Near-optimal Rendezvous Protocols for RDMA-enabled Clusters

Matthew Small Zheng Gu Xin Yuan
Department of Computer Science, Florida State University
Tallahassee, FL 32306
{small,zgu,xyuan}@cs.fsu.edu

Abstract—**Optimizing Message Passing Interface (MPI) point-to-point communication for large messages is of paramount importance since most communications in MPI applications are performed by such operations. Remote Direct Memory Access (RDMA) allows one-sided data transfer and provides great flexibility in the design of efficient communication protocols for large messages. However, achieving high performance on RDMA-enabled clusters is still challenging due to the complexity both in communication protocols and in protocol invocation scenarios. In this work, we investigate a profile-driven compiled-assisted protocol customization approach for efficient communication on RDMA-enabled clusters. We analyze existing protocols and show that they are not ideal in many situations. By leveraging the RDMA capability, we develop a set of protocols that can provide near-optimal performance for all protocol invocation scenarios, which allows protocol customization to achieve near-optimal performance when the appropriate protocol is used for each communication. Finally, we evaluate the potential benefits of protocol customization using micro-benchmarks and application benchmarks. The results demonstrate that the proposed protocols can out-perform traditional rendezvous protocols to a large degree in many situations and that protocol customization can significantly improve MPI communication performance.**

I. Introduction

Achieving high performance in Message Passing Interface (MPI) point-to-point communication for large messages is of paramount importance since most communications in MPI applications are performed by such operations. Traditionally, MPI point-to-point communications of large messages are realized by the *rendezvous* protocols, which avoid data copies in the library, but require the sender and the receiver to negotiate before the data are communicated.

Contemporary system area networks such as InfiniBand [5] and Myrinet [13] support Remote Direct Memory Access (RDMA) that allows one-sided direct data transfer. By allowing data transfer to be initiated by either the sender or the receiver, RDMA provides great flexibility in the design of communication protocols. Many efforts have been made to use the RDMA capability to improve the rendezvous protocols [16], [18], [19]. However, all existing protocols perform well in some situations, *but not all situations.*

Achieving high performance communication for large messages on RDMA-enabled clusters is challenging mainly for two related reasons. The first is the protocol complexity. Since the data size is large, copying data introduces significant overheads and should in general be avoided. Hence, all existing protocols are rendezvous protocols with multiple rounds of control messages, which can result in various problems such as unnecessary synchronizations and communication progress issues [3], [15], [16], [18], [19]. The second is the complexity in the protocol invocation scenario. MPI allows both the sender and the receiver to mark the times when a communication can start (e.g. `MPI_Isend/MPI_Irecv`) and when a communication must be completed (e.g. `MPI_Wait`). There are many combinations of the relative timing of these events, which can greatly affect the performance of a given protocol. We use the term *protocol invocation scenario* to denote the timing of the events in a communication. It is virtually impossible to design one scheme (even one that combines multiple protocols [16], [18]) that guarantees high performance for all cases.

In this work, we consider a profile-driven compiler-assisted *protocol customization* approach to maximize the performance for communicating large messages. Instead of using the same protocol for any protocol invocation scenario, our approach first identifies the protocol invocation scenario for each critical communication by analyzing the execution traces of an MPI program (program profile data) and/or by analyzing the program, and then chooses the most appropriate protocol for each scenario. We note that it might not be necessary to apply protocol customization for all communication routines in an application. Customizing the protocols in a small number of critical communication routines may yield significant improvement.

In order for profile-driven compiler-assisted protocol customization to be effective, (1) trace analysis and/or compiler analysis techniques must be developed to accurately determine protocol invocation scenarios; and (2) efficient communication protocols must be designed for all invocation scenarios. This paper focuses on the second item: obtaining efficient communication protocols. We analyze existing protocols for communicating large messages on RDMA-enabled clusters and show that they are not ideal in many situations. We develop a set of six protocols that can deliver near-optimal performance for **all** protocol invocation scenarios by leveraging the RDMA capability: when the protocol invocation scenario can be decided for a communication, one of the six protocols can be selected by the compiler or runtime system to achieve high performance. Finally, we implement all of the proposed protocols on InfiniBand and evaluate the potential benefits of protocol customization using micro-benchmarks and application benchmarks. The results indicate that protocol customization can significantly improve MPI communication performance.

The rest of the paper is organized as follows. Section II describes the related work. Section III discusses protocol invocation scenarios and analyzes existing rendezvous protocols. In Section IV, we present the six near-optimal protocols. Section V reports the experimental results. Finally, Section VI concludes the paper.

II. Related work

The performance issues with rendezvous protocols including unnecessary synchronizations, problems with communication progress, and limited opportunities for overlapping communication and computation, have been observed in many studies [1], [7], [15]. Various techniques have been developed to overcome these problems. The techniques can be broadly classified into three types: using interrupts to improve communication progress [1], [17], [19], using asynchronous communication progress to improve communication-computation overlaps [8], [9], [10], [12], [20], and improving the protocol design [3], [15], [16], [18], [19]. The interrupt driven message detection approach [1], [17], [19] allows each party (sender or receiver) to react to a message whenever the message arrives. The drawback is the non-negligible interrupt overhead. Asynchronous communication progress allows communications to be performed asynchronously with the main computation thread. This approach either needs a helper thread [10], [12], [20] or requires additional hardware support [8], [9]. Allowing communication and computation overlaps with a helper thread incurs performance penalties for synchronous communications. The third approach tries to improve the performance with better protocols, which can benefit both synchronous and asynchronous communications. It was shown that a sender-initiated RDMA read-based rendezvous protocol uses fewer control messages between the sender and the receiver than the RDMA write-based rendezvous protocol [19]. Pakin [15] showed that the receiver-initiated rendezvous protocol is simpler and can achieve higher performance in most cases than the sender-initiated protocol. Techniques that combine the sender-initiated and receiver-initiated protocols into one communication system have also been developed [16], [18]. These schemes, while more robust than other existing techniques, still cannot deliver high performance in some cases. The main problem with existing techniques is that all of them perform well in some situations, but not all situations: this motivates our research in using profile-driven compiler-assisted protocol customization that allows using different protocols in different situations to achieve the best performance. The closest work to this research is the Gravel library [3]. Although Gravel can be used for protocol customization, it supports a very limited set of protocols for communicating large messages.

III. Protocol invocation scenarios and existing rendezvous protocols

Each rendezvous protocol requires the exchange of one or more control messages. The control messages introduce implicit synchronizations between the sender and the receiver: when a protocol requires that a party P_1 responds a control message from the other side, P_1 cannot make progress unless the other party has sent the control message. Due to the implicit synchronizations, the performance of a rendezvous protocol can be significantly affected by the protocol invocation scenario, i.e., the timing of the communication related events in the sender and the receiver. Next, we will first discuss protocol invocation scenarios and then show why existing rendezvous protocols cannot deliver high performance in many situations.

A. Protocol invocation scenarios

There are four critical events in each MPI point-to-point communication: (1) the time when the sender can start the communication, which corresponds to the `MPI_Isend` call at the sender side and will be denoted as SS, (2) the time when the sender must complete the communication, which corresponds to the `MPI_Wait` call at the sender side and will be denoted as SW, (3) the time when the receiver can start the communication, which corresponds to the `MPI_Irecv` call at the receiver side and will be denoted as RS, and (4) the time when the receiver must complete the communication, which corresponds to the `MPI_Wait` at the receiver side and will be denoted as RW. We will use the notations SS, SW, RS, and RW to denote both the events and the timing of the events. The sender may or may not have computations between SS and SW; and the receiver may or may not have computation between RS and RW. When there are computations at those points, it is desirable to overlap the communication with these computations. After SW, the sender is blocked and does not perform any useful work until the communication is completed at the sender side. Similarly, after RW, the receiver is blocked and does not perform any useful work until the communication is completed at the receiver side.

Let $A, B \in \{SS, SW, RS, RW\}$. We will use the notion $A \leq B$ to denote that event A happens before or at the same time as event B, $A = B$ to denote that event A happens at the same time as event B, and $A < B$ to denote that A happens before B. Ordering events in one process is trivial: clearly, we have $SS \leq SW$ and $RS \leq RW$. Note that SS and SW happens at the same time in a blocking send call (`MPI_Send`); and RS and RW happens at the same time in a blocking receive call (`MPI_Recv`). For events in two processes, the order is defined as follows. Let event A happens in process P_A, and event B happens in process P_B. $A < B$ if after A, P_A has time to deliver a control message to P_B before B. $A = B$ denotes the case when each party does not have time to deliver a control message to other party before the event in that party happens. Fig. 1 shows the ordering of events in two processes.

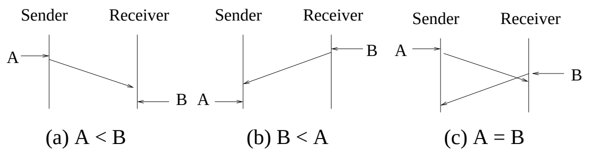

Fig. 1. The ordering of events in two nodes

Since $SS \leq SW$ and $RS \leq RW$, there are only six different orderings among the four events in a communication: $SS \leq SW \leq RS \leq RW$, $SS \leq RS \leq SW \leq RW$, $SS \leq RS \leq RW \leq RW$, $RS \leq RW \leq SS \leq SW$, $RS \leq SS \leq RW \leq SW$, and $RS \leq SS \leq SW \leq RW$. However, the ordering of the communication events is not the only factor that affects protocol design, the actual timing of the events also has an impact as will be shown in the Section IV.

B. Existing rendezvous protocols and their limitations

There are three existing rendezvous protocols developed for RDMA-enabled systems, the traditional sender-initiated RDMA write-based protocol [11], the sender-initiated RDMA read-based protocol [19], and the receiver-initiated protocol [15], [16]. We will briefly introduce each protocol and discuss their limitations. Ideally, in a rendezvous protocol, when both sender and receiver are ready for the communication, that is, both SS and RS happen, data transfer should start to maximize the overlap with the computations between SS and SW in the sender side and between RS and RW in the receiver side. None of these protocols can achieve this in all cases.

The sender-initiated RDMA write-based rendezvous protocol [11] is shown in Fig. 2 (a). In this protocol, the sender initiates the communication by sending a SENDER_READY packet; the receiver then responds with a RECEIVER_READY packet; after that the message data are RDMA written and a FIN packet is sent to indicate the completion of the communication. The sender must then wait for the communication of the data to complete before it can return from the operation.

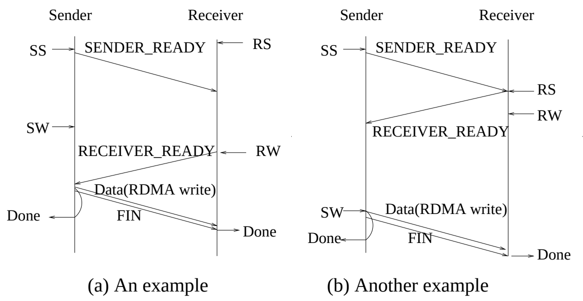

Fig. 2. Examples of sender-initiated RDMA write-based protocol with sub-optimal performance

Fig. 2 (a) and (b) show two examples that this protocol gives sub-optimal performance. In Fig. 2 (a), $SS = RS$ and since both parties for the rendezvous communication has arrived, ideally, data transfer should start within one control message time of SS and overlaps with the computation between SS and SW and between RS and RW. However, using this protocol, data transfer happens after RW with no communication-computation overlap (sender has been idling at SW, waiting to complete the protocol). In Fig. 2 (b), $SS < RS$ and the receiver responds to the SENDER_READY message right away. Again, ideally, data transfer should happen when both SS and RS happen (both sides are ready for the communication). However, since sender is in the computation when RECEIVER_READY arrives, the data transfer happens at a later time in SW: there is no overlap between computation and communication. Notice that the performance penalties for the inefficient protocol depend on the program structure: for Fig. 2 (a), the sender can idle in SW for a very long time depending on the amount of computation between RS and RW; for Fig. 2 (b), the receiver can idle in RW for a very long time depending on the amount of computation between SS and SW.

An example of the sender-initiated RDMA read-based protocol [19] is shown in Fig. 3 (a). In this protocol, the receiver responds to the SENDER_READY packet with a RDMA read operation. After the RDMA read operation is completed, the receiver sends a FIN packet to the sender and completes the operation. The sender exits the operation after it receives the FIN packet. In comparison to the RDMA write-based protocol, this protocol eliminates the RECEIVER_READY message, which may result in better communication progress [19]. However, this protocol also suffers from some limitations as shown in Fig. 3. In Fig. 3 (a), $SS = RS$. Using this protocol, data transfer happens at RW, which is not ideal. In Fig. 3 (b), $RS < SS$. With this protocol, the receiver does nothing at RS and data transfer still happens at RW.

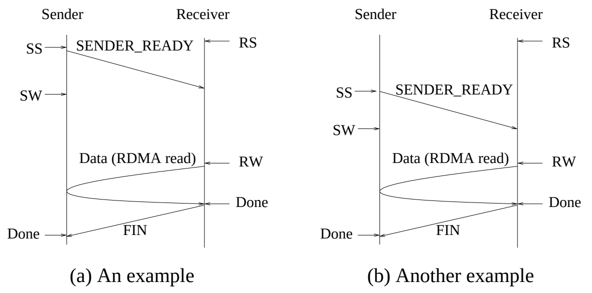

Fig. 3. Examples of sender-initiated RDMA read-based protocol with sub-optimal performance

An example of the receiver-initiated protocol [15] is shown in Fig. 4. In this protocol, the sender does nothing at SS if $SS < RS$. The receiver sends a RECEIVER_READY packet to the sender, which carries the receiving buffer information. When the sender gets this packet, it can directly deposit the data message into the receiver user space. As shown in Fig. 4, when $SS = RS$, the protocol is not ideal as the data transfer starts at SW.

There are other cases that all existing protocols can only give sub-optimal performance. Since none of the protocols is ideal in some cases (e.g. when $RS = SS$), the schemes [16], [18] that combine the sender-initiated protocol with the

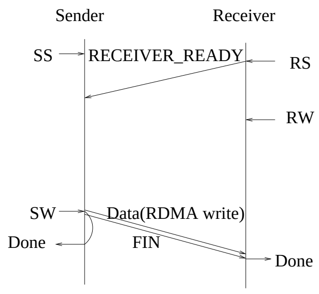

Fig. 4. Examples of receiver-initiated protocol with sub-optimal performance

receiver-initiated protocol are also not ideal in such cases. Hence, to effectively support profile-driven compiler-assisted protocol customization, new efficient protocols must be developed for the scenarios that existing protocols cannot perform well.

IV. Near-optimal communication protocols for large messages

In this section, we develop protocols for communicating large messages that can deliver near-optimal performance for all protocol invocation scenarios. We make the following assumptions:

- Data transfer cannot start unless both SS and RS happen. This is typical for sending large messages: both sides must be ready for the data to be communicated.
- The delay (cost) associated with sending and receiving a control message is negligible. The assumption is valid since the data message is large.
- RDMA read and RDMA write have similar performance.
- The sender can buffer the data message when necessary. Buffering at the sender side, even for large messages, is practical since it does not require the excessive per-pair buffers. However, buffering requires CPU time and thus, must be used with care. Hence, we further assume that buffering at the sender can only be performed when the sender is blocked.

Let $REND$ be the time when both SS and RS happen (the rendezvous time of the communication), $comm(msg)$ be the time to transfer the message with either RDMA write or RDMA read, and $copy(msg)$ be the time to make a local copy of the message. Under the above assumptions, an ideal communication scheme should have the following three properties.

- It should start the data transfer at the earliest time, which is $REND$. Starting the data transfer at the earliest time also maximizes communication-computation overlaps. It follows that both the sender and the receiver should complete the operation at $REND + comm(msg)$.
- When $REND \leq SW$, the sender should send the message at $REND$ and complete the operation at $REND + comm(msg)$.
- When $SW < REND$, the sender can buffer the data, use a control message to notify the receiver about the buffer, and return from the operation. The receiver can get the data from the buffer; and the buffer can be released in a later communication operation after the receiver gets the data. Thus, in this case, the sender should complete the operation at $SW + copy(msg)$

We note that this ideal communication scheme may not be optimal in that the communication completion times for the sender and the receiver may not be the earliest possible times in all cases. For example, we do not consider the concurrency of sending data and copying data simultaneously. Improvements can be made by exploiting such concurrency, as will be shown in one of our protocols. Hence, we will say that this ideal scheme is near-optimal. Our protocols are near-optimal in the sense that, ignoring the control message overheads, they have same three properties like the ideal scheme: our protocols have the same communication start times and completion times as the ideal communication scheme.

Next, we will present our protocols for all protocol invocation scenarios. We group all protocol invocation scenarios into three classes: $SS < RS$, $SS = RS$, and $RS < SS$. For a $SS < RS$ scenario, the sender arrives at the communication earlier than the receiver: the sender can notify the receiver that it is ready for the communication at SS and the receiver can get the notification at RS. Similarly, for a $RS < SS$ scenario, the receiver arrives at the communication earlier than the sender: the receiver can notify the sender that it is ready for the communication at RS and the sender can get the notification at SS. For a $SS = RS$ scenario, the sender and the receiver arrive at the communication at similar times: SS and RS are within one control message time.

Let us first consider the scenarios with $SS < RS$. The scenarios in this class are further partitioned into three cases with each case having a different protocol. The three cases are: $SS \leq SW < SW + copy(msg) < RS \leq RW$, $SS \leq SW < RS (\leq SW + copy(msg)) \leq RW$, and $SS < RS \leq \{SW\ and\ RW\}$. Here, $SW + copy(msg)$ is $copy(msg)$ time after SW. In the case $SS < RS \leq \{SW\ and\ RW\}$, SW and RW both happen no earlier than RS and the order between SW and RW does not matter.

Fig. 5 (a) shows the scenario for $SS \leq SW < SW + copy(msg) < RS \leq RW$, where SW is much earlier than RS. Our protocol for this case, shown in Fig. 5 (b), is called the *copy_get protocol*. In this protocol, the sender copies the message data to a local buffer at SW (this has no costs since sender is blocked for the communication and cannot do anything useful). After the data are copied, the sender issues a READY message, which contains the address of the local buffer and other related information to facilitate the RDMA read from the receiver, to the receiver. The task in the sender side is completed; and the sender can exit the operation. When the receiver gets the READY message, it performs a RDMA read to obtain the data from the sender buffer. The sender side buffer must be released at some point. Since this is a library buffer that the application will not access, the information for the receiver to notify the sender that the buffer can be released can be piggybacked in a later control message. The copy_get protocol leverages the RDMA capability and allows the sender to complete the communication before the receiver even arrives. For the sender, the operation completes at $SW + copy(msg)$. The data transfer starts at $REND = RS$ and the receiver completes the communication at $REND+comm(msg)$. Hence, this protocol is near-optimal for both the sender and the receiver.

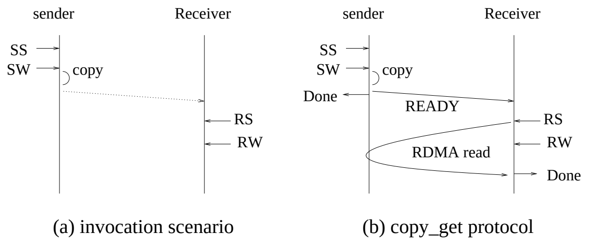

Fig. 5. $SS \leq SW < SW + copy(msg) < RS \leq RW$ scenario and the copy_get protocol

Fig. 6 (a) shows the scenario for $SS \leq SW < RS(\leq SW + copy(msg)) \leq RW$, where sender blocks ($SW$) slightly earlier than receiver arriving at the communication (RS) with not enough time to copy the whole message. Our protocol for this case, shown in Fig. 6 (b), is called the *copy_check_put protocol*. In this protocol, the sender sends a SENDER_READY message to receiver at SS. At SW, the sender starts copying the message data to a local buffer while monitoring control messages from the receiver. This can be implemented by repeatedly copying a small chunk of data and checking the message queue. Like in the copy_get protocol, these operations have no costs since the sender is blocked for the communication and cannot do anything useful. When the receiver arrives at RS, it will receive the SENDER_READY and send a RECEIVER_READY message, which should arrive at the sender before the sender finishes making a local copy. When the sender gets the RECEIVER_READY message, it sends partial data to the receiver while continuing to copy the message concurrently. We will assume that the system knows the copy and data transmission speeds and can determine the amount of data to be copied and to be transferred so that the combination of copied data and transferred data covers the whole message and that the (partial) data copy and (partial) data transfer complete at the same time. Note that this assumption can be approximated in practice. After that, the sender initiates the sending of the copied data and the FIN packet, and then returns from the communication. The copied data will be released in a later communication operation. For this protocol, the sender completes the operation before $SW+copy(msg)$ since it returns after the message is partially copied (initiating a communication does not take time). This is due to the concurrent sending and copying data in the protocol. The data transfer starts at $REND = RS$ and the receiver completes the operation at $REND + comm(msg)$. Hence, this protocol is near-optimal.

For $SS < RS \leq \{SW \text{ and } RW\}$ scenarios, the traditional sender-initiated RDMA read-based protocol is near-optimal.

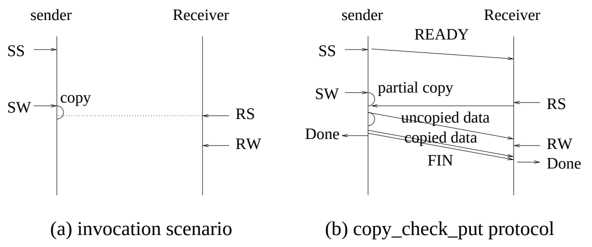

Fig. 6. $SS \leq SW < RS(\leq SW + copy(msg)) \leq RW$ scenario and the copy_check_put protocol

Using this protocol, data transfer starts at $REND = RS$; and both the sender and the receiver complete the communication at $REND + comm(msg)$.

Let us now consider the second class: $SS = RS$. This is one case when all existing rendezvous protocols are not ideal. However, if trace/profile/static analysis can draw the conclusion that SS and RS are within one control message time, the solution is straight-forward: waiting for the corresponding control message at SS or RS. We will call such protocols delayed sender-initiated protocols and delayed receiver-initiated protocols. Fig. 7 (a) shows a delayed sender-initiated RDMA read-based protocol. In this protocol, the receiver adds a delay time RS (marked as RS(begin) and RS(end) in Fig. 7 (a)). During the delay, the receiver repeatedly pulls the control message queue waiting for the SENDER_READY message to arrive at RS so that data transfer can start before the receiver leaves RS. Notice that the delay is less than one control message time and the communication starts within one control message time from $REND$. Hence, both the sender and receiver will complete the operation at $REND+comm(msg)$. Fig. 7 (b) shows the delayed receiver-initiated protocol where the delay is added to the sender at SS.

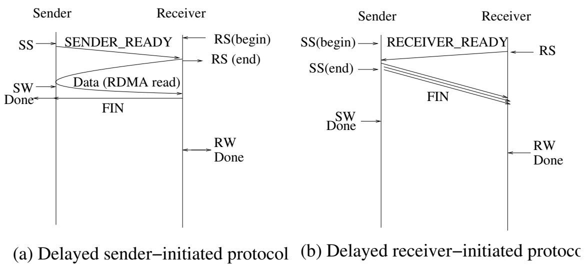

Fig. 7. Delayed rendezvous protocols for $SS = RS$ scenarios

Finally, for the third class where receiver arrives earlier than the sender ($RS < SS$), the traditional receiver-initiated protocol, as shown in Fig. 4, can achieve near-optimal performance. Data transfer starts exactly at $SS = REND$, which is the earliest time possible. Both sender and receiver will complete the operation at $REND+comm(msg)$, which is the same as the ideal communication scheme. Notice that when the receiver arrives at RS and RW much earlier than the sender, it must wait for the sender in order to complete the communication: there is an inherent synchronization from the sender to the receiver in every communication. As a result, the case when receiver arrives much earlier cannot be further optimized.

The six protocols that we discuss in this section, the copy_get protocol, the copy_check_put protocol, the sender-initiated RDMA read-based protocol, the delayed sender-initiated protocol, the delayed receiver-initiated protocol, and the receiver initiated protocol, should be able to achieve near-optimal performance for communicating large messages in any protocol invocation scenario. By combining these protocols with a profile driven static analysis scheme that identifies protocol invocation scenarios, protocol customization can potentially achieve near-optimal communication performance for all situations.

V. Performance study

In this section, we evaluate the performance of the proposed rendezvous protocols and study the potential benefits of protocol customization with micro-benchmarks and application benchmarks. We have implemented the six protocols discussed in the previous section over InfiniBand using the Verb API [6] in six versions of `MPI_Isend` , `MPI_Irecv` , `MPI_Send` , `MPI_Recv` , and `MPI_Wait` . Our library can co-exist with MVAPICH. MPI functions that are not supported by our library can be realized by MVAPICH. This allows us to compare the performance of MPI programs with point-to-point routines using our protocols to that with MVAPICH.

The evaluation is performed on an InfiniBand cluster that has 16 compute nodes with a total of 128 cores. Each node is a Dell Poweredge 1950 with two 2.33Ghz Quad-core Xeon E5345's (8 cores per node) and 8GB memory. All nodes run Linux with the 2.6.9-42.ELsmp kernel. The compute nodes are connected by a 20Gbps InfiniBand DDR switch (CISCO SFS 7000D). We compare the performance of protocol customization with that of the default MVAPICH2-1.2.rc1, which uses the sender-initiated RDMA write-based protocol to communicate large messages.

A. Micro-benchmark results

We use a micro-benchmark to evaluate the performance of different protocols with different protocol invocation scenarios. The micro-benchmark is shown in Fig. 8. In this benchmark, the time for 1000 iterations of the loop is measured. Inside the loop, a barrier is first called to synchronize the sender and the receiver. After that, the sender performs some computation $comp1$, calls *MPI_Isend* to start the send operation, performs some more computation $comp2$, calls *MPI_Wait* to complete the send operation, and performs some more computation $comp3$. Similarly, the receiver also performs some computation $comp4$ after the barrier, calls *MPI_Irecv* to start the receive operation, performs some more computation $comp5$, calls *MPI_Wait* to complete the receive operation, and performs some more computation $comp6$. The message size and the computation in between the communication routines are parameters. We will use the notation $(comp1, comp2, comp3, comp4, comp5, comp6)$ to represent the configuration of the benchmark, where $compX$ represents the duration of the computation (in the unit of a basic loop). For each computation, the larger the number is, the longer the computation lasts. By changing the values of the parameters, the benchmark can create all protocol invocation scenarios. In the discussion, we will use notation $C(compX)$ for the time for $compX$ computations, $T(msize)$ for the time to transfer a message of $msize$ bytes, and $copy(msize)$ for the time to copy a message of $msize$ bytes. In the experiment, $C(X)+C(Y) \approx C(X+Y)$, $C(1) \approx 18\mu s$ and $T(100KB) \approx 90\mu s$.

```
Process 1:            Process 2:
  Loop                  Loop:
    barrier()             barrier()
    comp1                 comp4
    MPI_Isend()           MPI_Irecv()
    comp2                 comp5
    MPI_Wait()            MPI_Wait()
    comp3                 comp6
```

Fig. 8. Micro-benchmark

We perform experiments using this micro-benchmark with different orderings of events and different relative timings. Protocol customization consistently achieves high performance. In the following, we will show the results for three representative cases. The first case has configuration $(1, 1, 48, 30, 19, 1)$, which emulates the case when SS and SW are much earlier than RS and RW as shown in Fig. 5 (a). The second case has configuration $(10, 30, 10, 10, 30, 10)$, which emulates the case when $SS = RS$ as shown in Fig. 2 (a). The third case has configuration $(10, 20, 20, 1, 40, 9)$, which emulates the case when $RS < SS$ as shown in Fig. 3 (b). Note that for all cases, both sender and receiver have a total of 50 units of computations, which translate to roughly $C(50) = 50 \times 18 = 900\ \mu s$ if both sides perform the computation concurrently.

Results for configuration $(1, 1, 48, 30, 19, 1)$ with different message sizes are shown in Fig. 9. Using the default rendezvous protocol in MVAPICH, there is an implicit synchronization from RS to SW, which results in the computation before RS (30 units) at the receiver and the computation after SW (48 units) at the sender to be sequentialized. Hence, the total time for each iteration is roughly $T(msize)+C(30+48)$. On the other hand, with protocol customization, the most effective protocol is the copy_get protocol in Fig. 5 (b), where the sender makes a local copy of the buffer and leaves the communication. With this protocol, the total time for each iteration is roughly $copy(msize) + C(50)$, which is much better than the result with the default MVAPICH as shown in Fig. 9. Notice that this is one scenario where no rendezvous protocol can perform well: copy_get is not a true rendezvous protocol since the sender leaves the communication before the receiver starts the communication. Notice also that copying data introduces significant overheads as shown in the upward slope for the curve for our scheme in Fig. 9.

Results for configuration $(10, 30, 10, 10, 30, 10)$ with differ-

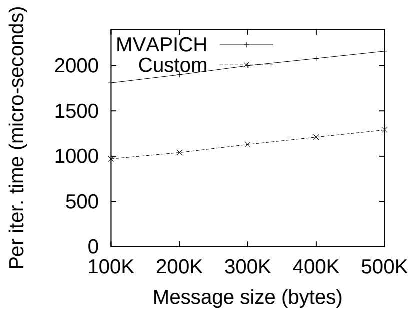

Fig. 9. Results for configuration $(1, 1, 48, 30, 19, 1)$

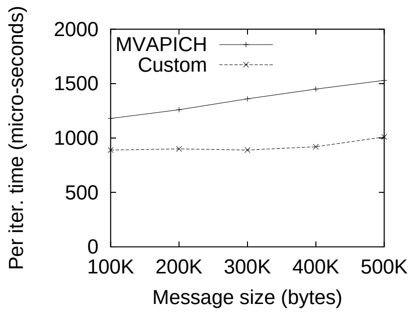

Fig. 11. Results for configuration $(10, 20, 20, 1, 40, 9)$

ent message sizes are shown in Fig. 10. This is the case when the READY messages from both sender and receiver pass each other and no existing protocol is ideal as discussed in Section III. With the default MVAPICH protocol, data transfer is performed at SW (and RW), and no communication-computation overlap is achieved. The per iteration time is thus roughly $T(msize) + C(50)$: the time increases linearly with the message size as shown in Fig. 10. The most effective protocol for this situation is the delayed receiver-initiated protocol shown in Fig. 7 (b), where the sender repeatedly polls the incoming message queue for the RECEIVER_READY message. Using this protocol, the communication can be completely overlapped with computations between SS and SW at the sender side and RS and RW at the receiver side; and the per iteration time is roughly $C(50)$, shown as a flat line in Fig. 10.

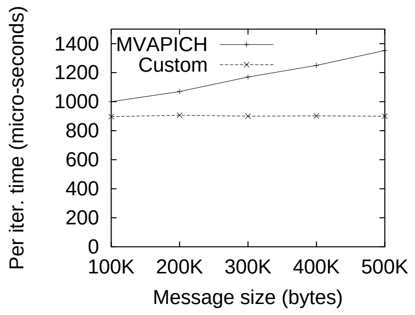

Fig. 10. Results for configuration $(10, 30, 10, 10, 30, 10)$

Results for configuration $(10, 20, 20, 1, 40, 9)$ with different message sizes are shown in Fig. 11. This case emulates the situation at Fig. 3 (b). With the default protocol, the communication starts at RW. Hence, the per iteration is roughly $C(41) + T(msize) + C(20) = C(61) + T(msize)$. Using the near-optimal receiver initiated protocol, the communication is overlapped completely with computation and the total time is roughly $C(50)$.

These results demonstrate that by using near-optimal protocols for different protocol invocation scenarios, protocol customization avoids the performance penalties due to the mismatch between the protocol and the protocol invocation scenarios and can achieve higher performance in comparison to traditional rendezvous protocols in many cases. Moreover, the improvement from protocol customization depends not only on the system communication performance, but also on program structures.

B. Application benchmark results

We use five application benchmarks to investigate the potential performance benefits of protocol customization. Three benchmarks are from the NAS parallel benchmarks [14]: BT, CG, and SP. We use the CLASS C problem size for all of the three programs. The other two programs are jacobi and sparsemm. The jacobi program uses Gauss-Siedel iterations to solve Laplace equations on a $8K \times 8K$ discretized unit square with Dirichlet boundary conditions. The sparsemm is a message passing implementation of the sparse SUMMA sparse matrix-matrix multiplication algorithm [2]. The program performs multiple times the self multiplication of a sparse matrix stored in file G3_circuit.mtx that is available in the University of Florida sparse matrix collection [4]. The matrix in G3_circuit.mtx is a sparse 1585478×1585478 matrix with 4623152 non-zero entries.

Since we have not developed trace and compiler analysis techniques to identify the most effective protocol for each communication, we are not able to thoroughly evaluate the benefit of protocol customization. Our methods to select protocols in this experiment, which will be described next, are preliminary and may not select the most efficient protocols for all communications. Hence, the results in this section only represent the potential improvement achievable through protocol customization. In the experiments, we do the following. We manually examine MPI program execution traces, which gives the timing of each MPI routine calls, and use the timing information to decide the protocol for each communication. This approach is not always effective since the relative timing of critical communication events may change when the communication protocol is changed. More sophisticated techniques that can select protocols more effectively are still under development. In addition to the trace-driven protocol customization, we also run each individual protocol for each of the programs. The reported performance results for protocol customization are the best communication times from both the trace-driven execution and the individual protocol execution.

Table I shows the total application times, total communica-

	MVAPICH		Customization		Comm.
	total (sec.)	comm. (sec.)	total (sec.)	comm. (sec.)	improv. percentage
BT	321.76	10.11	315.55	3.65	177.0%
CG	35.66	3.41	35.12	3.02	12.9%
SP	180.57	6.53	176.94	2.78	134.9%
sparsemm	16.35	12.75	10.51	6.92	84.2%
jacobi	282.94	2.88	282.33	2.35	22.6%

TABLE I
PERFORMANCE ON 16 PROCESSES (ONE PROCESS PER NODE)

	MVAPICH		Customization		Comm.
	total (sec.)	comm. (sec.)	total (sec.)	comm. (sec.)	improv. percentage
BT	99.90	21.00	98.52	16.45	27.7%
CG	16.45	6.87	16.16	6.60	4.1%
SP	56.28	13.77	55.70	12.12	13.6%
sparsemm	14.06	12.44	11.97	10.56	17.8%
jacobi	97.15	63.78	92.60	58.95	8.2%

TABLE II
PERFORMANCE ON 121/128 PROCESSES (8 PROCESSES PER NODE)

tion times, and the communication improvement percentages using our protocol customization scheme over MVAPICH for the programs running on 16 processes (one process per node). The communication time includes all Send, Isend, Recv, Irecv, and Wait times, which account for the majority of all communication times in these benchmarks. As can be seen from the table, protocol customization achieves significant improvement over MVAPICH for all the programs. The reason that protocol customization provides better performance for different programs are different. For BT, SP, and jacobi, the main reason is that protocol customization can explore the communication and computation overlapping opportunities better than the traditional protocol. For sparsemm, the main reason is the use of the copy-get protocol that eliminates the unnecessary synchronization from the sender to the receiver: the computation load is not balanced in this sparse matrix-matrix multiplication program and unnecessary synchronizations can introduce large waiting time, which is reduced with protocol customization. For CG, the performance gain is mainly from using a simpler receiver initiated protocol to carry out the communication. As can be seen from the table, although communication does not account for a large percentage of the total application time in BT, CG, SP, and jacobi, the improvement in communication times transfers into improvement of the total application time. For sparsemm, the total application time is also significantly improved since the communication time dominates this program.

Table II shows the results for the programs running on 121/128 processes (121 for BT and SP, 128 for CG, sparsemm, and jacobi) with 8 processes running on each node. One of the main difference between running one process per node and 8 processes per node is the intra-node communication. Since intra-node communication does not use the rendezvous protocol, our protocol customization is only applied to a portion of all communications (inter-node communications with large messages) in this experiment. This is the main reason that the communication improvement percentage is much lower for the 121/128 processes cases. However, as shown in the table, having a better inter-node communication mechanism with protocol customization still provides noticeable improvement for all the benchmarks.

VI. CONCLUSION

In this work, we show that existing protocols for handling large messages are not ideal in many cases and develop a set of protocols that can achieve near-optimal performance for any protocol invocation scenario. These protocols can be used in profile-driven compiler assisted protocol customization. Our preliminary evaluation with micro-benchmarks and application benchmarks demonstrates that protocol customization can significantly improve MPI communication performance. We are currently working on developing trace analysis techniques for automatic MPI protocol customization.

REFERENCES

[1] G. Amerson and A. Apon, “Implementation and Design analysis of a Network Messaging Module using Virtual Interface Architecture,” *International Conference on Cluster Computing*, 2004.
[2] A. Buluc and J.R. Gilbert, “Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication,” *International Conference on Parallel Processing*, 2008.
[3] A. Danalis, A. Brown, L. Pollock, M. Swany, and J. Cavazos, “Gravel: A Communication Library to Fast Path MPI,” *Euro PVM/MPI*, September 2008.
[4] Tim Davis, “The University of Florida Sparse Matrix Collection,” http://www.cise.ufl.edu/research/sparse/matrices/.
[5] InfiniBand Trade Association, http://www.infinibandta.org.
[6] “InfiniBand Host Channel Adapter Verb Implementer’s Guide”, Intel Corp., 2003.
[7] J. Ke, M. Burtscher, and E. Speight, “Tolerating Message Latency through the Early Release of Blocked Receives,” *Euro-Par 2005*, LNCS 2648, pp 19-29, 2005.
[8] C. Keppitiyagama and a. Wagner, “MPI-NP II: A Network Processor Based Message Manager for MPI,” *International Conference on Communications in Computing*, 2000.
[9] C. Keppitiyagama and a. Wagner, “Asynchronous MPI Messaging on Myrinet,” *IEEE International Parallel and Distributed Processing Symposium* (IPDPS), 2001.
[10] R. Kumar, A. Mamidala, M. Koop, G. Santhanaraman and D.K. Panda, “Lock-free Asynchronous Rendezvous Design for MPI Point-to-Point Communication,” *EuroPVM/MPI*, Sept. 2008.
[11] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, “High Performance RDMA-Based MPI Implementation over InfiniBand,” the *17th International Conference on Supercomputing* (ICS), pages 295-304, June 2003.
[12] S. Majumder, S. Rixner, and V. S. Pai, “An Event-Driven Architecture for MPI Libraries,” The *Los Alamos Computer Science Institute Symposium*, 2004.
[13] Myricom, http://www.myricom.com.
[14] NAS Parallel Benchmarks, http://www.nas.nasa.gov/Software/NPB/
[15] S. Pakin, “Receiver-initiated Message Passing over RDMA Networks,” the *22nd IEEE International Parallel and Distributed Processing Symposium* (IPDPS), April 2008.
[16] M. J. Rashti and A. Afsahi, “Improving Communication Progress and Overlap in MPI Rendezvous Protocol over RDMA-enabled Interconnects,” the *22nd High Performance Computing Symposium* (HPCS), June 2008.
[17] D. Sitsky and K. Hayashi, “An MPI Library Which Uses Polling, Interrupts, and Remote Copying for the Fujitsu AP1000+,” *International Symposium on Parallel Architectures, Algorithms, and networks*, 1996.

[18] M. Small and X. Yuan, “Maximizing MPI Point-to-point Communication Performance on RDMA-enabled Clusters with Customized Protocols,” the *23th ACM International Conference on Supercomputing (ICS)*, pages 306-315, June 2009.

[19] S. Sur, H. Jin, L. Chai, D. K. Panda, “RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits,” *ACM PPoPP*, pages 32-39, 2006.

[20] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D.K. Panda, “Host-Assisted Zero-Copy Remote Memory Access Communication on InfiniBand,” *IEEE International Parallel and Distributed Processing Symposium*, 2004.