closer these events are to the actual physical data transfer events, the more accurate the resulting overlap estimate.

`CALL_ENTER` and `CALL_EXIT` events mark the beginning and end of calls *from* the application code *into* the communication library. These demarcate user computation and communication call regions.

2.2 Measurement Approach

The communication library is instrumented to capture the time-stamps for the four events defined above. From these time-stamps, we can derive the following measures on a per-process basis:

1. *data transfer time* - the time for physical transfer of all user messages sent and received by a process.
2. *minimum overlapped transfer time* - a fraction of data transfer time, denoting a lower bound on overlapped transfer time.
3. *maximum overlapped transfer time* - a fraction of data transfer time, denoting an upper bound on overlapped transfer time.
4. *user computation time* - time incurred in performing user computations.
5. *communication call time* - aggregate time spent executing communication calls.

Figure 1 illustrates the events and the relevant time measures in calculating the overlap bounds for an RDMA Read-based point-to-point exchange.

The amount of overlap is determined on a per-data-transfer basis and these individual estimates are added to give overlap bounds for each process. Either `XFER_BEGIN` or `XFER_END` or both events may have been time-stamped for a given transfer operation. Since communication events are timed in the library and the library only has an approximate view of the actual movement of data, the real overlap is estimated to lie within a range defined by lower (minimum overlapped transfer time) and upper (maximum overlapped transfer time) bounds. We consider three cases:

1. `XFER_BEGIN` and `XFER_END` are time-stamped within the same communication call. This means that the associated transfer happened while the application was executing inside the communication library and no useful computation could be performed during this period. Hence, both the minimum and maximum overlapped transfer times are zero.

2. `XFER_BEGIN` and `XFER_END` are not time-stamped within the same communication call. In other words, there might be a sequence of user computation and library communication periods interleaved in between. If there is sufficient interleaved computation (*computation_time*) to overlap with the associated transfer (*xfer_time*) i.e. *computation_time* $>=$ *xfer_time*, then there is potential for complete overlap and therefore the maximum overlapped transfer time is *xfer_time*. Here, *xfer_time* denotes the time for the data transfer operation on the network that is measured a priori by running a standard microbenchmark test and *computation_time* is the total length of computation appearing between `XFER_BEGIN` and `XFER_END`. On the other hand, if there is insufficient intermediate computation i.e. *computation_time* $<$ *xfer_time*, only a limited amount of computation could possibly be overlapped. So, maximum overlapped transfer time is set to *computation_time*.

   The minimum overlapped transfer time is calculated on similar lines. If the net time incurred within the communication library (*noncomputation_time*) exceeds the associated transfer time (*xfer_time*) i.e. *noncomputation_time* $>=$ *xfer_time*, then there was potentially zero overlap and minimum overlapped transfer time is accordingly set to zero. Otherwise, there was at least (*xfer_time* – *noncomputation_time*) amount of overlap. So, in this case, minimum overlapped transfer time is set to be (*xfer_time* – *noncomputation_time*). Here, *noncomputation_time* is the time spent in the communication library between `XFER_BEGIN` and `XFER_END`.

3. Only `XFER_BEGIN` or `XFER_END` is time-stamped but not both. As it is impossible to be conclusive regarding the achieved overlap, the minimum overlapped transfer time is set to zero and the maximum overlapped transfer time is set to *xfer_time*.

Data transfer time is the net time for operations that perform the physical transfer of all user messages sent and received by a process, excluding any control messages. The sum of *xfer_time* of individual data transfer operations gives the total data transfer time. The time interval between every pair of `CALL_EXIT` to next `CALL_ENTER` events is aggregated as user computation time. The time interval between every pair of `CALL_ENTER` to next `CALL_EXIT` events is aggregated as communication call time.

It is worth noting that for portability and ease of support, we have designed our approach to be implementable within the communication library, without requiring lower-level support from the hardware or drivers. However, if it were possible to obtain time-stamps on data transfers from the network interface card, a more precise characterization of the overlap would be possible.

2.3 Interpretation of derived measures

We now explain what the measures mean to application developers or system designers from a performance and scalability perspective.

1. The difference between *data transfer time* and *maximum overlapped transfer time* gives the minimum duration of communication that was not usefully overlapped with computation. This can be an indicator of overall application performance loss as non-overlapped communication will typically manifest itself as time spent waiting, during which no useful work could be done. An application developer engaged in a performance or scalability exercise is essentially interested in learning how much time was consumed in different portions of the application and whether there is scope for improvement. Thus, a high non-overlapped transfer time points to a significant source of overhead. A breakdown of this time as a function of message size distribution, such as "short" versus "long", or a more detailed size distribution, will reveal the particular message transfers that are affecting application performance the most.

   We allow for this level of detail in our implementation of the framework. We also provide application-level control over sections of code to be monitored. Combined, the two features are useful in performance studies. We have used these capabilities in not only diagnosing but also improving overlap in a particular portion of the NAS SP benchmark. For this benchmark, even though explicit overlap was being attempted within the code structure, in reality the full potential for overlap was not being exploited. As we show in Sec. 4.3, the data from instrumentation was used to make code changes that lead to a sizeable reduction in overall MPI time.

2. The absolute value for *minimum overlapped transfer time* is a clear savings in execution time due to achieved overlap. The impact of code changes on values of both bounds is a useful indicator of the effectiveness of those changes from an overlap standpoint.

2.4 Implementation of Measurement Framework

We have implemented the framework within the Open MPI (v1.0.1) [10] and MVAPICH2 (v0.6.5) [15] implementations of the MPI standard on InfiniBand, the ARMCI (v1.1) [20] one-sided communication library on InfiniBand, and the Open MPI (v1.0.2) message-passing library on Myrinet. The implementation comprises of instrumentation incorporated into the library to generate time-stamped events during application execution, and the monitoring infrastructure to determine overlap estimates by online processing of events.

In section 2.4.1, we explain our instrumentation technique using Open MPI as an example. We have applied the same ideas in instrumenting MVAPICH2 and ARMCI, and hence we briefly discuss the communication protocols employed in these implementations, omitting details of their instrumentation. In section 2.4.2, we illustrate the structure of our monitoring framework.

2.4.1 Instrumentation of communication protocols

Open MPI uses eager and rendezvous protocols for communicating short and long messages, respectively [35]. For short messages, a send descriptor is allocated and user data is packed into the descriptor. The descriptor is then sent by invoking a transfer operation on the underlying communication software. For example, a `VAPI_SEND` work request is posted by making a `VAPI_post_sr` or `EVAPI_post_inline_sr` call to the Mellanox VAPI layer on InfiniBand in the v1.0 series of Open MPI. For the Myrinet GM library, `gm_send_with_callback` will be called. On send completion, the send request is returned and the descriptor is freed. On receiver side, messages are received via the progress engine. When data is matched to a receive request, it is unpacked into the user's buffer and the receive request is signaled complete at the MPI level.

The instrumented eager protocol is shown in figure 2. For the sending process, the posting of a send descriptor is marked as `XFER_BEGIN` and send completion is recorded as `XFER_END`. A short message transfer is transparent to the receiving process and so there is no `XFER_BEGIN` event on the receiver side. `XFER_END` refers to the arrival of the message, detected by the progress engine of the receiving process.

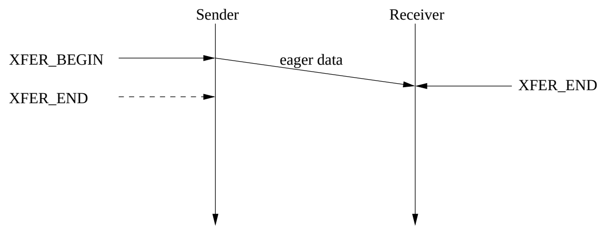


Figure 2. Instrumentation of eager protocol in Open MPI

Open MPI employs two alternate approaches for long messages. In the default scheme, a long message is fragmented and the fragments can be scheduled for delivery across multiple NICs. Initially, a combined send request along with the first fragment descriptor is sent, which has to be acknowledged by the receiver. Once the acknowledgment is received, the sender pipelines remaining fragments using a scheduling algorithm. On networks with RDMA capabilities, the receiver can schedule RDMA operations by the sender. The sender pipelines send descriptors to overlap initial registration/setup costs of the RDMA. The receiver schedules RDMA by communicating a PUT control message with the RDMA target address to the sender. Upon receiving this message, the sender performs an RDMA Write into the receive application buffer, followed by a FIN message to indicate write completion.

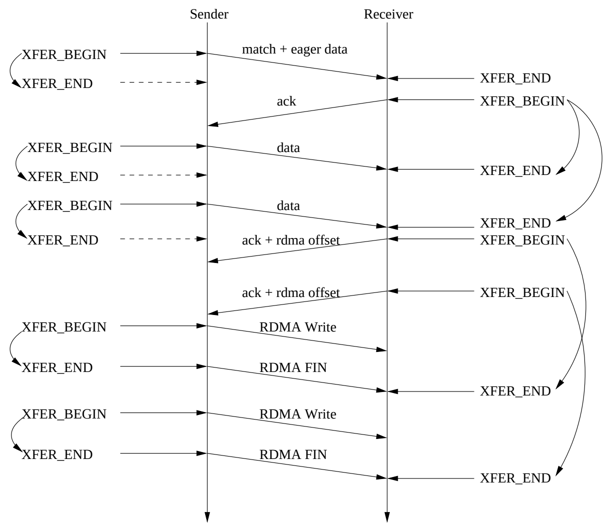


Figure 3. Instrumentation of pipelined rendezvous protocol in Open MPI

Figure 3 illustrates the instrumented rendezvous protocol for the default scheme. As before, the sending process times the posting/completion of individual send descriptors as `XFER_BEGIN`/`XFER_END`. Likewise, the invocation/completion of individual RDMA Write operations are marked as `XFER_BEGIN`/`XFER_END`. The RDMA transfer, though transparent to the receiving process, can only happen in the interval between posting of PUT packet and arrival of FIN packet. Hence, the receiving process identifies the time at which the PUT message is scheduled for delivery as `XFER_BEGIN` and the time at which the FIN message is discovered as `XFER_END`. Further, the acknowledgment of send request is recorded as `XFER_BEGIN`, with `XFER_END` referring to the arrival of eager data.

Open MPI defines a run-time parameter named `mpi_leave_pinned` that can be used to bypass pipelined RDMA for contiguous data. This second approach supports caching of registrations in a most recently used list. A single RDMA Read or Write may be initiated on the registered user buffer. For RDMA Write, the sender notifies the receiver of the request and the receiver responds with a PUT control message bearing the receive buffer address. The sender then writes the data into the receive buffer. And on completion of the write operation, a FIN message is sent to the receiver. For RDMA Read, the receiver is notified of the request as well as the sending buffer address in a GET control message. The receiver performs a read of the send user buffer and communicates a FIN message to indicate read completion.

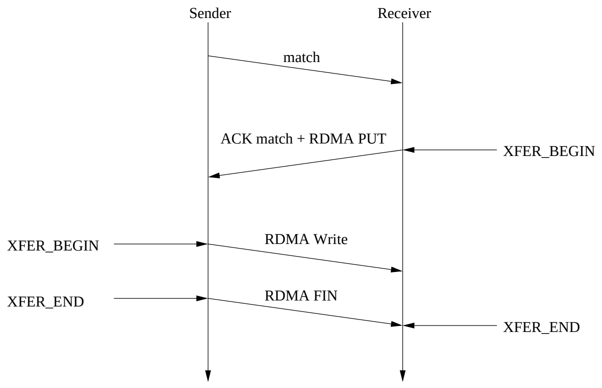


Figure 4. Instrumentation of rendezvous protocol with direct RDMA Write in Open MPI

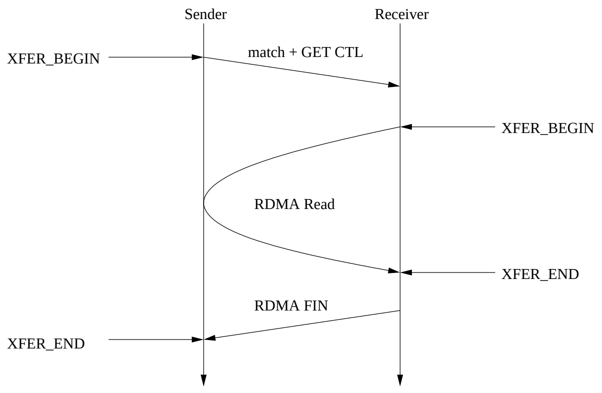


Figure 5. Instrumentation of rendezvous protocol with direct RDMA Read in Open MPI

The placement of events for the two leave pinned schemes appears in figures 4 and 5. The side doing RDMA denotes the start/end of the operation as `XFER_BEGIN`/`XFER_END`. For the other side, the posting of PUT/GET message is identified as `XFER_BEGIN`; the discovery of FIN message is marked as `XFER_END`.

MVAPICH2-0.6.5 has been designed at the MPICH2 RDMA Channel interface for InfiniBand [15]. MVAPICH2 implements put and get routines defined by the interface for both eager and rendezvous transfers using RDMA support in InfiniBand. With the eager protocol, the sender copies the message to pre-registered buffers and the message is written out into the receiving pre-registered buffers by an RDMA Write operation. Rendezvous transfer is zero-copy, with the sending user's buffer being pinned on-the-fly and the receiver doing an RDMA Read on this buffer.

ARMCI supports the remote memory access (RMA) communication model over Mellanox VAPI on InfiniBand using a host-assisted zero-copy approach [30], provided source and destination memories are registered. This approach involves a helper thread on the remote side to assist in the completion of communication. With both source and destination memories registered, PUT and GET of contiguous data are done using RDMA Write and RDMA Read, respectively.

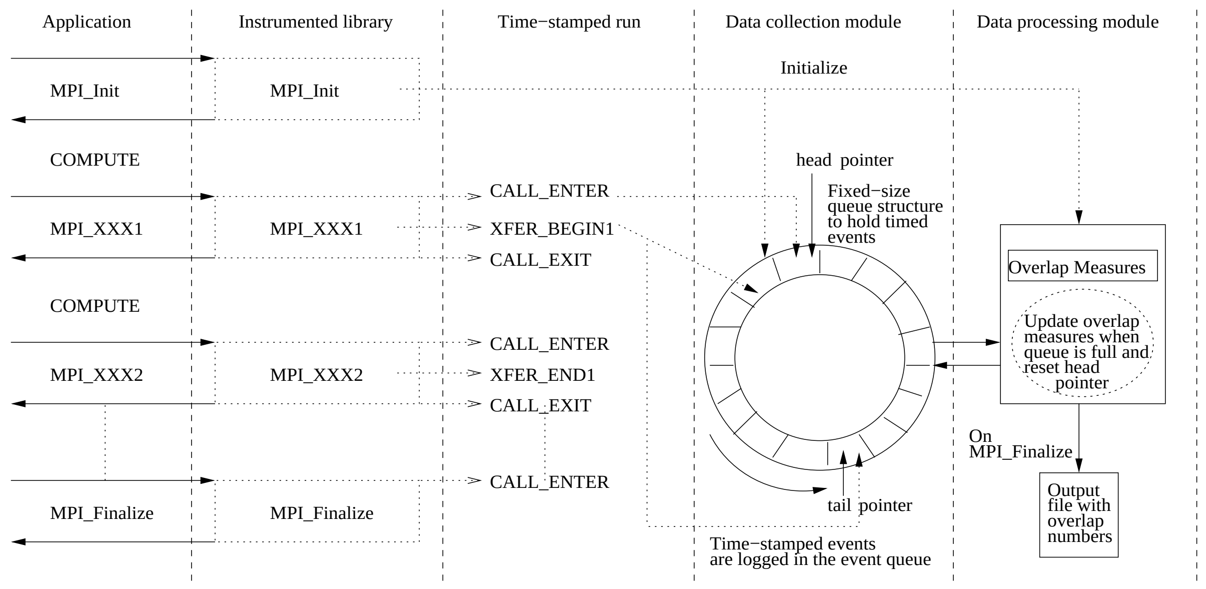


Figure 6. Structure of framework

2.4.2 Structure of framework

Figure 6 displays the structure of the monitoring framework. The framework is instantiated at the individual process level and operates locally without performing any interprocessor communication. When the application terminates, an output file is generated for each process, with information about overlap achieved by that process. The reported information only characterizes the local process communication activity.

The monitoring framework comprises of two main components, a data collection module and a data processing module. During application startup, a file containing transfer times is read from disk into memory. As the application executes, the data collection module logs time-stamped events in a circular-queue data structure. This is a fixed-size, in-memory, statically allocated structure that is processed periodically by the data processing module. No tracing is performed; information is maintained only for the set of currently active events. When the queue is full, the data processing module examines the events, updates overlap numbers on-the-fly, and resets the head pointer to allow for subsequent events to be stored. At the end of the application run, the obtained overlap readings are written out to a file. Overall, this is very similar to the profiling approach that is widely used in current performance monitoring tools. Because the instrumentation itself involves no interprocessor communications, and is not dependent on the number of processors used by the application (except for the startup and shutdown), it is scalable to large processor counts.

3 Experimental Evaluation with Microbenchmarks

In this section, we report on experimental measurements with instrumentation incorporated into the Open MPI implementations on InfiniBand and Myrinet. We focus primarily on Open MPI because it was deployed for different underlying networks, allowing measurements on more than one communication network.

3.1 Experiment

We ran an overlap test in which two processes communicate a message using different combinations of point-to-point MPI calls with increasing computation inserted between the initiating and wait non-blocking methods.

One process acts as a sender calling only `MPI_Send` or `MPI_Isend` methods, while the other process acts as a receiver calling only `MPI_Recv` or `MPI_Irecv` methods. The two processes were run on separate nodes, and measurements were made of a loop of 1,000 transfers with message sizes of 10 KB and 1 MB. We measured the average time spent in `MPI_Wait` as well as the minimum and maximum percentage overlap for the side doing non-blocking communication in each run and plotted these against increasing computation time.

3.2 Open MPI on InfiniBand

3.2.1 Test Platform

The experiments were conducted using a cluster at the Ohio Supercomputer Center. Each compute node has 4GB RAM, two 2.4GHz Intel P4 Xeon processors, 512KB of secondary cache and runs a patched version pre1 of Linux 2.4.24 OS. The nodes are connected by a switched 8 Gbit/s InfiniBand network, using MT23108 InfiniHost (rev a1) by Mellanox Technologies on the 64-bit 133MHz PCI-X bus. The driver version is ibgd-1.6.0. All codes including libraries and test codes were compiled using Intel 8.0 suite of optimizing compilers with -O3 flag. Our experiments were run with one MPI process per dual-processor node in order to test communication across the network. The `perf_main` utility provided with Mellanox IB drivers was used a priori to characterize data transfer times for various message sizes.

3.2.2 Eager exchange

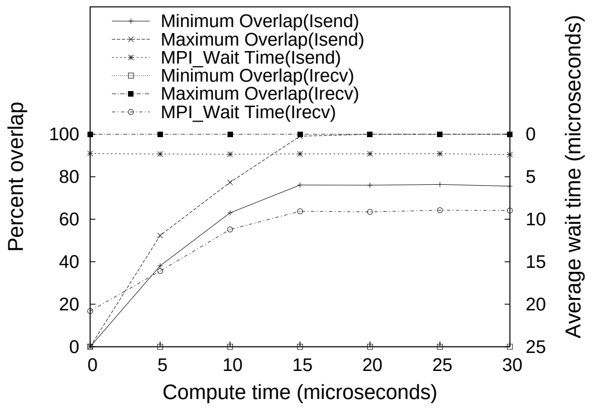


Figure 7. Isend-Irecv and eager protocol

Eager transfers are characterized by similar behavior across point-to-point calls and hence only `Isend` – `Irecv` results are plotted in figure 7. Since the precise arrival time of a message is never known in the polling mode of operation and the initiation of the send is transparent to the receiver, we always assert minimum overlap as zero and maximum overlap as the message transfer time for the receiver. From the standpoint of sender, greater computation implies greater scope for overlapping communication. The increase in overlap percentages with increase in computation time reflect this observation. Also note that once the overlap percentages cease to increase, the wait time for the receiver does not change further. To summarize, short message transfers exhibit full overlap ability.

3.2.3 Rendezvous exchange

We evaluated both the pipelined scheme and the direct(leave pinned) RDMA Read scheme for overlap.

1. `Isend` – `Recv` pair

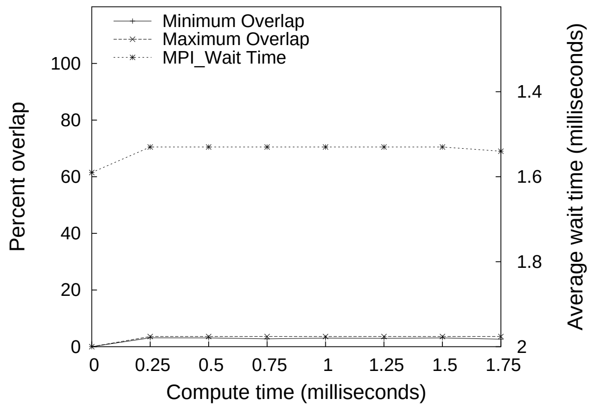


Figure 8. Isend-Recv and pipelined RDMA

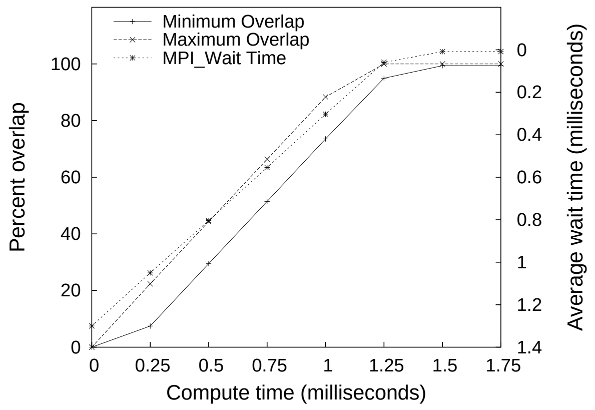


Figure 9. Isend-Recv and direct RDMA

The overlap plots for the sender in `Isend` – `Recv` exchange appear in figures 8 and 9. The pipelined RDMA scheme is only able to overlap the initial fragment. Therefore, the overlap curves remain flat even with increasing computation. This happens because the sender posts the initial request in `MPI_Isend`, starts computing and detects the acknowledgment message from the receiver after it enters the wait method. It then schedules additional fragments which do not get overlapped. In contrast, the receiver is free to read the sending application's buffer on arrival of the initial request in the direct RDMA scheme. This explains the improved overlap when computation is increased and the progressive drop in wait time further confirms this trend. With full computation-communication overlap, the wait time does not change any further. Note that the wait time curve closely follows the overlap curves.

2. `Send` – `Irecv` pair

Both schemes exhibit minimal overlap in `Send` – `Irecv` communication as seen from the overlap plots for the receiver in figures 10 and 11. Since the progress engine is polling-based, the receiver detects the initial request only on entering `MPI_Wait`. This means there is zero overlap for direct RDMA whereas pipelined RDMA is able to overlap the first fragment. Consequently, the wait time is high and is unchanged for varying computation lengths.

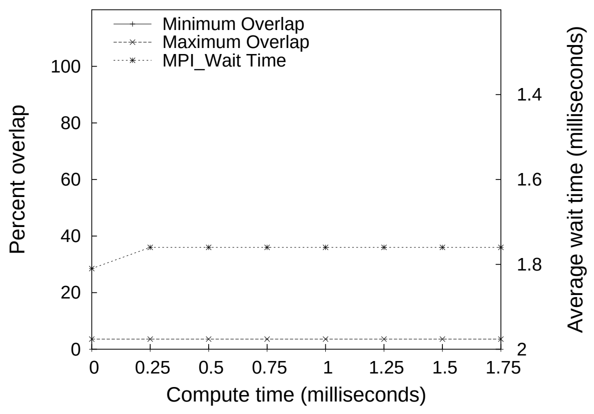


Figure 10. Send-Irecv and pipelined RDMA

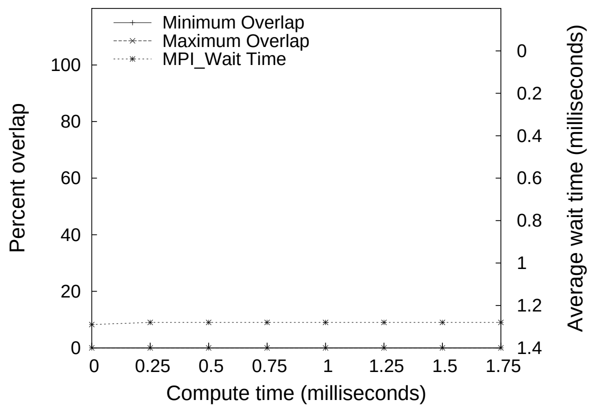


Figure 11. Send-Irecv and direct RDMA

3. `Isend` – `Irecv` pair

   As depicted in figures 12 and 13, while the direct RDMA approach allows the possibility of complete overlap for the sender, the initiating fragment is the only portion of the message that is overlapped in pipelined RDMA.

3.3 Open MPI on Myrinet

3.3.1 Test Platform

The experiments were conducted using a cluster at the Ohio Supercomputer Center. Each compute node has 4GB RAM and two Intel 64 bit processors running Linux 2.4.21-sgi306rp21 OS. There are two partitions of compute nodes, one with 900MHz processors, and second with 1.3 GHz processors. The nodes are connected by 2 Gbit/s Myrinet high speed interconnects, using Myrinet 2000 C or D NICs on the 64-bit 133MHz PCI-X buses. The driver version is GM-2.1.4. All codes including libraries and test codes were compiled using Intel 8.0 suite of optimizing compilers with -O3 flag. Our experiments were run with one MPI process per dual-processor node in order to test communication across the network. The `gm_allsize` utility provided with GM drivers was used a priori to characterize data transfer times for various message sizes.

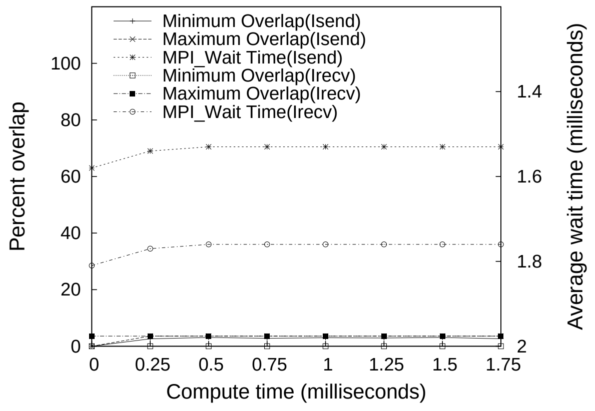


Figure 12. Isend-Irecv and pipelined RDMA

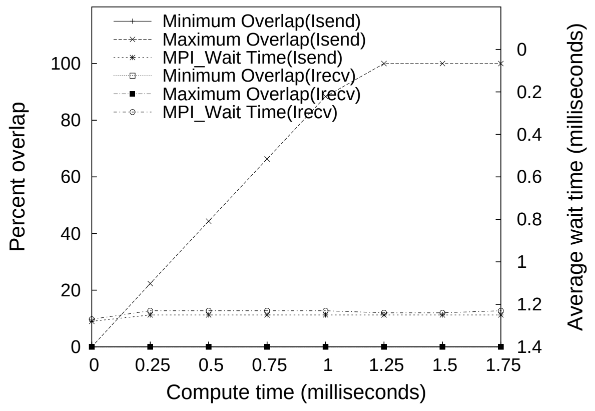


Figure 13. Isend-Irecv and direct RDMA

3.3.2 Eager exchange

As depicted in figure 14, the short message overlap behavior on Myrinet mirrors the behavior on InfiniBand.

3.3.3 Rendezvous exchange

Overlap results in pipelined rendezvous mode for `Isend` – `Irecv` exchange are plotted in figures 15 and 16. From the standpoint of both sender and receiver, only the initial fragment can be overlapped. Therefore, the overlap curves remain flat even with increasing computation. The sender posts the initial request in `MPI_Isend`, starts computing and detects the acknowledgment message from the receiver after entering the wait method. It then schedules additional fragments which do not get overlapped. Since the polling engine is progress-based, the receiver detects the initial request on entering `MPI_Wait`. Consequently, following fragments are received while in the `MPI_Wait` method.

4 Application Experience

The setup for the experiments in this section was identical to that described in 3.2.1 and 3.3.1. We characterized each NAS benchmark from the NPB 3.2 suite in one of the communication environments previously described. To

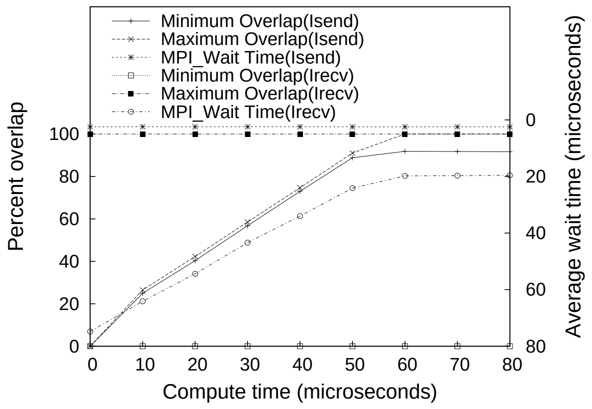


Figure 14. Overlap in eager protocol

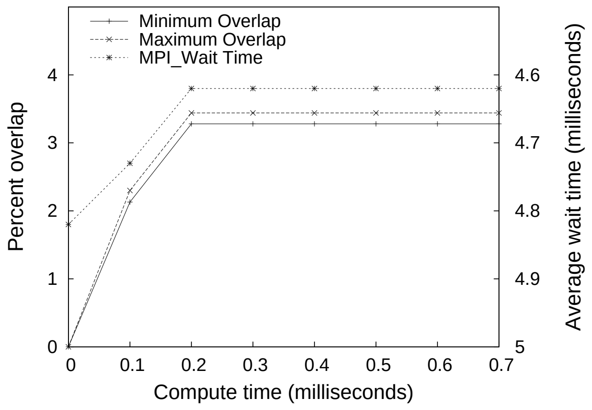


Figure 15. Sender overlap in pipelined rendezvous protocol

provide some representative data, we report performance of BT and CG with Open MPI v1.0.1 on InfiniBand; LU and FT with MVAPICH2-0.6.5; and MG with ARMCI v1.1. We used the instrumentation results to achieve greater overlap in SP with MVAPICH2 on InfiniBand and Open MPI v1.0.2 on Myrinet. We do not report performance of EP as it performs minimal communication and IS because it exhibits similar overlap behavior to FT. Each process was individually monitored for overlap and we present data for process 0. Data was gathered for different message size ranges, to provide information on the degree of overlap for messages of different sizes. We briefly discuss our findings in each case.

4.1 NAS BT and CG with Open MPI

Overlap results for NAS BT and CG with Open MPI supporting the pipelined RDMA mode are graphed in figures 17 and 18 respectively. Both BT and CG primarily use point-to-point messages and do limited collective communication. While long messages constitute the majority of communication for BT, CG sends a larger proportion of short messages. As the microbenchmark evaluation showed, short messages with low transfer times allow for complete overlap. Consequently the overlap results are higher for CG than for BT. For larger problem sizes and smaller processor counts, long messages dominate. Since long messages have less potential for overlap, observed overlaps drop for both applications.

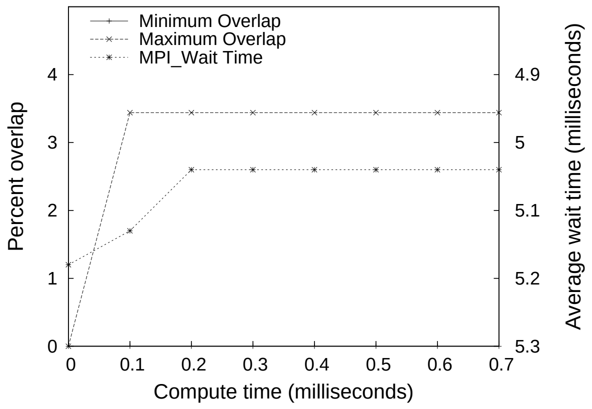


Figure 16. Receiver overlap in pipelined rendezvous protocol

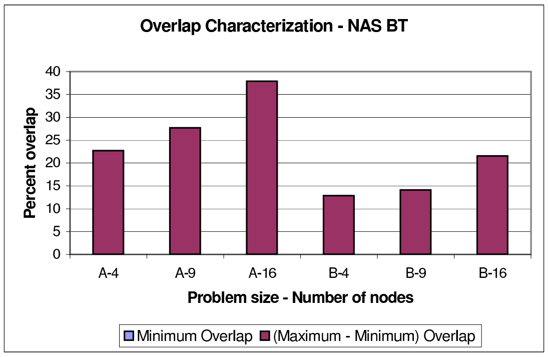


Figure 17. Overlap Characterization - NAS BT

4.2 NAS LU and FT with MVAPICH2

Figure 19 depicts very high overlap potential for the LU benchmark. LU primarily performs point-to-point communication with a mix of short and long messages. A substantial portion of the payload comprises of short messages. With decreasing problem size and increasing number of processors, the percentage of short messages increases further. Short message transfers exhibit good overlap behavior - LU overlap numbers are above 70% and increase as the problem size is reduced or the processor count is increased. The non-overlapped time is incurred in communicating long messages.

FT has low scope for overlap as is brought out in figure 20. Most of the communication in FT is done by the `Alltoall` collective which sends long messages. These transfers do not get overlapped with computation. The limited amount of overlap is due to short messages being exchanged in collectives like `Reduce` and `Bcast`.

4.3 Improving overlap in NAS SP

In this section, we explain how we used the overlap measurements for characterizing and improving achieved overlap with the NAS SP benchmark. We provide results on two platforms, with Open MPI on Myrinet and MVAPICH2 on InfiniBand. SP explicitly attempts overlap of computation and communication in the `x_solve`, `y_solve`, and `z_solve` routines, which employ the Thomas algorithm for solution of the approximate factorization step in the x, y, and z directions respectively. This is done at two places in the code, by computing in between the posting

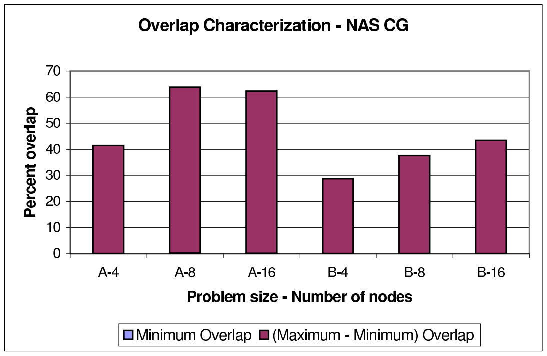


Figure 18. Overlap Characterization - NAS CG

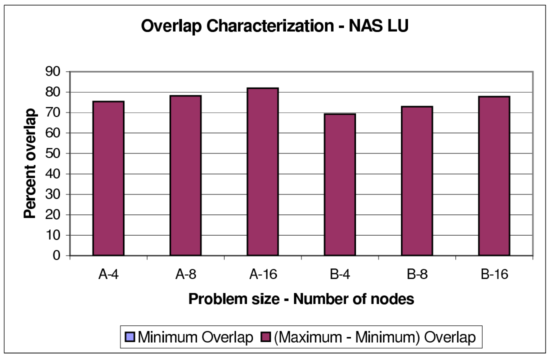


Figure 19. Overlap Characterization - NAS LU

of an `Irecv` and waiting for the communication to complete.

We obtained the overlap estimates for the entire code as well as limited to the code region where overlap is attempted. We gathered overlap readings for messages of different sizes and observed a high non-overlapped overhead for messages that are communicated in the overlapping section. We then placed `Iprobe` calls at multiple locations in the computation region of the overlapping section. We tried different numbers as well as positions of `Iprobe` calls, each time measuring the change in overlap. For all tested problem sizes and processor counts, there is a performance benefit with overall MPI time showing a drop as revealed in figures 21 and 22. We were able to make substantial improvements for the overlapping section in all cases as can be seen in figures 23 and 24. We provide the original and modified overlap numbers for a complete run in figures 25 and 26. The gains over the complete code are limited by a substantial volume of data being communicated in routine `copy_faces` with no computation to overlap.

4.4 NAS MG with ARMCI

In a previous effort [29], the NPB2.4 MPI version of the MG benchmark was modified to replace point-to-point blocking and non-blocking message-passing communication calls first with blocking and then non-blocking ARMCI calls. The ARMCI non-blocking version achieved improved performance over the ARMCI blocking version by issuing non-blocking update in the next dimension before actually working on the data in the current dimension. The improvement was attributed to greater overlap being achieved by non-blocking calls. We instrumented ARMCI v1.1 to quantify the degree of overlap for the two codes. The results are plotted in figure 27.

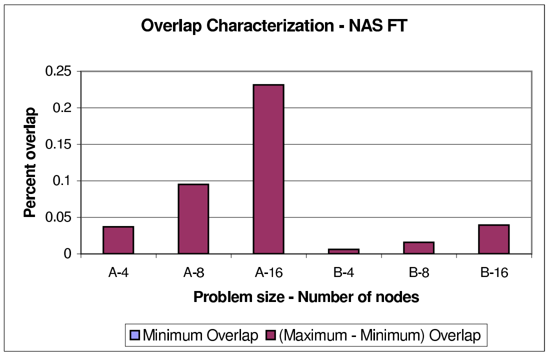


Figure 20. Overlap Characterization - NAS FT

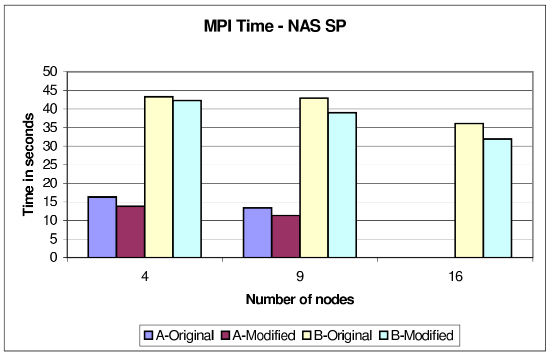


Figure 21. Original and Modified MPI times of NAS SP on Myrinet with Open MPI for problem sizes A and B

The non-blocking code shows very high maximum overlap percentage, with 99% overlap being reported for all processor counts with problem size B.

4.5 Instrumentation Overhead

To illustrate the impact of the overlap characterization instrumentation on the application run time, we re-ran the NAS benchmarks using the original, uninstrumented versions of Open MPI and MVAPICH2. The results, shown in figure 28, show an instrumentation overhead of less than 0.9% of the total execution time for all test cases. These results are expected to scale to much larger processor counts, since the actual monitoring is process-local. However, we must point out the one-time initial penalty of storing transfer times for messages of different sizes from a disk-resident file into memory. The reported execution times do not reflect this cost as it is being done during `MPI_Init`.

5 Related Work

A number of tools, both commercial and research, have been developed for obtaining performance data for message-passing codes on high-end computing platforms [18]. However, the focus of these efforts has been on different aspects of MPI performance rather than the ability to overlap communication and computation. Profiling

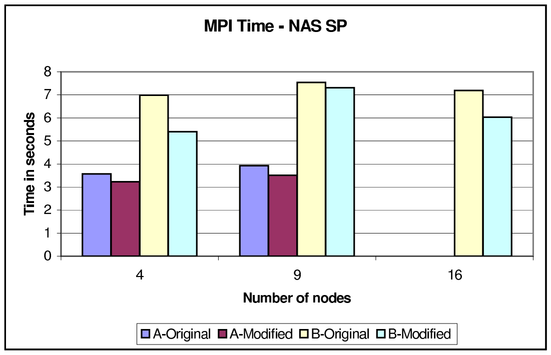


Figure 22. Original and Modified MPI times of NAS SP on InfiniBand with MVAPICH2 for problem sizes A and B

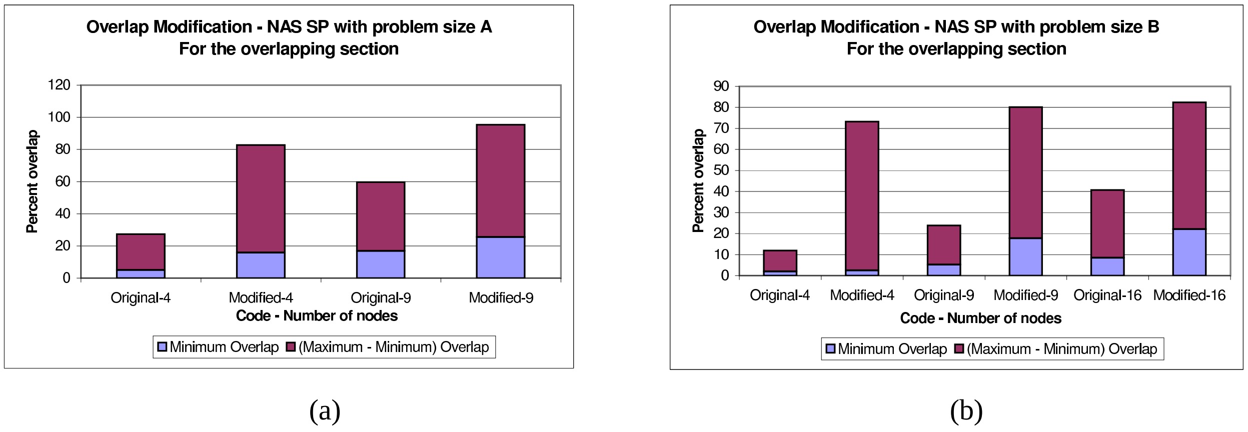
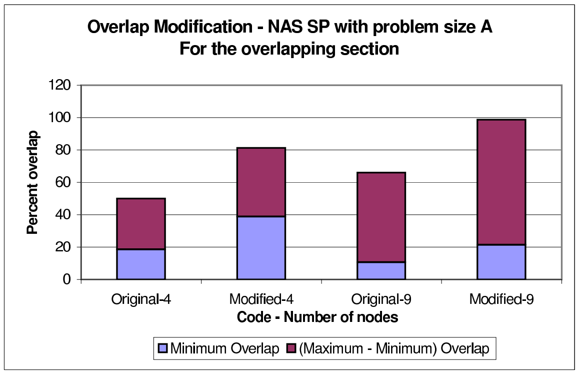


Figure 23. Original and Modified case overlap measurement over the overlapping section of NAS SP on Myrinet with Open MPI for problem size (a) A (b) B

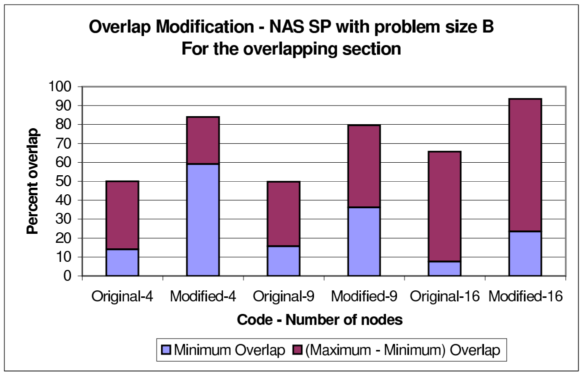
or tracing at the PMPI interface, supplemented by various analysis and visualization techniques is the widely adopted approach in current tools. Current performance tools also differ in their approach to instrumentation and analysis of performance data.

TAU (Tuning and Analysis Utilities) [28] profile data summarizes application behavior in terms of metrics like message count, message size distribution, inclusive/exclusive time, number of invocations, number of child subroutine calls etc. SvPablo [22] supports interactive and automatic instrumentation of subroutine calls and loops, can be configured to perform profiling or tracing, and reports performance via numerous metrics. KOJAK [13] instruments MPI programs using the PMPI interposition library, followed by manual or automatic analysis of gathered trace data. Paradyn [23] dynamically instruments an executing parallel program and generates performance outputs in real-time. mpiP [19, 33] is a lightweight, scalable, profiling library for MPI applications. While the above mentioned tools are publicly accessible, there are commercial tools in use, like the Intel Trace Analyzer and Collector [12], DEEP/MPI [5], Paraver [24], Dimemas [6] etc. Many of these tools are integrated with PAPI, thereby allowing application developers to access hardware performance counters in modern microprocessors.

PERUSE [25] reveals inner details of the MPI library and events occurring in lower system software layers. ChaMPIon/Pro and MPI/Pro [8] are commercial middleware products from Verari Systems Software that sup-

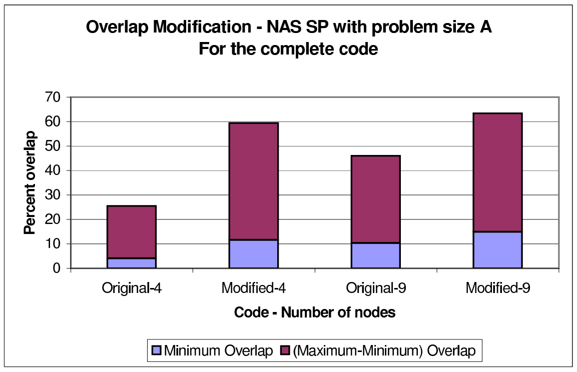


(a)

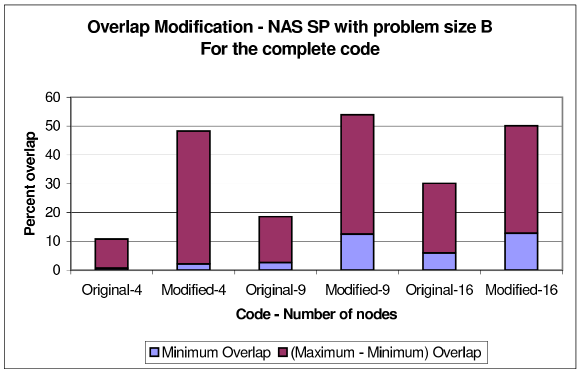


(b)

Figure 24. Original and Modified case overlap measurement over the overlapping section of NAS SP on InfiniBand with MVAPICH2 for problem size (a) A (b) B

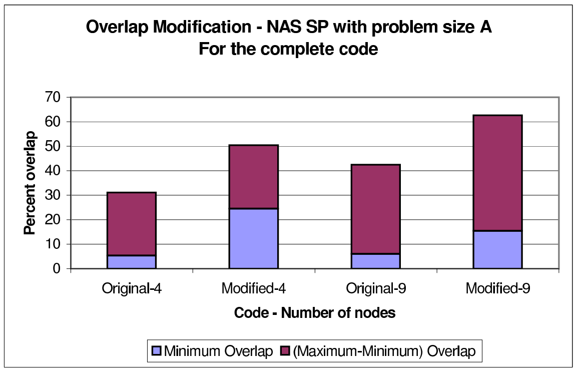


(a)

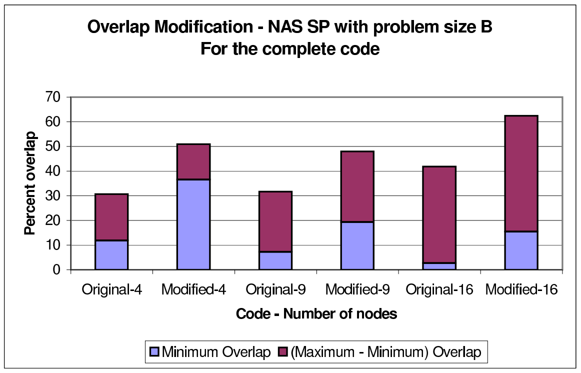


(b)

Figure 25. Original and Modified case overlap measurement over the complete code of NAS SP on Myrinet with Open MPI for problem size (a) A (b) B



(a)



(b)

Figure 26. Original and Modified case overlap measurement over the complete code of NAS SP on InfiniBand with MVAPICH2 for problem size (a) A (b) B

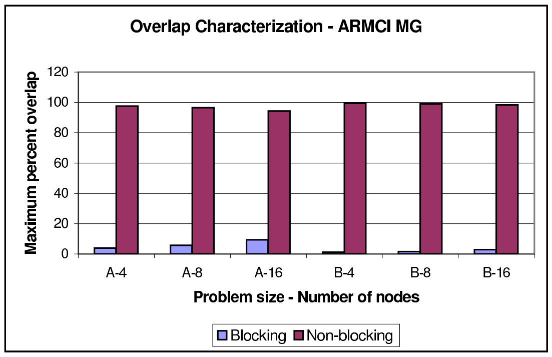


Figure 27. Overlap Characterization - ARMCI MG

port the PERUSE specification. Vetter's work [32, 31] enables performance analysis of communication behavior of message-passing applications. COMB [14] is a portable benchmark suite that assesses the ability of cluster networking hardware and software to overlap MPI communication and computation. It is not targeted at evaluating overlap from an application standpoint. The work in [4, 3] explores the advantages of overlap, independent progress, and offload in MPI implementations for NAS benchmarks by running different implementations on identical hardware.

To the best of our knowledge, there is no previous work to determine the degree of computation-communication overlap for applications in message-passing systems. Trace-based approaches may reveal scope for overlap, but not in a direct or easy to interpret fashion. Identifying bottlenecks from voluminous trace outputs is tedious. Also, trace-based approaches have to deal with problems like increases in wall-clock execution time due to the overhead of instrumentation, possibility of perturbing application behavior, and the overhead of storing voluminous trace files. Unlike tracing, we numerically quantify the extent of non-overlapped communication. Our instrumentation introduces very little overhead, and our technique can be used for evaluating different algorithm designs on different systems.

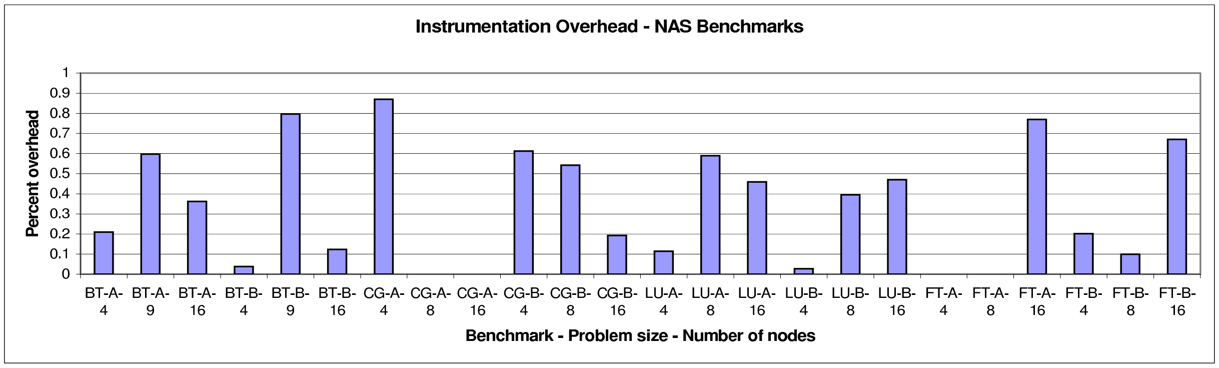


Figure 28. Instrumentation Overhead - NAS Benchmarks

6 Conclusions

In this report, we have described a framework for performance instrumentation, aimed at characterizing computation-communication overlap for message-passing applications. We instrumented the Open MPI, MVAPICH2 and ARMCI libraries and performed tests with micro-benchmarks and the NAS benchmarks on InfiniBand and Myrinet interconnects for varying problem sizes and processor counts. The overlap measures seek to quantify slowdown attributable to non-overlapped communication. Due to the unavailability of time-stamping support in NICs, and the characteristics of middleware and protocols used for communication, it is not possible to precisely identify the times of various communication events. We therefore developed an approach that generates minimum and maximum bounds on overlap of communication with user computation. The use of the instrumentation framework in enhancing the amount of overlap for one of the NAS benchmarks was demonstrated. We established that the instrumentation did not adversely impact running time of the tested applications. Because the instrumentation resides entirely within the communication library, it fits well with other performance monitoring approaches that operate outside the library.

The absence of NIC-level time-stamps on data transfers makes the overlap characterization dependent on details of the underlying communication subsystem. As communication libraries typically operate in user mode, they are often unaware of when the firmware and hardware actually perform the physical transfer of data over the network. Future work could harness time-stamps from NICs in precisely identifying communication time intervals. This will enable more accurate measurements of overlap of computation and communication.

7 Acknowledgments

We would like to thank Professor D.K. Panda, Wei Huang and Gopal Santhanaraman of the Network-Based Computing Laboratory at The Ohio State University for their help in understanding the internals of the MVAPICH2 implementation, and Jeff Vetter, Trey White, Pat Worley, Mark Fahey and Richard Barrett at Oak Ridge National Laboratory for very useful discussions on performance monitoring/instrumentation.

References

[1] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal. User-Level Network Interface Protocols. In *IEEE Computer*, pages 53–60, November 1998.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. In *IEEE Micro*, volume 15(1), pages 29–36, February 1995.

[3] R. Brightwell and K. D. Underwood. An Analysis of the Impact of MPI Overlap and Independent Progress. In *International Conference on Supercomputing (ICS)*, 2004.

[4] R. Brightwell, K. D. Underwood, and R. Riesen. An Initial Analysis of the Impact of Overlap and Independent Progress for MPI. In *EuroPVM/MPI*, 2004.

[5] DEEP/MPI. http://www.crescentbaysoftware.com/ deep_mpi_top.html.

[6] Dimemas. http://www.cepba.upc.es/dimemas.

[7] R. Dimitrov. *Overlapping of communication and computation and early binding: Fundamental mechanisms for improving parallel performance on clusters of workstations*. PhD thesis, Mississippi State University, 2001.

[8] R. Dimitrov. ChaMPIon/Pro - The Complete MPI-2 for Massively Parallel Linux, Linux Clusters: The HPC Revolution, 2004.

[9] R. Dimitrov and A. Skjellum. Impact of Latency on Applications' Performance. In *Fourth MPI Developer's and User's Conference*, March 2000.

[10] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[11] InfiniBand Trade Association. InfiniBand. http://www.infinibandta.org.

[12] Intel Trace Analyzer and Collector. http://www.intel.com/cd/software/products/asmona/eng/cluster/tanalyzer/index.htm.

[13] KOJAK. http://www.fz-juelich.de/zam/kojak.

[14] B. Lawry, R. Wilson, A. B. Maccabe, and R. Brightwell. COMB: A Portable Benchmark Suite for Assessing MPI Overlap. In *IEEE Cluster*, 2002.

[15] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2004.

[16] Mellanox Technologies. Mellanox VAPI Interface, July 2002.

[17] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. March 1994.

[18] S. Moore, D. Cronk, K. London, and J. Dongarra. Review of Performance Analysis Tools for MPI Parallel Programs. In *8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science*, volume 2131, pages 241–248. Springer Verlag, Berlin, September 2001.

[19] mpiP. http://www.llnl.gov/CASC/mpip.

[20] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High Performance Remote Memory Access Communications: The ARMCI Approach. In *International Journal of High Performance Computing Applications*, volume 20(2), pages 233–253, 2006.

[21] J. Nieplocha, V. Tipparaju, M. Krishnan, G. Santhanaraman, and D. K. Panda. Optimisation and performance evaluation of mechanisms for latency tolerance in remote memory access communication on clusters. In *International Journal of High Performance Computing and Networking (IJHPCN)*, volume 2, Issue 2/3/4, 2004.

[22] SvPablo. http://www.renci.org/software/pablo/svpablo.

[23] Paradyn. http://www.paradyn.org.

[24] Paraver. http://www.cepba.upc.es/paraver.

[25] PERUSE. http://www.mpi-peruse.org.

[26] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *Hot Interconnects 9*, August 2001.

[27] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, March 2006.

[28] TAU. http://www.cs.uoregon.edu/research/tau.

[29] V. Tipparaju, M. Krishnan, J. Nieplocha, G. Santhanaraman, and D. K. Panda. Exploiting Nonblocking Remote Memory Access Communication in Scientific Benchmarks on Clusters. In *HiPC*, 2003.

[30] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D. K. Panda. Host Assisted Zero-Copy Remote Memory Access Communication on InfiniBand. In *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2004.

[31] J. S. Vetter. Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficiencies. In *International Conference on Supercomputing (ICS)*, 2000.

[32] J. S. Vetter. Dynamic Statistical Profiling of Communication Activity in Distributed Applications. In *International Conference on Measurement and Modeling of Computer Systems*, 2002.

[33] J. S. Vetter and M. O. McCracken. Statistical Scalability Analysis of Communication Operations in Distributed Applications. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2001.

[34] J. B. White and S. W. Bova. Where's the Overlap? An Analysis of Popular MPI Implementations. In *Third MPI Developers' and Users' Conference*, March 1999.

[35] T. Woodall, R. Graham, R. Castain, D. Daniel, M. Sukalski, G. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. TEG: A high-performance, scalable, multi-network point-to-point communications methodology. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 303–310, Budapest, Hungary, September 2004.