Profile Guided MPI Protocol Selection for Point-to-Point Communication Calls

Aniruddha Marathe and David K. Lowenthal
Department of Computer Science
The University of Arizona
Tucson, AZ
{*amarathe,dkl*}*@cs.arizona.edu*

Zheng Gu, Matthew Small, and Xin Yuan
Department of Computer Science
Florida State University
Tallahassee, FL
{*zgu,small,xyuan*}*@cs.fsu.edu*

***Abstract*—Improving communication performance is critical to achieving high performance in message-passing programs. Designing new, efficient protocols to realize point-to-point and collective communication operations has therefore been an active area of research. However, the best protocol for a given communication routine is both application and architecture specific.**

This paper contributes a new method of selection of the optimal protocol for a given point-to-point communication pair. Our technique analyzes the MPI communication call profile of an application and uses a computation and communication model we have developed to choose the proper protocol for each communication phase. We have applied our system to MPI applications such as CG, Sweep3D and Sparse Matrix multiplication, as well as synthetic applications. Our scheme yields an improvement in total execution time of up to 20% compared to MVAPICH2 and up to 3.2% compared to the best, highly optimized communication protocol for the real applications. Furthermore, experiments on the synthetic applications show that the savings can be much more pronounced.

***Keywords*-MPI; point-to-point communication; protocol selection**

I. Introduction

Message Passing Interface (MPI) is the de-facto standard for developing distributed applications. Scientific MPI applications spend a significant amount of time executing communication calls. The problem of reducing the total execution times of MPI communication routines in scalable distributed applications has been a key area of research.

The MPI standard provides two general types of communication operations: point-to-point and collectives. This paper focuses on improving the execution times of point-to-point communication calls. A point-to-point communication operation can be realized using many different low-level protocols. For a given MPI application, different protocols will result in different execution times. Hence, the choice of optimal implementation is application specific.

In this paper, we improve the execution time of point-to-point communication calls in an application by using its communication call profile. From this profile, we use a model of computation and communication to predict the best protocol from a set of many potential protocols that can be used to implement the communication. Notably, we allow different protocols for different communication pairs.

This paper makes three *contributions*. First, we develop a model to predict the cost of each available protocol for the supported point-to-point communication operations. The model takes into account the message size, the function call arrival times and the effect of each protocol on the computation operations. Second, we present a system that analyzes the communication call profile of the application and predicts the optimal communication protocol based on the modeled execution times for each protocol. The execution times are modeled based on observed computation times between communication calls, wait times for the task in each communication pair and remote data copy times. Finally, based on our experience, we develop a novel point-to-point communication protocol.

We apply our system to real scientific and synthetic programs. The former group consists of CG, Sparse Matrix multiplication and Sweep3D. The synthetic programs are used to exercise all aspects of our system. We consider a large number of efficient MPI point-to-point protocols for each communication pair, and our results show that our system yields an improvement in total execution time up to 20% compared to the MVAPICH2[3] implementation, which itself is efficient. Compared to the best, highly optimized communication protocol for a given application, our technique saves over 14% for synthetic applications and up to 3.2% for our three real applications. While the savings depend on application characteristics, they can be quite pronounced.

The rest of the paper is organized as follows. We provide an overview of the related work in Section II. The implementation of the new protocol and the system is described in Section III. Next, Section IV discusses the measured results on a real cluster. Finally, Section V summarizes and describes future work.

II. Related Work

The existing approaches to improving the execution time of point-to-point MPI communication routines are static, and the selection of the protocol is carried out at compile time. Each available communication protocol is designed to execute different point-to-point communication operations with different execution times. The choice of protocol is based on the buffer size passed to the MPI communication

operation. An eager protocol is used for small messages, while a rendezvous protocol is used for large messages.

MPI communication over RDMA has been successfully implemented [5]. The use of RDMA over InfiniBand for MPI communication has been widely accepted as the preferred method for remote data copy operations due to several advantages previously studied [4]. The initiating task performs a copy of the remote buffer into the local buffer. The remote data copy operation happens without the intervention of the remote CPU, thus providing the opportunity for computation-communication overlap.

Recent work [9] introduces a new protocol for point-to-point communication over RDMA for certain message sizes. The protocol decouples sender from receiver by making a local copy of the user buffer and notifies the receiver about the address of the buffer. This reduces the execution time of the send operation. Upon arrival, the receiver performs a remote copy of the buffer over RDMA and notifies the sender of completion.

Yuan et al. refine the traditional rendezvous protocol over RDMA [9] presented by others [10], [7], [8] through two protocols they denote *receiver initiated* and *sender initiated*. In the case of receiver initiated rendezvous protocol, the receiver notifies the sender of the user buffer's address. When the sender initiates the send operation, it performs a RDMA write operation on the user buffer. Once the copy operation is complete, the sender notifies the receiver of its completion. In the case of the sender- initiated protocol, the sender notifies the receiver of the address of the user buffer to be sent. Upon arrival, the receiver performs a RDMA read operation on the sender's buffer to transfer the data. A speculative and adaptive approach for MPI rendezvous over RDMA was proposed [8] in which the task that arrives early chooses the communication protocol.

Static protocol decision schemes suffer from the lack of knowledge of (1) the critical path of the application, (2) the function call arrival times; and (3) the call sequence. The global critical path of the application follows the operations that affect the total execution time of the application. The choice of a communication protocol affects the execution time of the operations on the critical path. Each protocol is designed to exploit specific communication call arrival patterns to minimize the communication time resulting in improved execution time of the application. These difficulties can be overcome by collecting and analyzing the communication call trace of the application.

Profile guided optimization of MPI programs has been a widely researched topic [1], [2], [14], [11], [13]. Our work dynamically collects MPI application profiles, analyzes the application behavior, and proposes ways to improve the execution time of applications. The work by Venkata et al [12] on optimizing the MPI communication protocols is closely related to our work. The effort is aimed at switching the open RDMA connections between the communicating tasks. Our work differs from this work in two ways. First, our system collects and analyzes the application profile dynamically. Second, the optimal protocol is chosen based on the modeled execution times of the communication phase.

In summary, efforts to handle different MPI function call arrival sequences and message sizes have been handled statically. In contrast, our work models the execution time for a communication phase of a given application dynamically. Based on the modeled execution time, our system predicts the optimal protocol for the communication phase.

III. Profile Guided Protocol Selection

This section presents implementation details of our system. First, we discuss protocols we consider, including our pre-copy protocol [9]. Then we discuss the novel parts of our system, including a new sender-initiated post-copy protocol (the dual of our pre-copy protocol), our protocol execution models, and our online profile-based protocol decision system. We model the protocols based on message size and system-dependent factors. Finally, we describe our profile generation scheme and protocol decision mechanism.

A. Protocols

This section describes the protocols we consider in implementing MPI operations.

Pre-Copy Protocol: The sender-side pre-copy protocol presented by Yuan et al. [9] de-couples the sender from the receiver by making a local copy of the incoming buffer. The local buffer is RDMA-registered during `MPI_Init` to save the registration cost for every send call. The sender notifies the receiver of the address of the copied buffer and marks the completion of the send operation. The receiver registers the destination user buffer for RDMA operation upon arrival at the receive call. It then performs an RDMA read operation and a remote buffer copy. Figure 1 (left) shows the scenario in which the receiver arrives at the receive call after the sender arrives at the send call. Upon completion of the RDMA operation, the receiver notifies the sender and the sender's buffer is marked for release.

The cost of RDMA remote buffer copy operation is about four times the cost of local buffer copy operation for a given message size. As shown in Figure 1 (left), when the receiver arrives late at the receive call, it spends at least four times the time spent by the sender at the send call. Consider a communication pattern in which the receiver is always on the critical path and the receive operation is synchronous. In this case, the pre-copy protocol puts the RDMA operation on the critical path and increases the total execution time of the application.

Post-Copy Protocol: We developed *post-copy*, a novel protocol where the receiver spends the least time at the receive call when the sender is early. Upon arrival, the sender performs an RDMA remote buffer copy operation on a buffer that is pre-allocated at the receiver task. The size of the preallocated buffer is configured at startup based

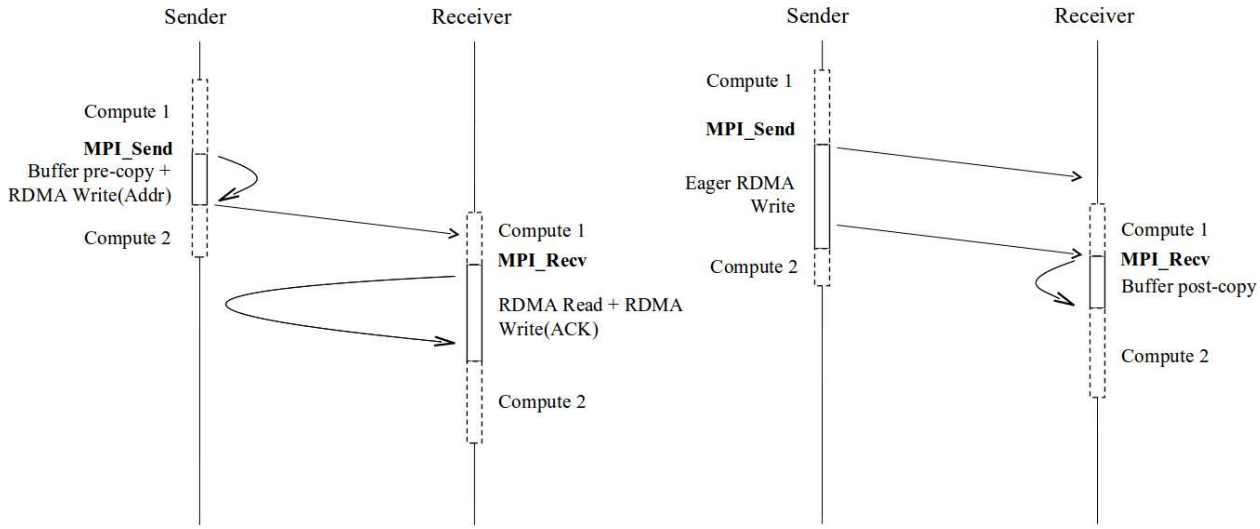

Figure 1: Communication with the pre-copy protocol (left) and the post-copy protocol (right).

on the application's communication pattern and memory requirements. The sender maintains the usage for the receiver's user buffer and then marks the completion of the send call.

Figure 1 (right) shows the scenario in which the receiver arrives late at the receive call. The receiver performs a local copy of the buffer upon arrival when the buffer has been completely written, and then the buffer is released. Unlike the sender pre-copy protocol, the expensive RDMA operation is performed by the sender. This provides a counterbalancing protocol for pre-copy protocol.

Other Protocols: We consider two other protocols in our model: receiver-initiated and sender-initiated rendezvous. These are described in detail elsewhere [9]. The essential details are as follows. For message sizes larger than pre-copy threshold, rendezvous protocols are used to negotiate transfer of the user buffer between sides. Both the sender and the receiver register the user buffers for the RDMA operation upon arrival. In case of the receiver-initiated rendezvous protocol, the receiver notifies the sender of the address of the user buffer upon arrival. The sender performs an RDMA write operation on the remote user buffer and notifies the receiver upon completion. Similarly, in case of the sender-initiated protocol, the sender notifies the receiver of the address of the user buffer; the receiver performs an RDMA read on the remote user buffer and then notifies the sender upon completion.

B. Modeling Protocol Execution Time

We model the four protocols based on both system-independent and system-dependent parameters as well as the type of MPI call. The MPI point-to-point communication calls consist of three basic operations: `MPI_Isend`, `MPI_Irecv` and `MPI_Wait`. The synchronous operations are carried out by the corresponding asynchronous operation followed by `MPI_Wait`. All depend on the following system-dependent factors. First, denote $t_{rdma\_read}(b)$ and $t_{rdma\_write}(b)$ as the time required to complete RDMA read and write operations, respectively, for a message of size b. Also, we denote $t_{memreg}(b)$ as the time to register buffer b for an RDMA operation and $t_{memcopy}(b)$ as the time to make a local copy of the buffer. Also, three system-dependent constants are used: t_{func_delay}, t_{rdma_async}, $t_{rdma\_roundtrip}$; these are the times spent in the MPI calls to initiate an RDMA write operation and the time for the RDMA write to complete.

The models also depend on several system independent factors. These include t_{send_enter}, t_{send_leave}, t_{recv_enter}, and t_{recv_leave}, as the time at which the sender (receiver) enters (leaves) the send (receive) operation.

Figure 2 (left) shows modeled times for the post-copy protocol when the sender is early. Upon arrival, the sender registers the user buffer in time t_{memreg}. It spends t_{rdma_write} time to perform an RDMA write into the receiver's remote buffer. The modeled time spent in the rest of the send function is denoted by t_{func_delay}. Therefore, t(`MPI_Isend`) = t_{memreg} + t_{rdma_write} + t_{func_delay}. At the receiver, the receive operation spends $t_{memcopy}$ time to perform a local copy into the user buffer, thus t(`MPI_Irecv`) = $t_{memcopy}$ + t_{func_delay}. There is no time in `MPI_Wait` apart from t_{func_delay}, so t(`MPI_Wait`$_{send}$) = t_{func_delay} and t(`MPI_Wait`$_{recv}$) = t_{func_delay}.

Figure 2 (right) shows the case in which the receiver is early. In this case, the `MPI_Wait` operation at the receiver waits for the sender to copy the remote buffer for t_{rdma_write} time starting from t_{send_enter}. The total time spent in `MPI_Wait` is calculated based on time t_{enter} when `MPI_Wait` is called and time $t_{memcopy}$ to perform the local copy operation. So while t(`MPI_Isend`) and t(`MPI_Wait`$_{send}$) are identical to the previous case, we now have t(`MPI_Irecv`) = t_{func_delay} and t(`MPI_Wait`$_{recv}$)

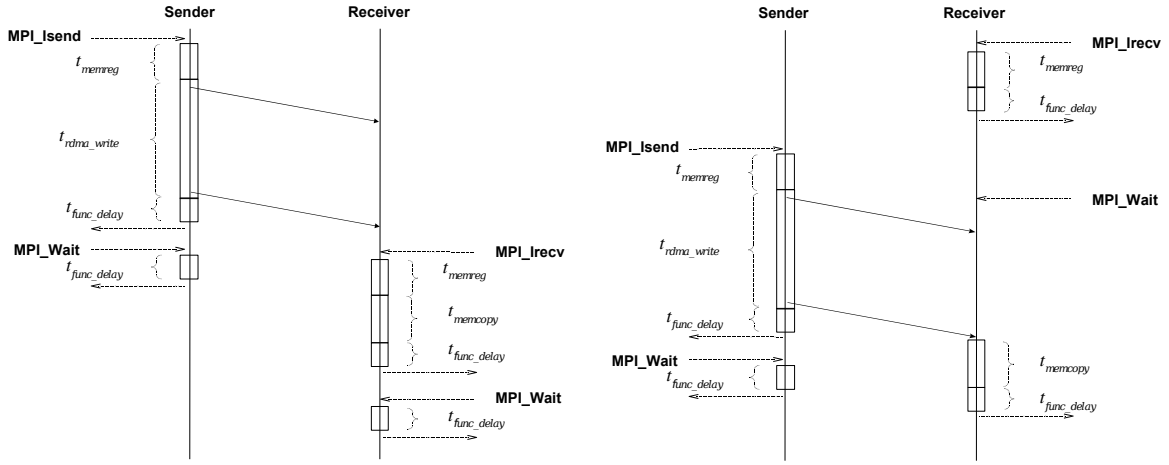

Figure 2: Execution model for post-copy when either sender (left) or receiver (right) is early.

$= t_{send\_enter} + t_{memreg} + t_{rdma\_write} + t_{memcopy} + t_{func\_delay}$.

Figure 3 shows the execution model for pre-copy when the sender arrives early. Figure 4 shows the execution models for receiver-initiated and sender-initiated protocols when the receiver arrives early.

C. Online Protocol Selection Algorithm

This section describes how we dynamically select the best protocol for each communication pair in a phase. First, the runtime system collects an MPI communication call profile for each task of the application with the legacy (MVAPICH2) implementation of the communication operations.

The profile contains the following information: communication call arrival times, communication pattern, usage of the user buffer by subsequent MPI calls, and time spent in the computation operations between MPI calls for pre-copy and post-copy protocols. Our system collects the communication call arrival times by recording the arrival time relative to the arrival of `MPI_Init` call. We define a *communication phase* as a sequence of communication calls delineated by the programmer using `MPI_Pcontrol` calls. The execution times are calculated for the matching MPI operations in one communication phase. The buffer usage information for the user buffers is collected by protecting the buffers for read and write accesses. Specifically, the system profiles the MPI calls following an `MPI_Send` call until the user buffer is modified. Similarly, the MPI calls following an `MPI_Recv` call are profiled until the buffer is accessed.

Armed with the information above, at each MPI point-to-point call, the run time system models the execution of the ongoing MPI call to predict the execution time. The modeling occurs at each task using collected arrival times for the participating tasks. The predicted execution time is then piggy-backed on the task's subsequent MPI call.

At the end of two iterations, we collect the total execution times for each task for each protocol. The optimal protocol for the communication phase is chosen using our model, basing the overall run time on the longest-running task.

D. Converting Synchronous Calls to Asynchronous Calls

A communication operation can be overlapped with a computation operation or another communication operation if the two operations do not share a common user buffer. Previous work by Preissl et al.[6] showed that certain synchronous MPI communication calls can be converted into corresponding asynchronous calls by analyzing the MPI call sequence. The source code is modified so that the asynchronous calls that use the user buffer are placed ahead of the computation operations that do not touch the buffer. This enables a communication operation to be executed with an overlapping computation operation, resulting in speedup for the application. However, such a scheme needs the capability of analyzing and modifying the application source code. We have developed a novel way of identifying the opportunity of communication overlap and converting synchronous calls into asynchronous calls without source code transformations.

The hybrid approach based on usage of the buffer passed to the communication routine is as follows. If the buffer used by `MPI_Send` has more than one MPI call in the profile, the corresponding `MPI_Isend` is called with the buffer. The last MPI call in the list of MPI calls associated with the user buffer waits for the completion of the asynchronous communication operation. To handle the situation where an operation attempts to modify the user buffer before it is transferred, the buffer is write protected. If invoked, the handler for the write access waits for the transfer of the buffer before it returns to the faulting operation. `MPI_Recv` is handled in a similar fashion. The Online Protocol Selection algorithm incorporates the mechanism by modeling asynchronous calls for the corresponding synchronous calls. The mechanism can be enabled by placing *MPI_Pcontrol* calls at the start of the communication phase.

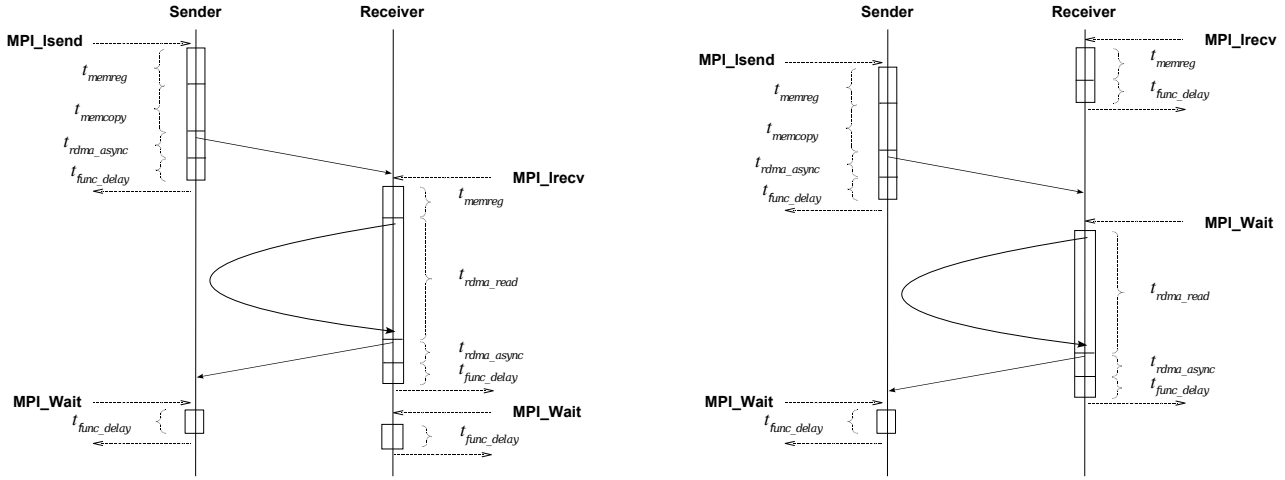

Figure 3: Execution models for pre-copy when the sender is early (left) and the receiver is early (right).

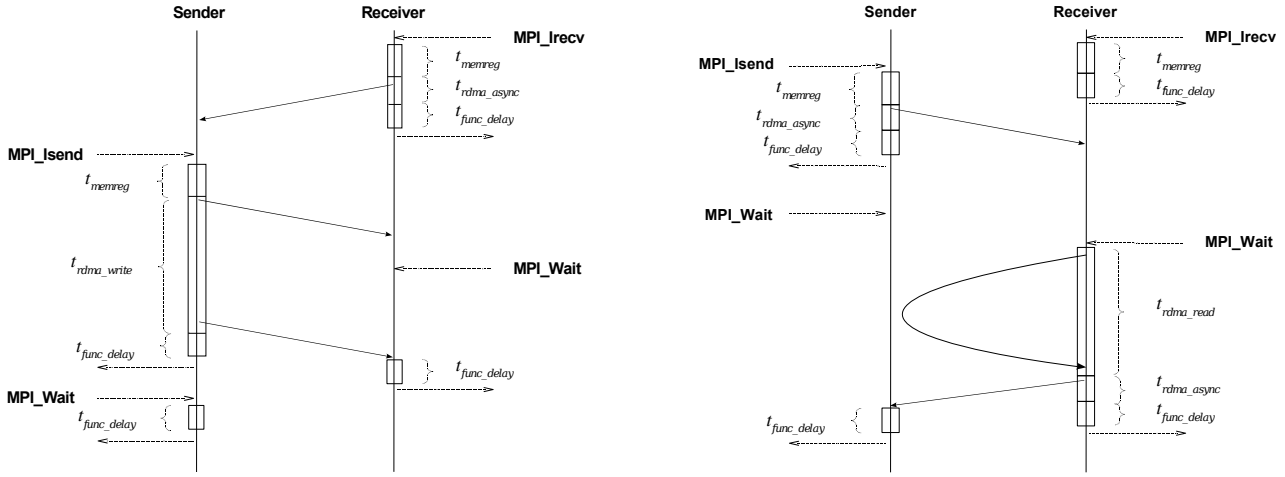

Figure 4: Execution models for receiver- and sender-initiated protocols when the receiver is early.

IV. Performance Evaluation

We tested our system on a cluster with 16 nodes each with 64 bit 8-core 2.33 GHz Intel Xeon processors and 8 GB of system memory. The nodes are interconnected with InfiniBand fabric. To evaluate our system for a combination of multiple protocols, we used standard MPI benchmarks along with our own microbenchmarks that combine multiple communication phases, each optimal for one protocol.

Our standard MPI benchmarks were Sparse Matrix, CG and Sweep3d. Figure 5 (left) shows a comparison of the application runs for the four protocols with MVAPICH2. Each application has a characteristic communication phase that is repeatedly executed by the application. We require the programmer to place *MPI_Pcontrol*(n) calls to demarcate the start and end of communication phase n. Based on this input, our system applies the protocol models and chooses the optimal protocol for the communication phase. Because each real MPI benchmark has a single communication phase that is executed repeatedly, our system outputs a single optimal protocol for each application.

We indicated the repeating communication phases in Sparse Matrix, CG and Sweep3D before running the applications. For the Sparse Matrix application, our system chose the post-copy protocol as optimal. For 16 tasks the total execution time is 11% better than MVAPICH2. In the case of CG, the system chose the pre-copy protocol as optimal. For 16 tasks, the total execution time is 19.8% better than MVAPICH2. With Sweep3D, the modeled execution times for each protocol were similar and the system chose receiver-initiated protocol as optimal. For 16 tasks, it showed 2.9% improvement in total execution time over MVAPICH2. The execution times for the real benchmarks indicate that the system was dynamically able to choose the optimal protocol.

Figure 5 (right) presents the normalized execution times for the microbenchmarks compared to MVAPICH2. The first microbenchmark consists of two communication phases, one of which executes best with the receiver- initiated protocol and the other with the pre-copy protocol. Our system is able

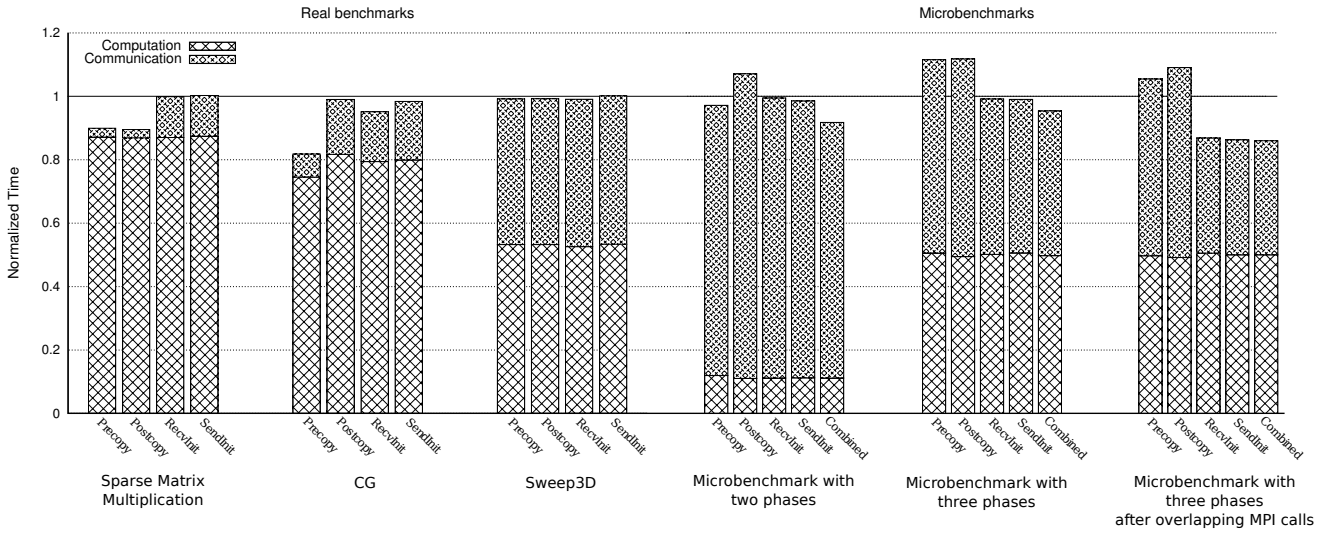

Figure 5: Normalized execution times for real benchmarks (left) and (right) microbenchmarks.

to find this combination automatically and results in an 8.2% improvement in total execution time over MVAPICH2.

Next, we evaluate the scheme for conversion of synchronous calls to asynchronous calls. This microbenchmark consists of three communication phases, each suitable for pre-copy, post-copy and receiver-initiated protocols, respectively. Here, our system chooses the optimal protocol for each communication phase and results in a 4.9% improvement compared to MVAPICH2.

The rightmost graph shows the normalized execution times after enabling the mechanism for the conversion of synchronous MPI calls to asynchronous MPI calls. Due to the overlap of multiple MPI calls during each phase, the system chooses precopy for the first phase and receiver-initiated protocol for the last two phases. This results in a 14% improvement in the total execution time of the application.

Finally, we observe that switching the protocol affects the execution times of the computation routines. Specifically, in case of the microbenchmarks and CG, we observed that the computation routine between two *MPI_Send* calls showed a twofold speedup when we switched the protocol from receiver-initiated to pre-copy. With pre-copy, *MPI_Send* call prepares a local copy of the user buffer and places the pages of the buffer into the data cache. The subsequent computation operation writes into the user buffer while the pages are in the cache. This speeds up the computation operation due to fewer cache misses. We observed a similar effect for the computation after an *MPI_Recv* at the receiver in the case of post-copy. For the rest of the protocols, the message transfer happens over RDMA and does not result in placing the buffer pages in the cache. To capture this effect, we perform explicit local buffer copy operations of the incoming and outgoing buffers at the sender and the receiver, respectively. We perform a memory copy in the PMPI layer for the first iteration of MVAPICH2 execution of each communication phase. The execution times for the computation operations collected during profile generation are used by the system to execute the protocol models.

Table I shows the accuracy of the online protocol modeling scheme by comparing the real and modeled execution times of Sparse Matrix application for 4 tasks. For the available protocols, the accuracy of the model is 94% to 99%.

Table II lists the overhead and total execution times for Sparse Matrix application. The overhead scales up linearly with the number of tasks. In comparison to the total execution times of the real MPI benchmarks, the observed overhead is negligible.

V. Summary

Our system successfully analyzes the execution profile of a given application and predicts the optimal protocol for each communication phase. The evaluation shows that a profile-based protocol decision can be effectively carried out dynamically with minimal overhead. This eliminates the need to execute the *entire* application once to collect the profile and re-execute the application with the optimal protocols. The sender-initiated post-copy protocol provides an alternative to the pre-copy protocol. As future work, we plan to apply our system to large-scale scientific MPI applications.

References

[1] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *20th International Conference on Supercomputing*, 2006.

[2] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of SPMD codes using expectations. In *21st International Conference on Supercomputing*, 2007.

	Precopy (sec)		Postcopy (sec)		Receiver Initiated (sec)		Sender Initiated (sec)	
	Real	Modeled	Real	Modeled	Real	Modeled	Real	Modeled
Task 0	89.675	87.104	86.324	86.648	95.115	92.668	94.252	92.759
Task 1	49.885	46.968	47.438	46.289	54.746	53.044	53.885	53.136
Task 2	50.938	48.238	47.896	47.213	54.741	54.031	54.752	54.123
Task 3	95.191	92.543	92.386	91.766	99.334	98.555	99.257	98.647

Table I: Real and modeled execution times for each task of Sparse Matrix application

No. of Tasks	Overhead (sec)	Total Execution Time (sec)
4	0.0016	92.703897
9	0.00283	80.526248
16	0.02598	72.904912
25	0.02845	76.507838
49	0.02628	68.059476
64	0.03786	59.827659
81	0.07361	70.621333
100	0.11884	62.050093

Table II: Overhead of the protocol modeling scheme

[3] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda. Design of high performance MVAPICH2: MPI2 over InfiniBand. In *6th IEEE International Symposium on Cluster Computing and the Grid*, 2006.

[4] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. Panda. Performance comparison of MPI implementations over InfiniBand, myrinet and quadrics. In *ACM/IEEE Supercomputing*, 2003.

[5] J. Liu, J. Wu, S. Kini, P. Wyckoff, and D. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *17th International Conference on Supercomputing*, 2003.

[6] R. Preissl, M. Schulz, D. Kranzlm, B. Supinski, and D. Quinlan. Using MPI communication patterns to guide source code transformations. In *8th International Conference on Computational Science, Part III*. 2008.

[7] M. Rashti and A. Afsahi. Improving communication progress and overlap in MPI rendezvous protocol over RDMA-enabled interconnects. In *22nd Intl. Symp. on High Performance Computing Systems and Applications*, June 2008.

[8] M. Rashti and A. Afsahi. A speculative and adaptive MPI rendezvous protocol over RDMA-enabled interconnects. *International Journal of Parallel Programming*, 37:223–246, April 2009.

[9] M. Small and X. Yuan. Maximizing MPI point-to-point communication performance on RDMA-enabled clusters with customized protocols. In *23rd International Conference on Supercomputing*, 2009.

[10] S. Sur, H. Jin, L. Chai, and D. K. Panda. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In *11th ACM Symposium on Principles and Practice of Parallel Programming*, 2006.

[11] R. Susukita, H. Ando, M. Aoyagi, H. Honda, Y. Inadomi, K. Inoue, S. Ishizuki, Y. Kimura, H. Komatsu, M. Kurokawa, K. Murakami, H. Shibamura, S. Yamamura, and Y. Yu. Performance prediction of large-scale parallel system and application using macro-level simulation. In *ACM/IEEE Supercomputing*, 2008.

[12] M. Venkata, P. Bridges, and P. Widener. Using application communication characteristics to drive dynamic MPI reconfiguration. In *IEEE International Symposium on Parallel and Distributed Processing*, 2009.

[13] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: a performance framework for distributed parallel systems. In *ACM/IEEE Supercomputing*, 2000.

[14] J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng. FACT: fast communication trace collection for parallel applications through program slicing. In *ACM/IEEE Supercomputing*, 2009.