Int J Parallel Prog (2013) 41:682–703
DOI 10.1007/s10766-013-0242-0

Protocol Customization for Improving MPI Performance on RDMA-Enabled Clusters

Zheng Gu · Matthew Small · Xin Yuan · Aniruddha Marathe · David K. Lowenthal

Received: 8 February 2012 / Accepted: 11 February 2013 / Published online: 28 February 2013

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 point-to-point communication performance on RDMA-enabled clusters is challenging due to both the complexity in communication protocols and the impact of the protocol invocation scenario on the performance of a given protocol. In this work, we analyze existing protocols and show that they are not ideal in many situations, and propose to use protocol customization, that is, different protocols for different situations to improve MPI performance. More specifically, by leveraging the RDMA capability, we develop a set of protocols that can provide high performance for **all** protocol invocation scenarios. Armed with this set of protocols that can collectively achieve high performance in all situations, we demonstrate the potential of protocol customization by developing a trace-driven toolkit that allows the appropriate protocol to be selected

Z. Gu · M. Small · X. Yuan (✉)
Department of Computer Science, Florida State University, Tallahassee, FL 32306, USA
e-mail: xyuan@cs.fsu.edu

Z. Gu
e-mail: zgu@cs.fsu.edu

M. Small
e-mail: small@cs.fsu.edu

A. Marathe · D. K. Lowenthal
Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA
e-mail: amarathe@cs.arizona.edu

D. K. Lowenthal
e-mail: dkl@cs.arizona.edu

for each communication in an MPI application to maximize performance. We evaluate the performance of the proposed techniques using micro-benchmarks and application benchmarks. The results indicate that protocol customization can out-perform traditional communication schemes by a large degree in many situations.

Keywords MPI · Point-to-point communication · Protocol customization

1 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 [22]. 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 data are communicated.

Contemporary system area networks such as InfiniBand [8] and Myrinet [18] 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. Much effort has been made to use the RDMA capability to improve the rendezvous protocols [23,25,27]. However, it is well known that existing protocols perform well in some situations, *but not all situations* [25].

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 [5,21,23,25,27]. The second is the significant impact of the protocol invocation scenario on the performance of a given protocol. 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. We use the term *protocol invocation scenario* to denote the timing of the events in a communication. As will be shown later, the protocol invocation scenario can significantly affect the performance of a given protocol; and it is virtually impossible to design one scheme (even one that combines multiple protocols [23,25]) that guarantees high performance for all scenarios.

In this work, we propose a *protocol customization* approach to overcome the limitations in traditional MPI libraries. Instead of using the same protocol for any protocol invocation scenario as in a traditional MPI library, protocol customization uses different protocols for different communications. The idea is to infer protocol invocation scenario by some means such as program analysis, runtime analysis, profiling, and trace analysis, and to select the most effective protocol for each communication based on the protocol invocation scenario. This work focuses on trace-driven protocol customization for deterministic communications where the information is obtained by trace analysis. Non-deterministic communications with MPI_ANY_SOURCE and

MPI_ANY_TAG are not considered in this work. In a recent work, we have applied a limited form of protocol customization at runtime [16].

In order for protocol customization to be effective, (1) efficient communication protocols must be designed for *all* invocation scenarios; and (2) techniques must be developed to accurately determine protocol invocation scenarios. We analyze existing protocols for communicating large messages on RDMA-enabled clusters and show that they are not ideal in many situations. There exist protocol invocation scenarios for which no existing protocol performs well. We develop a set of six protocols that can deliver high performance for **all** protocol invocation scenarios by leveraging the RDMA capability. These protocols form a foundation for protocol customization: one of the six protocols can be selected to achieve high performance for any given scenario. Armed with this set of protocols, we demonstrate the effectiveness of protocol customization by developing a trace-driven protocol customization toolkit that automatically selects the appropriate protocol for each communication in an MPI application. We implement the set of six protocols and the trace-driven protocol customization toolkit on InfiniBand and evaluate the performance using micro-benchmarks and application benchmarks. The results indicate that protocol customization can significantly improve MPI communication performance in many situations.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 outlines protocol invocation scenarios and analyzes existing rendezvous protocols. In Sect. 4, we present the six customized rendezvous protocols. In Sect. 5, we present our trace-driven protocol customization toolkit. Section 6 reports the experimental results. Finally, Sect. 7 concludes the paper.

2 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 [1,11,21]. Various techniques have been developed to overcome these problems. The techniques can be broadly classified into three types: using interrupts to improve communication progress [1,24,27], using asynchronous communication progress to improve communication–computation overlaps [12,13,15,28], and improving the protocol design [5,21,23,25,27]. The interrupt driven message detection approach [1,24,27] 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 [13,15,28] or requires additional hardware support [12]. 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 [21,23,25,27].

All existing schemes perform well in some situations, but not all situations. The closest work to this research is the Gravel library [5] that supports limited protocol

customization with compiler and/or user support. Protocols in Gravel cannot provide high performance for all scenarios while our protocols cover all cases. Another closely related work is the dynamic MPI reconfiguration with application communication characteristics [29]. In this work, a profiler is used to obtain application communication information, which is used to reconfigure the underlying byte-transfer layer in Open MPI. Although this technique also reconfigures protocols based on application information, the protocols were not designed to deal with different protocol invocation scenarios. As a result, the techniques are very different from the one proposed in our work. For example, the technique in [29] is not concerned with the relative timing of communication events, which is the focus of our work. The work [7] uses timing of communication events to optimize collective communications. This work applies to a similar approach to point-to-point communications.

3 Protocol Invocation Scenarios and Rendezvous Protocols

3.1 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` (or `MPI_Waitall`) 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. Notations SS, SW, RS, and RW will be used 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. Note that high-end systems are usually equipped with communication co-processors that allow communication and computation to progress simultaneously [3]. After SW, the sender is blocked and does not perform any useful work until the communication is completed at the sender side. Similar, 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 happen at the same time in a blocking send call (`MPI_Send`); [1] and RS and RW happen 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 happen in

[1] According to the MPI specification, MPI_Send blocks until the user buffer can be reused. The definition of SS and SW follows this convention.

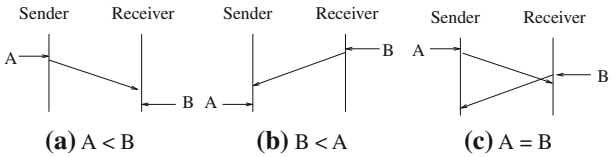


Fig. 1 The ordering of events in two nodes

process P_A, and event B in process P_B. $A < B$ if after A, P_A can notify B; and P_B will receive the notification before B. $A = B$ denotes the case when each party does not have time to deliver the notification (a control message) to other party before the event in that party happens. Figure 1 shows the ordering of events in two processes. Note that this ordering of events is theoretical and used to illustrate protocol invocation scenarios and protocols. In practice, even if the relative timing is mis-predicted, the communication will still be carried out as long as both sender and receiver agree on the same protocol.

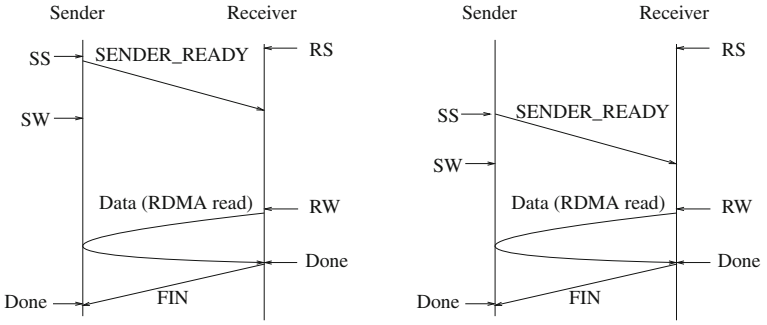
Since $SS \leq SW$ and $RS \leq RW$, there are only six different orderings among the four events in a communication: $SS \leq SW \leq RS \leq RW$, $SS \leq RS \leq SW \leq RW$, $SS \leq RS \leq RW \leq SW$, $RS \leq RW \leq SS \leq SW$, $RS \leq SS \leq RW \leq SW$, and $RS \leq SS \leq SW \leq RW$. 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 Sect. 4.

3.2 Existing Rendezvous Protocols and Their Limitations

In this paper, we assume that protocol operations are performed in MPI routines. All major MPI implementations including MVAPICH [17] and Open MPI [20] use this approach by default. Other schemes such as interrupt-driven communication [1,24,27] and asynchronous communication progress [12,13,15,28] allow protocol operations to be performed outside MPI routines. However, they have significant limitations as discussed earlier, and are not considered.

There are three existing rendezvous protocols developed for RDMA-enabled systems, the traditional sender-initiated RDMA write-based protocol [14], the sender-initiated RDMA read-based protocol [27], and the receiver-initiated protocol [21,23]. We will briefly introduce the protocols 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 discussion of the limitations of sender-initiated RDMA write-based protocol can be found in [26]. Here, we will give examples for the other two more recently developed protocols.

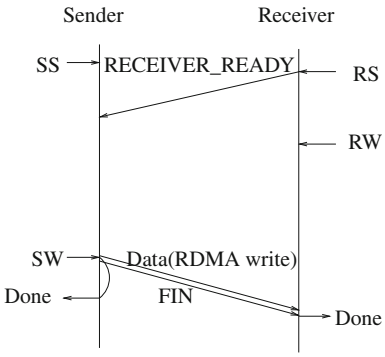
An example of the sender-initiated RDMA read-based protocol [27] is shown in Fig. 2a. In this protocol, the receiver responds to the `SENDER_READY` packet with a RDMA read operation. After the RDMA read operation is completed, the receiver



(a) An example **(b)** Another example

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

Fig. 3 Examples of receiver-initiated protocol with sub-optimal performance



sends a FIN packet to the sender and completes the operation. The sender exits the operation after it receives the FIN packet. This protocol suffers from some limitations as shown in Fig. 2. In Fig. 2a, $SS = RS$. Using this protocol, since `SENDER_READY` misses RS, data transfer cannot happen until RW, which is sub-optimal: ideally, the data transfer should happen when both SS and RS happen, which in this case would have allowed communication to overlap with computation. In Fig. 2b, $RS < SS$. With this protocol, the receiver does nothing at RS and data transfer still happens at RW instead of SS.

An example of the receiver-initiated protocol [21] is shown in Fig. 3. 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. 3, when $SS = RS$, the protocol is not ideal as the data transfer starts at SW.

There are cases that all existing protocols can only give sub-optimal performance [26]. The protocol invocation scenarios can significantly affect the performance of a given protocol: the performance penalties for the inefficient protocol can be very large and depend on the program structure rather than the performance of the underlying

communication system. Our performance study in Sect. 6.1 further demonstrates this. Hence, protocol customization that allows the most effective protocol to be used for each communication can potentially significantly improve the communication performance in comparison to the traditional scheme that uses one protocol for all scenarios. However, to effectively support protocol customization, new efficient protocols must be developed for efficient communication in **all** scenarios.

4 Efficient Protocols for All Protocol Invocation Scenarios

We present 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. Note that relative to communicating large messages, the control message overhead is small. For example, on our test system, a control message takes around 5 μs, while the transmission of a 500KB message takes around 450 μs, about 90 times of the control message. This assumption is for illustrating the protocols. In the implementation, our models take control message overheads into consideration.
- RDMA read and RDMA write have similar performance. This is to simplify the discussion: the difference can be easily accommodated in the communication models for the protocols that are used for protocol selection.
- 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 memory 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 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 the receiver should complete the operation at $REND + comm(msg)$. Here, $REND + comm(msg)$ is $comm(msg)$ time after $REND$.
- 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 $SW + copy(msg)$

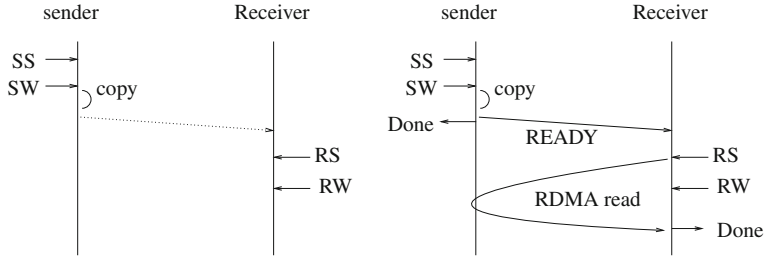
- Given a message size, the copy time and transmission time of that message can be estimated. The memory bandwidth and network bandwith at the message passing layer can be obtained by empirical measurements. With the data in the system, the copy time and transmission time can be estimated.

We note that this ideal communication scheme may not be optimal in that the 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. Hence, we say that this ideal scheme is near-optimal. Our protocols are near-optimal in the sense that, ignoring the control message overheads, they have the same communication time as this ideal 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.

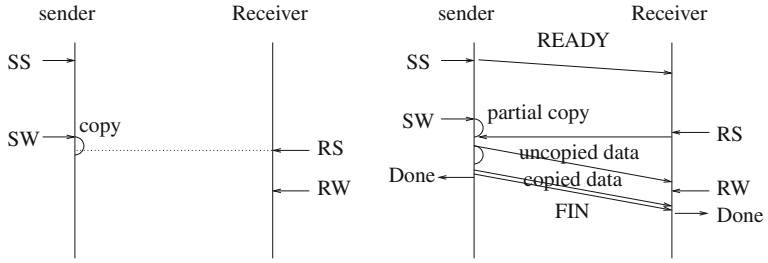
Let us first consider the scenarios with $SS < RS$. The scenarios in this class are further partitioned into three cases with each case having a different protocol. The three cases are: $SS \leq SW < SW + copy(msg) < RS \leq RW$, $SS \leq SW < RS (\leq SW + copy(msg)) \leq RW$, and $SS < RS \leq \{SW\ and\ RW\}$. 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.

Figure 4a 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. 4b, is called the *copy_get protocol*. In this protocol, the sender copies the message data to a local buffer at SW. This does not directly increase the protocol execution time since the sender is blocked for the communication and cannot do anything useful. However,



(a) invocation scenario **(b)** copy_get protocol

Fig. 4 $SS \leq SW < SW + copy(msg) < RS \leq RW$ scenario and the copy_get protocol



(a) invocation scenario **(b)** copy_check_put protocol

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

it requires additional memory to be reserved and creates additional traffic on the memory bus. After the data are copied, the sender issues to the receiver a `READY` message, which contains the address of the local buffer and other related information to facilitate the RDMA read from 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.

Figure 5a 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. 5b, is called the *copy_check_put protocol*. In this protocol, the sender sends a `SENDER_READY` message to the 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 do not delay the useful computation further 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 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. 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 significant 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 $SS < RS \leq \{SW\ and\ RW\}$ scenarios, the traditional sender-initiated RDMA read-based protocol (Fig. 2) is near-optimal. Using this protocol, data transfer starts at $REND = RS$; and both the sender and the receiver complete the communication at $REND + comm(msg)$.

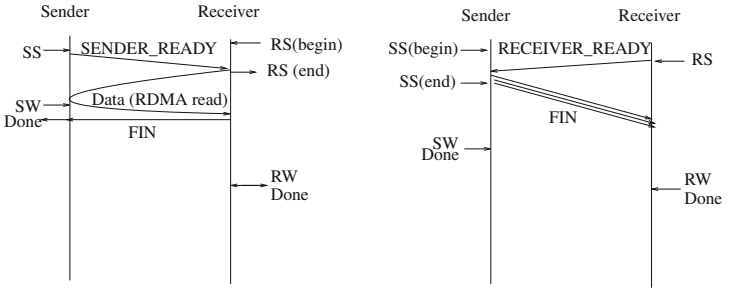
Let us now consider the second class: $SS = RS$. This is one case when no existing rendezvous protocol is ideal. However, if trace/profile/static analysis can draw the conclusion that SS and RS are within one control message time, the solution is straight-forward: waiting for the corresponding control message at SS or RS. We will call such protocols delayed sender-initiated protocols and delayed receiver-initiated protocols. Figure 6a shows a delayed sender-initiated RDMA read-based protocol. In this protocol, the receiver adds a delay time in RS (marked as `RS(begin)` and `RS(end)` in Fig. 6a). During the delay, the receiver repeatedly polls 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 $REND + comm(msg)$. Figure 6b shows the delayed receiver-initiated protocol where the delay is added to the sender at SS.

Finally, for the third class where receiver arrives earlier than sender ($RS < SS$), the receiver-initiated protocol (Fig. 3) 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 scheme.

5 Trace-Driven Protocol Customization

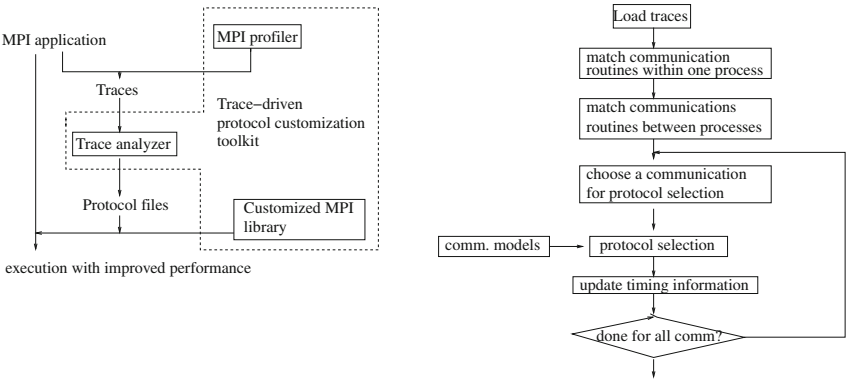
The protocols in the previous section collectively offer high performance for all protocol invocation scenarios: when the protocol invocation scenario information for a communication can be derived, it is possible to select a protocol that is most effective for the communication. To demonstrate the effectiveness of protocol customization, we develop a trace-driven protocol customization toolkit that applies trace analysis techniques to obtain the protocol invocation scenario information for each communication and select the appropriate protocol for the communication. We note that this tool, which is designed for our protocol customization effectiveness study, has various limitations. For example, it does not handle non-deterministic communications with MPI_ANY_TAG and MPI_ANY_SOURCE.

Figure 7a shows the three components in the toolkit, an MPI profiler, a trace analyzer, and a customized MPI library. The MPI profiler uses the PMPI interface to capture accurate timings and all parameters of MPI calls including the routine name,



(a) Delayed sender–initiated protocol **(b)** Delayed receiver–initiated protocol

Fig. 6 Delayed rendezvous protocols for $SS = RS$ scenarios



(a) trace–driven protocol customization toolkit **(b)** The trace analyzer

Fig. 7 Trace-driven protocol customization toolkit

message size, sender id, receiver id, tag, communicator, routine entry time, routine exit time, etc, and store the information in trace files, one for each process. The trace files contain information for all MPI point-to-point routines in the application. The trace analyzer examines the traces, selects protocol for each communication for the application based on the trace information, and generates protocol files, one for each MPI process, that specify the communication protocol to be used by each (point-to-point) communication of the application. The customized MPI library implements the six protocols in Sect. 4 as well as the eager protocol for small messages for InfiniBand clusters. The library supports `MPI_Isend`, `MPI_Irecv`, `MPI_Send`, `MPI_Recv`, `MPI_Wait`, and `MPI_Waitall`. It has wrappers for `MPI_Init` and `MPI_Finalize` to initialize and finalize data structures in the library. All six protocols and the eager protocol are incorporated in the communication progress engine using a similar approach as that in our earlier work [25]. In the `MPI_Init` wrapper, the protocol file that specifies the protocol to be used for each communication is loaded. The customized MPI library can co-exist with MVAPICH. MPI functions that are not supported by our library can be realized by MVAPICH. Protocol customization is performed in the trace analyzer, which we will discuss in more detail next.

5.1 Trace Analyzer

Figure 7b shows the logic in the trace analyzer. After the analyzer reads all traces into memory, it matches nonblocking calls (`MPI_Isend` and `MPI_Irecv`) with corresponding blocking events (`MPI_Wait` and `MPI_Waitall`). For each non-blocking send or receive call, the profiler stores the pointer to request; for each wait and waitall call, the profiler stores the request pointer and count information. These information are used to match non-blocking send and receive calls with wait and waitall operations. The analyzer then matches sending calls with the corresponding receiving calls. After that, the analyzer enters into a loop. In each iteration in the loop, the analyzer uses a heuristic to determine a communication (matching send events and receive events) for protocol selection. Once a communication is determined, the analyzer uses communication models that model the performance of the protocols and select a protocol based on a performance criterion. After the protocol is determined for a communication, the exit time for communication blocking events are updated based on the communication model, along with the timings for the nonblocking events following the blocking events with updated times (before another blocking event in the process). Notice that different protocols will result in different exit times for the blocking events in a communication, which in turn affect the timing of the communication events after them. The time adjustment step accommodates this effect. After the time adjustment, the loop is repeated until protocols are selected for all communications.

We note that the time adjustment may accumulate prediction errors, which in turn can result in errors in protocol selection. We emphasize that the relative timing of communication events is mainly determined by the computation workload in each process. Although prediction in our tool may have cascading errors, their impact on communication timing is secondary as compared to the workload. In Sect. 6.2, we study the protocol selection accuracy of the tool and find that the protocol selection is mostly accurate, with the prediction accuracy ranging from 95.0 to 98.1 % in five benchmarks used in our experiments. This demonstrates that the time adjustment in the tool is sufficiently accurate for a class of applications such as those used in our experiments. In cases when the prediction results significantly deviate from the actual execution, some dynamic runtime approach may be needed to address this problem, which is beyond the scope of this paper.

Determining a communication for protocol selection The analyzer uses a heuristic to choose a communication (the matching send and receive events) for protocol selection, one at a time. Since the selection of a protocol for a communication time will change the exit times for the blocking events (and the times for other events after the blocking events), the ideal order is to follow the global order of blocking events in the whole program so that the protocol invocation scenario of a communication whose protocol is determined earlier will not be affected by one that is selected later. However, this ideal order is not possible due to the inter-dependence of the communication events in different processes: when a protocol is selected for send events, the protocol is also determined for the corresponding receive events in another process. Our system uses the following heuristic that works well in practice: the unselected communication with the smallest global starting time stamp for one of its blocking events is chosen.

Communication models A protocol is selected for a communication based on the estimated performance from our communication models for the protocols. The models assume the memory copy time and data/control message transmission time for different message sizes are known. There are various models for message transmission time such as LogP [4]. To have high accuracy, our system uses a table based approach: we measure the time for memory copy and network transmissions for messages (data and control) of sizes in power of two from 1 byte to 64 MB and establish two tables. The times for other sizes are obtained by extrapolating the data points from the table. The following notations are used in our model: the control message time is $cntl\_time$; the transmission time for message of size msg is $comm(msg)$; memory copy time for message of size msg is $copy(msg)$. SS, SW, RS, and RW denote the starting time of the corresponding events. The models try to estimate the sender and receiver finishing times (times when the SW and RW events complete). We will use the notion end_time to denote the communication completion time for both sender and receiver. When communication is completely overlapped with computation, end_time may be earlier than SW and RW: the communication can be completed before the wait calls. In this case, the exit time for SW and RW should be the same as their entry time. When end_time is later than SW and RW, it should be the exit time for SW and RW.

The models basically emulate the behavior of the protocols in all protocol invocation scenarios. We will discuss the model for the sender-initiated RDMA-read based protocol. The models for other protocols are similar. Figure 8 shows the model for the sender-initiated RDMA-read based protocol. There are three different scenarios to calculate the sender end time and the receiver end time. (1) If the initiate control message arrives at the receiver side before $RS (SS < RS)$, the transmission can start as soon as the receiver arrives at RS. The sender end time is: $end\_time = RS + comm(msg) + cntl\_time$ and the receiver end time is: $end\_time = RS + comm(msg)$. Figure 8a shows this case. (2) If the initiate control message arrives later than RS but earlier than RW ($RS \leq SS < RW$), the transmission can start as soon as the receiver wait is called (RW). The sender end time is: $end\_time = RW + comm(msg) + cntl\_time$ and the receiver end time is: $end\_time = RW + comm(msg)$. Figure 8b shows this case. (3) If the initiate control message arrives later than RW ($RW \leq SS$), the receiver is blocked and the transmission time depends on the sender. The sender end time is: $end\_time = SS + cntl\_time + comm(msg) + cntl\_time$ and the receiver's end time is: $end\_time = SS + cntl\_time + comm(msg)$. Figure 8c shows this case.

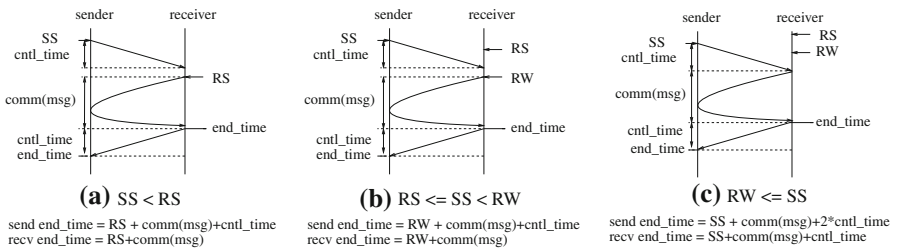


Fig. 8 Model for the sender-initiated RDMA-read based protocol

Notice that the sender and receiver *end_time* may depend on the timing for some of the SS, SW, RS, and RW events. As such, Fig. 8 only shows events that are relevant to the *end_time*. Notice also that the RDMA-read and RDMA-write time can be easily incorporated into the model by replacing $comm(msg)$ with the specific RDMA read time or RDMA write time.

The models give the exit time of the `MPI_Wait` routine. To deal with `MPI_Waitall` that blocks a number of MPI_Isend's, MPI_Irecv's, we treat each communication separately. The exit time for `MPI_Waitall` is determined only after the protocols for all communications in the operation are selected. We accumulate the incoming and outgoing transmissions for the process with each other parties and use the one with the maximum time as its exit time.

Protocol selection Protocol selection assumes that SS, SW, RS, and RW are known. With the models for all protocols, protocol selection seems trivial: for each communication, using the model to decide the communication time for both sender and receiver for each protocol and selecting the protocol with the best performance. There are, however, some complications. First, since a communication involves two parties (sender and receiver) at different processes, optimizing the performance could have different meanings: (1) minimizing the effective communication time at the sender side, (2) minimizing the effective communication time at the receiver side, and (3) minimizing the total effective communication time (the sum of the sender side time and receiver side time). The effective communication time is the time that communication routines take (the real time that the application spends on communications), which excludes the communication time that is overlapped with computation. Each of these options may result in better performance for an application depending on the communication structure in the application. Our analyzer supports any of the three options. In our experiments, minimizing the total communication is used and results in good performance.

Another consideration in protocol selection is the robustness of each protocol. As discussed in the previous section, each protocol offers high performance for the protocol invocation scenario that the protocol is designed for. However, in the protocol selection, the predicted protocol invocation scenario by our analyzer may not always be accurate and the analyzer must cope with mis-prediction since using one protocol in scenarios that the protocol is not designed for can cause penalty that is much higher than other protocols. For example, the delayed rendezvous protocols will have much higher overhead than other protocols and hence, should be used more cautiously. Our protocol selection scheme takes this factor into account.

Using the models described in the previous section as well as our experiments, we compare the robustness of each of the protocols. The *copy_get* protocol is more robust than all other protocols for scenarios that a protocol is not designed for. Intuitively, this is because the copy_get protocol eliminates the dependence from the receiver to the sender. Hence, our analyzer uses the copy_get protocol as the default protocol and selects other protocol only when the protocol invocation scenario can be determined with sufficiently large error margins. It must be pointed out that the *copy_get* protocol increases memory traffic (and buffer space requirement), it may not be ideal when the number of cores per processor is large.

6 Performance Study

We implemented the trace-driven protocol customization toolkit including the MPI profiler, the trace analyzer, and the customized MPI library, for InfiniBand Clusters. The evaluation is performed on an InfiniBand cluster with 16 compute nodes. Each node is a Dell Poweredge 1950 with two 2.33 Ghz Quad-core Xeon E5345's (8 cores per node) and 8 GB memory. All nodes run Linux with the 2.6.9-42.ELsmp kernel. The compute nodes are connected by a 20 Gbps 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 for large messages. Since rendezvous protocols are designed for inter-node communication, our protocol customization only optimizes inter-node communication. To show the effectiveness of the optimization, we run one process on each node and all communications are inter-node.

We compare the performance of our scheme with that of MVAPICH (MVAPICH2-1.2.rc1). To obtain the performance of a program with our scheme, we first run the program linked with the MVAPICH library to collect communication traces. The trace-analysis protocol customization tool then analyzes the traces and produces the protocol file for each process. In the next run when the performance of our protocol customization scheme is measured, our MPI library reads the protocol files at `MPI_Init` and operates the specific protocol for each communication. We have experimented using traces produced by protocols other than the default MVAPICH protocol including the receiver-initiated protocol, the sender-initiated protocol and the copy_get protocol. The performance of the protocol customization scheme is very similar to that using the default MVAPICH to generate the traces. Hence, we conclude that our scheme is not sensitive to methods to generate traces, and report results for traces generated with MVAPICH.

6.1 Micro-Benchmark Results

We use a micro-benchmark to evaluate the performance of the libraries with different protocol invocation scenarios. The micro-benchmark is shown in Fig. 9. 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 unit of 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

Fig. 9 Micro-benchmark

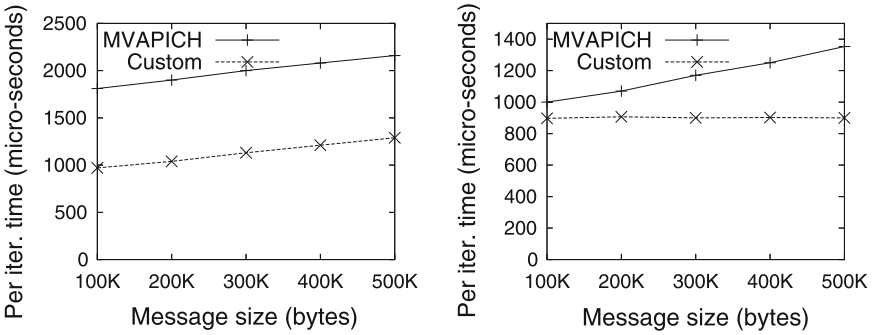
```
Process 1:          Process 2:
  Loop                Loop:
    barrier()           barrier()
    comp1               comp4
    MPI_Isend()         MPI_Irecv()
    comp2               comp5
    MPI_Wait()          MPI_Wait()
    comp3               comp6
```

protocol invocation scenarios. In the discussion, we will use notation $C(compX)$ for the time for $compX$ units of computation, $M(msize)$ 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 $M(100KB) \approx 90\mu s$. This benchmark controls the protocol invocation scenarios for the communications and allows the performance of a library with different protocol invocation scenarios to be evaluated.

We perform experiments using this micro-benchmark with different protocol invocation scenarios. Protocol customization consistently achieves higher 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. 4a. The second case has configuration $(10, 30, 10, 10, 30, 10)$, which emulates the case when $SS = RS$ as shown in Fig. 2a. The third case has configuration $(10, 20, 20, 1, 40, 9)$, which emulates the case when $RS < SS$ as shown in Fig. 2b. 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. 10a. 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 $M(msize) + C(30 + 48)$. On the other hand, with protocol customization, the most effective protocol is the copy_get protocol in Fig. 4b. 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. 10a. Notice that copying data introduces significant overheads as shown in the upward slope for the curve for our scheme in Fig. 10a.

Results for configuration $(10, 30, 10, 10, 30, 10)$ with different message sizes are shown in Fig. 10b. 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 Sect. 3. 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 $M(msize) + C(50)$: the time increases linearly with the message size as shown in Fig. 10b. The most effective protocol for this situation is the delayed receiver-initiated protocol shown in Fig. 6b. Using this protocol, the communication



(a) Configuration $(1, 1, 48, 30, 19, 1)$ **(b)** Configuration $(10, 30, 10, 10, 30, 10)$

Fig. 10 Micro-benchmark results

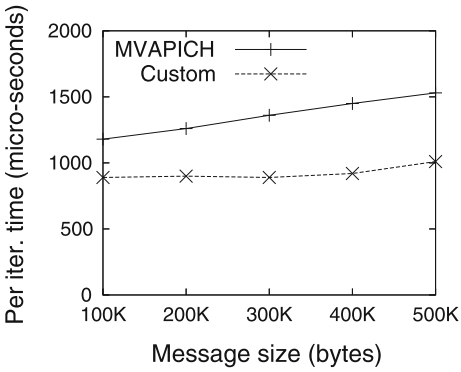


Fig. 11 Results for configuration (10, 20, 20, 1, 40, 9)

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. 10b.

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. 2b. With the default protocol, the communication starts at RW. Hence, the per iteration is roughly $C(41) + M(msize) + C(20) = C(61) + M(msize)$. 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 scenarios, protocol customization avoids the performance penalties due to the mismatch between the protocol and the protocol invocation scenarios and can achieve much higher performance in comparison to traditional schemes in many cases. Moreover, the improvement from protocol customization depends not only on the system communication performance, but also on program structures.

Table 1 Protocol selection accuracy

	# of nodes	# of comm.	# of mis-prediction	Selection accuracy (%)
BT (Class B)	4	9, 672	204	97.9
CG (Class B)	8	39, 520	1, 804	95.4
SP (Class B)	4	19, 272	514	95.0
sparsemm	4	240	11	95.4
jacobi	8	27, 356	514	98.1

6.2 Application Benchmark Results

We use five application benchmarks in the evaluation. Three benchmarks are from the NAS parallel benchmarks [19]: BT, CG, and SP. We use the CLASS B and C problem sizes for all of the three programs. The other two programs are jacobi and sparsemm. The jacobi program uses Gauss–Seidel iterations to solve Laplace equations [10] 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 from the University of Florida sparse matrix collection [6]. The matrix in G3_circuit.mtx is a sparse $1,585,478 \times 1,585,478$ matrix with 4,623,152 non-zero entries.

The execution of the benchmarks on our platform has fairly consistent timing results. We run each benchmark with different communication protocols on different numbers of nodes five times. The timing variation is very small for the programs. The run with the largest standard deviation is CG class B on 8 nodes. The times are 25.38, 25.28, 25.28, 25.37, 25.36 s: the standard deviation is 0.05, less than 0.2 % of the mean value. Since the time variation is small, we will report the performance using the average.

We first investigate the protocol selection accuracy of our trace-driven tool. Table 1 shows the number of large-message communications to which we apply protocol customization, the number of mis-predicted communications, and the selection accuracy for the programs. The number of mis-predicted communications is obtained by profiling the programs with trace-driven protocol customization, and then re-examining the timing of the communication events to determine if the selected protocols are still ideal for the actual protocol invocation scenarios during execution. As can be seen from the table, for the benchmarks that we consider, the protocol selection accuracy is very high, ranging from 95.0 to 98.1 % for the programs. We believe that this is because the timing of the communication events are mainly determined by the computation load at each process, which is measured accurately in the trace and does not change with different communication protocols. We note that using different communication protocols will change the timing to a degree and that the time adjustment approach that we use in the tool may accumulate errors. Such errors, however, are secondary compared to effect of the computation load. This experiment validates to a degree the techniques that we use in the trace-driven protocol customization tool.

Table 2 shows the total application times, total communication times, and the communication improvement percentages using our protocol customization scheme over

Table 2 Performance on 16 processes (one process per node)

	MVAPICH		Customization		Comm. improve. percentage
	Total (s)	Comm. (s)	Total (s)	Comm. (s)	
BT (Class B)	79.32	4.12	76.31	1.29	219.4 %
CG (Class B)	14.64	2.09	14.18	1.70	23.0 %
SP (Class B)	41.7	2.95	40.04	1.32	123.5 %
BT (Class C)	321.79	10.06	315.7	3.95	154.7 %
CG (Class C)	35.78	3.44	35.22	3.07	12.1 %
SP (Class C)	181.23	6.76	177.2	2.75	145.8 %
sparsemm	16.35	12.75	10.51	6.92	84.2 %
jacobi	282.94	2.88	282.33	2.35	22.6 %

MVAPICH for the programs running on 16 processes (one process per node). The communication time includes all Send, Isend, Recv, Irecv, Wait, and Waitall 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 in terms of communication time. 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 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.

To further confirm that protocol customization is the major factor for the improvement, we run the benchmarks with major protocols including the default MVAPICH protocol (traditional rendezvous protocol), the receiver-initiated protocol, the sender-initiated RDMA read-based protocol, the copy_get protocol, and the profile driven customization. Table 3 shows the communication times for the applications with different protocols. As can be seen from the table, different protocols result in different communication times: this indicates that protocols have significant impact on the communication performance and protocol customization is the major contributor for the performance improvement in the study. Protocol customization achieves the best performance among all the schemes for CG, SP, and jacobi, and is slightly worse than the best scheme for BT and sparsemm. Under the theoretical perfect condition, protocol customization should out-perform all other schemes: the fact that it some-

Table 3 Communication time (s) with different protocols (one process per node)

	# of nodes	MVAPICH	Recv_init	Send_init	Copy_get	Customization
BT (Class B)	4	2.95	2.47	2.53	2.72	2.51
CG (Class B)	8	2.12	1.85	2.06	2.27	1.81
SP (Class B)	4	2.88	1.94	2.18	2.29	1.85
sparsemm	4	12.13	12.10	12.11	6.19	6.35
jacobi	8	5.08	4.90	4.83	5.46	4.67

times performs slightly worse than the best performing protocol can be attributed to various inaccuracies in our tool. However, protocol customization significantly outperforms each of the other protocols in different applications: it performs much better than MVAPICH, the receiver-initiated protocol, and the sender initiated protocol on sparsemm, and than copy_get on BT, CG, SP and jacobi. This demonstrates the strength of protocol customization.

7 Conclusion

In this work, we investigate using protocol customization to optimize MPI performance. 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. Building upon the set of protocols, we demonstrate the potential of protocol customization with a trace-driven protocol customization toolkit that automatically selects an efficient protocol for each communication based on the protocol invocation scenario information obtained through trace analysis. Our evaluation with micro-benchmarks and application benchmarks demonstrates that protocol customization can significantly improve MPI communication performance. We point out that parallel programs in the future on massively parallel machines are likely to be communication bound: protocol customization can be a very effective communication optimization technique for such environments.

Acknowledgments This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

References

1. Amerson, G., Apon, A.: Implementation and design analysis of a network messaging module using virtual interface architecture. In: Proceedings of the 6th International Conference on Cluster Computing, San Diego, CA, pp. 255–265, September 2004
2. Buluc, A., Gilbert, J.R.: Challenges and advances in parallel sparse matrix-matrix multiplication. In: Proceedings of the 37th International Conference on Parallel Processing, Portland, OR, pp. 503–510, September 2008
3. Chen, D., Eisley, N.A., Heidelberger, P., Senger, R.M., Sugawara, Y., Kumar, S., Salapura, V., Satterfield, D., Steinmacher-Burow, B., Parker, J.: The IBM blue gene/Q interconnection network and message unit. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis,vol. 26, (2011)

4. Culler, D. et al.: LogP: towards a realistic model of parallel computation. In: Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, pp. 1–12 (1993)
5. Danalis, A., Brown, A., Pollock, L., Swany, M., Cavazos, J.: Gravel: a communication library to fast path MPI. In: Proceedings of the 15th European PVM/MPI Users' Group Meeting, LNCS 5205, Dublin Ireland, pp. 111–119, September 2008
6. Davis, T.: The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/
7. Faraj, A., Patarasuk, P., Yuan, X.: A study of process arrival patterns for MPI collective operations. Int. J. Parallel Program. **36**(6), 543–570 (2008)
8. InfiniBand Trade Association. http://www.infinibandta.org
9. InfiniBand Host Channel Adapter Verb Implementer's Guide. Intel Corp. (2003)
10. Hennings, A.: Matrix Computation for Engineers and Scientists. Wiley, New York (1977)
11. Ke, J., Burtscher, M., Speight, E.: Tolerating message latency through the early release of blocked receives. In: Proceedings of the 11th European Conference on Parallel Processing (Euro-Par), LNCS 3648, Lisboa, Portugal, pp. 19–29, August 2005
12. Keppitiyagama, C., Wagner, a.: Asynchronous MPI messaging on myrinet. In: Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, CA, pp. 50–57, April 2001
13. Kumar, R., Mamidala, A., Koop, M., Santhanaraman, G., Panda, D.K.: Lock-free Asynchronous rendezvous design for mpi point-to-point communication. In: Proceedings of the 15th European PVM/MPI Users' Group Meeting, LNCS 5205, Dublin Ireland, pp. 185–193, September 2008
14. Liu, J., Wu, J., Kini, S.P., Wyckoff, P., Panda, D.K.: High performance RDMA-based MPI implementation over InfiniBand., In: Proceedings of the 17th International Conference on Supercomputing (ICS), San Francisico, CA, pp. 295–304, June 2003
15. Majumder, S., Rixner, S., Pai, V.S.: An event-driven architecture for MPI libraries. In: Proceedings of the 5th Los Alamos Computer Science Institute Symposium (CD-ROM proceedings), Santa Fe, NM, October 2004
16. Marathe, A., Lowenthal, D., Gu, Z., Small, M., Yuan, X.: Profile guided MPI protocol selection for point-to-point communication calls. In: The 1st IPDPS workshop on Communication Architecture for Scalable Systems (CASS), Atlanta, GA, pp. 1–7, May 2011
17. MVAPICH. http://mvapich.cse.ohio-state.edu/
18. Myricom. http://www.myricom.com
19. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/
20. Open MPI: open source high performance computing. http://www.open-mpi.org/
21. Pakin, S.: Receiver-initiated message passing over RDMA networks. In: Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), Miami FL, pp. 1–12, April 2008
22. Rabenseinfner, R.: Automatic MPI counter profiling of all users: first results on CRAY T3E900-512. In: Proceedings of the 3rd Message Passing Interface Developer's and User's Conference, Atlanta, GA, pp. 77–85, March 1999
23. Rashti, M.J., Afsahi, A.: Improving communication progress and overlap in MPI rendezvous protocol over RDMA-enabled Interconnects. In: Proceedings of the 22nd High Performance Computing Systems and Applications Symposium (HPCS), Quebec City, Canada, pp. 96–101, June 2008
24. Sitsky, D., Hayashi, K.: An MPI library which uses polling, interrupts, and remote copying for the Fujitsu AP1000+. In: Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks, Beijing, China, pp. 43–49 (1996)
25. Small, M., Yuan, X.: Maximizing MPI point-to-point communication performance on RDMA-enabled clusters with customized protocols. In: Proceedings of the 23th ACM International Conference on Supercomputing (ICS), Yorktown Heights, NY, pp. 306–315, June 2009
26. Small, M., Gu, Z., Yuan, X.: Near-optimal rendezvous protocols for RDMA-enabled clusters. In: Proceedings of the 38th International Conference on Parallel Processing, San Diego, CA, pp. 644–652, September 2010
27. Sur, S., Jin, H., Chai, L., Panda, D.K.: RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In: Proceedings of the 11th ACM SIGPLAN symposium on principles and practice of parallel programming, New York, NY, pp. 32–39, March 2006

28. Tipparaju, V., Santhanaraman, G., Nieplocha, J., Panda, D.K.: Host-assisted zero-copy remote memory access communication on InfiniBand. In: Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium, Santa Fe, NM, p. 31a, April 2004
29. Venkata, M.G., Bridges, P.G., Widener, P.M.: Using application communication characteristics to drive dynamic MPI reconfiguration. In: Proceedings of the 9th IPDPS Workshop on Communication Architecture for Clusters, Rome Italy, pp. 1–6, May 2009