Int J Parallel Prog (2009) 37:223–246
DOI 10.1007/s10766-009-0094-9

A Speculative and Adaptive MPI Rendezvous Protocol Over RDMA-enabled Interconnects

Mohammad J. Rashti · Ahmad Afsahi

Received: 21 September 2008 / Accepted: 10 February 2009 / Published online: 6 March 2009

Abstract Overlapping computation with communication is a key technique to conceal the effect of communication latency on the performance of parallel applications. Message Passing Interface (MPI) is a widely used message passing standard for high performance computing. One of the most important factors in achieving a good level of overlap is the MPI ability to make progress on outstanding communication operations. In this paper, we propose a novel speculative MPI Rendezvous protocol that uses RDMA Read and RDMA Write to effectively improve communication progress and consequently the overlap ability. Performance results based on a modified MPICH2 implementation over 10-Gigabit iWARP Ethernet reveal a significant (80–100%) improvement in receiver side overlap and progress ability. We have also observed up to 30% improvement in application wait time for some NPB applications as well as the RADIX application. For applications that do not benefit from this protocol, an adaptation mechanism is used to stop the speculation to effectively reduce the protocol overhead.

Keywords MPI rendezvous protocol · Overlap · Communication progress · RDMA · High-performance interconnects

M. J. Rashti · A. Afsahi (✉)
Department of Electrical and Computer Engineering, Queen's University,
Kingston, ON K7L 3N6 Canada
e-mail: ahmad.afsahi@queensu.ca

M. J. Rashti
e-mail: mohammad.rashti@ece.queensu.ca

1 Introduction

Most scientific applications running on clusters are written on top of Message Passing Interface (MPI) [1]. MPI implementations typically use two different protocols for transferring small and large messages: *Eager* and *Rendezvous,* respectively. In the Eager protocol, the sender sends the entire message to the receiver, where the receiver provides sufficient buffering space for the incoming message. This protocol is mainly used for sending small messages. The Rendezvous protocol is used for large messages, where the cost of copying is prohibitive. The sender and receiver negotiate the availability of the receiver side buffer before the actual data transfer.

Overlapping computation with communication is one of the basic techniques in hiding communication latency, thereby improving application performance. Using non-blocking communication calls at the application level [2,3], supporting independent progress for non-blocking operations at the messaging layer [4,5], and offloading communication processing to the Network Interface Card (NIC) [4,6] are the main steps in achieving efficient communication/computation overlap.

NICs in modern interconnects are designed to offload most of the network processing tasks from the host CPU, providing excellent opportunity for communication libraries such as MPI to hide the communication latency using non-blocking calls. To utilize the offload engines efficiently, non-blocking communications need to make progress independently. While some MPI implementations support independent progress, others require subsequent library calls in order to make progress in outstanding non-blocking calls. This may have a significant impact on performance when a computation phase follows a non-blocking call. Specifically, communication progress becomes more important for the Rendezvous protocol where a negotiation exists prior to the actual data transfer. This is simply because a non-blocking call may return without completing the negotiation. In [5], the authors have shown that how lack of a complete and independent progress in the MPI Rendezvous protocol on top of modern interconnects can negatively affect the overlap ability.

While most MPI implementations use polling-based progress engines, some use interrupts for communication progress [7–9]. Although interrupt-based approaches activate the progress engine any time communication progress is needed, they impose high overhead and fluctuation on the communication time, which cannot be overlooked. In this research, we focus on addressing the shortcomings of current polling-based protocols, by proposing a speculative and adaptive MPI Rendezvous protocol that could increase communication progress and overlap. Unlike the current Rendezvous protocols that rely only on the sender to initiate the Rendezvous, our proposed protocol lets either the sender or the receiver initiate the negotiation so that the data transfer can be started before the non-blocking send or receive call returns. This will enable overlapping the computation phase following the non-blocking calls with the communication. To the best of our knowledge, this is the first study that proposes a speculative method in designing the MPI Rendezvous protocol for high performance clusters.

We have implemented the new protocol on MPICH2 [10] over 10-Gigabit iWARP Ethernet [5,11]. The assessment is done using our overlap and progress micro-benchmarks for both sender and receiver and under two timing scenarios, where either the sender arrives first in the communication call, or the receiver arrives first. Our

experimental results indicate that the proposed protocol is able to effectively improve the receiver-side progress and overlap from almost 0% to nearly 100% when the receiver arrives first, at the expense of only 2–14% degradation for the sender-side overlap and progress. Our implementation also significantly improves the receiver-side progress and overlap ability when the sender-side arrives earlier, without any negative effect on the sender-side overlap and progress.

The basic protocol was presented in [12]. In this paper, we have extended our proposed speculative MPI Rendezvous protocol, and conducted some more experiments at the micro-benchmark level, and worked with some MPI applications in the NAS Parallel Benchmark (NPB) suite [13] as well as the RADIX application [14] that sorts a series of integer keys using the radix algorithm. We have also equipped the protocol with an adaptation mechanism that disables the speculative protocol when it cannot benefit the application due to the application timing, application communication pattern, and protocol mispredictions. In our experiments, up to 30% reduction in total application wait time has been observed. Meanwhile, the overhead associated with the implementation is shown to be very low, when the adaptation mechanism is in use.

The rest of this paper is organized as follows. Related work is reviewed in Sect. 2. Section 3 discusses the proposed Rendezvous protocol. In Sect. 4, we present and analyze the experimental results. Finally, Sect. 5 concludes the paper.

2 Related Work

In this section, we will first review the related work on the evaluation and analysis of overlap and communication progress in MPI implementations. We will then discuss related research on improving the overlap and communication progress in MPI.

In [4], the authors compared the impact of six MPI implementations on application performance running on two platforms with Quadrics QsNet [15] and CNIC network interface cards. Their results show that in almost all benchmarks, combination of offload, overlap, and independent progress significantly contributes to the performance. The study in [16] concerns a similar work on InfiniBand (IB) [17] and Quadrics QsNet^{II} [18]. While the work in [4,16] is focused at analyzing application's ability in leveraging overlap and progress, we optimize the MPI Rendezvous protocol itself in this paper.

The authors in [19] have proposed an overlap measurement method for MPI. In their method, the communication overhead is first computed and then application availability is calculated using the overhead amount. While [20,21] address combined send and receive overlap, our proposed overlap measurement method in this paper, along with [8,19], target non-blocking send and receive overlaps separately.

In [5], the authors analyze the overlap and communication progress ability in InfiniBand [17,22], Myrinet-10G [23] and 10-Gigabit iWARP Ethernet [24,25], and conclude that transferring small messages makes an acceptable level of independent progress. On the other hand, in most cases transferring large messages does not make progress independently, decreasing the chances of overlap in applications. The results in [5] confirm that independent progress is required, at least for data transfer, to achieve high overlap ability with non-blocking communications. The paper also shows how

different Rendezvous protocols affect overlap and communication progress in different networks.

Several methods have been proposed to improve overlap and communication progress in message passing systems. On the issue of using interrupts in the Rendezvous protocol, the solution in [26] is based on RDMA Write. They show that the interrupt-based notification tends to yield more overlap ability, compared to the completion queue (CQ) polling method in InfiniBand. Researchers have also proposed RDMA-Read MPI Rendezvous protocols [8] to improve communication progress and overlap. In their work, a traditional two-way handshake followed by RDMA Write is replaced by a one-way handshake followed by RDMA Read. They also use an interrupt-based scheme with a progression thread to alleviate the bottleneck when the receiver arrives first, achieving nearly complete overlap and up to 50% progress improvement in MPI over InfiniBand relative to the RDMA Write Rendezvous protocol, plus about 30% MPI wait time improvement in some parallel applications.

The results in [8] and also in [5] show that shortened Rendezvous protocols using RDMA Read help achieve a good level of overlap and progress at the send side. However, both one-way Rendezvous (used in the latest MVAPICH over IB [27] and MPICH2 over iWARP) and two-way Rendezvous (used in the latest MVAPICH2 over IB [27]) protocols are not able to provide independent progress and good overlap for receiving large messages [5].

The most limitation of the work in [8] is the use of locks (for shared data structures) and interrupts that make the communication time unpredictable and sometimes costly. A more enhanced version of this work has been recently proposed in [28], where the authors propose a lock-free mechanism in which the auxiliary thread is interrupted and then signals the main thread to take care of the progress when a new control message is arrived. With his method, they are able to achieve higher overlap results compared to [8]. Although they remove some of the interrupts from their earlier work in [8], interrupts to awaken the progression thread are still inevitable due to asynchrony of the progress.

The work in [29] is perhaps the closest to our research in this paper, in which the author introduces a similar method for improving MPI communication over Cell Broadband Engine [30] processors. Contrary to our work that is focused at increasing communication/computation overlap, the work in [29] seeks to improve the MPI communication latency by making the receiver start the communication. The proposed protocol [29] is basically an amendment to the RDMA-Write based protocol by running the protocol in two steps rather than three steps, when the receiver arrives earlier than the sender. It is not concerned about protocol prediction, mispredictions, or race conditions, apparently because the MPI implementation under study is limited only to a single protocol that is similar to the Rendezvous protocol.

3 The Proposed Speculative MPI Rendezvous Protocol

In this section, we will discuss the details of our proposed MPI Rendezvous protocol. Section 3.1 explains the basics of the proposed protocol for two different timing scenarios. We will then discuss the protocol design specification in detail in Sect. 3.2.

Sections 3.3 and 3.4 cover the methodologies in preventing race and deadlock conditions. Protocol usability is discussed in Sect. 3.5. Finally, Sect. 3.6 describes the adaptation mechanism used to reduce the protocol overhead.

3.1 Protocol Preliminaries

In the current RDMA-Read based Rendezvous protocol [8], used in the implementation of MPI non-blocking communication calls for large messages, the sender sends a *Request to Send* (RTS) message to the receiver that includes the sender's data buffer address. Upon receiving the RTS, the receiver process will transfer the data using RDMA Read. Basically, there are two send and receive timing scenarios in the current Rendezvous protocol that could happen at runtime: (1) the sender arrives first at the send call; and (2) the receiver arrives first at the receive call.

In the first scenario, when the receiver arrives at the receive call the RTS negotiation message is assumed to be already in the receive buffer. Thus, the data transfer can start right away. In the second scenario, the receiver calls the non-blocking receive call before receiving the RTS message. It will not see any RTS to start the RDMA-Read based data transfer, and therefore starting data transfer will remain for the progress engine, which is activated either by a costly interrupt [8] or in the next MPI communication call. Consequently, any computation phase after the non-blocking receive call can delay the data transfer [5]. The method that is proposed in this paper allows the receiver to initiate the communication when it arrives earlier. In the following, we discuss the basics of our proposed protocol for the two timing scenarios of the current protocol.

3.1.1 Sender Arrives First

In this timing scenario, the receiver is supposed to transfer the data using RDMA Read after receiving a matching RTS from the sender. Figure 1 depicts the theoretical behavior for the current Rendezvous protocol when the sender arrives first. However, due to the one-sided nature of the RDMA operation used to transfer the RTS from the sender, in addition to inefficiency in the current implementation of MPICH2 over RDMA-enabled channels, the receiver is not able to find the already arrived RTS. Therefore, a *receive request* (Rreq) will be enqueued in the *receive queue* (Recvq),

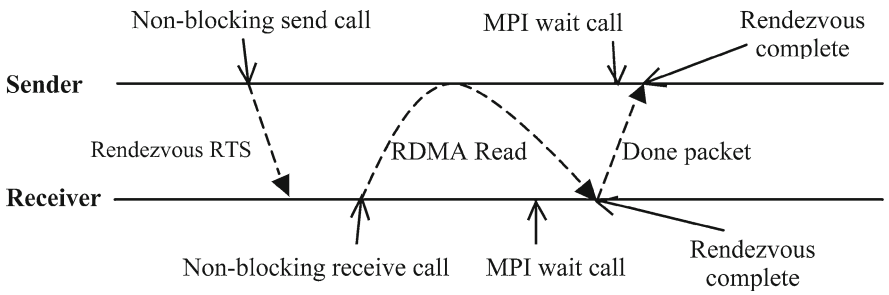

Fig. 1 Rendezvous protocol in MPICH2-iWARP when sender arrives first

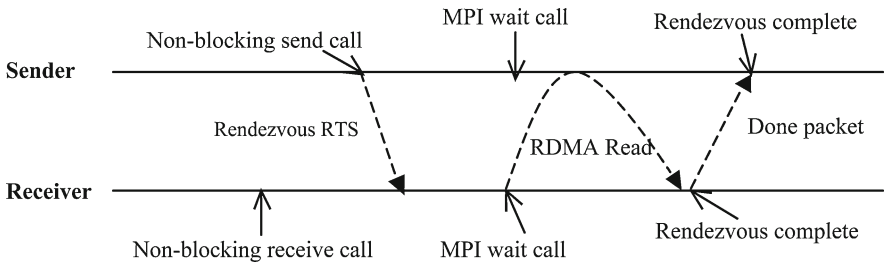

Fig. 2 Rendezvous protocol in MPICH2-iWARP when receiver arrives first

leaving the communication progress to a future progress engine call. The results presented for both small and large messages in [5] (over RDMA-enabled iWARP and IB networks) highlight the existence of such inefficiency, resulting in non-independent progress for both Eager and Rendezvous protocols. Investigating the issue inside the implementation of MPICH2 prompted us that an initial progress engine call is needed to put the RTS message from the channel-related buffers into the receiver *unexpected queue* (Unexq). The receiver is then able to recognize the arrived message and take appropriate action immediately, before returning from the non-blocking call. This initial progress engine call is also beneficial for the send side, as will be described in Sect. 3.2.2.

3.1.2 Receiver Arrives First

In this timing scenario, the receiver enqueues an Rreq in the Recvq and returns from the non-blocking call. The communication will progress when the progress engine becomes active by a subsequent library call. Figure 2 depicts the current Rendezvous protocol when the receiver arrives first. In the current protocol, the receiver does not start the Rendezvous negotiation because it is only the sender that knows the communication mode (e.g. synchronous or ready mode) and/or the exact size of the message.

In our proposal, the early-arrived receiver predicts the communication protocol based on its own local message size. If the predicted protocol is Rendezvous, a message similar to RTS (we call it *Request to Receive* or RTR) is prepared and sent to the sender. This message also contains information about the receiver's registered user buffer. At the sender side, if the Rendezvous protocol is chosen, the RTR message (arrived in the Unexq) is used to transfer the data to the receiver using RDMA Write. Otherwise, if the Eager protocol is chosen, the arrived RTR will be simply discarded. Figure 3 shows a sample timing diagram for our proposed receiver-initiated Rendezvous protocol. Using this protocol, there is more potential for communication progress and overlap of data transfer with the computation phase following a non-blocking send or receive call.

3.2 New Protocol Design Specification

In this section, we will elaborate on the design and implementation details of the proposed Rendezvous protocol. Basically, the protocol is executed by three MPI library

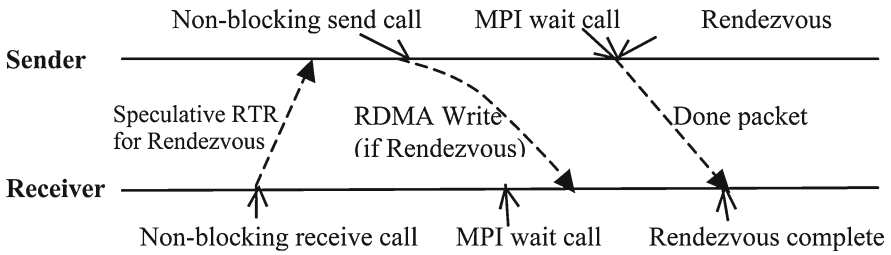

Fig. 3 Timing diagram for receiver-initiated Rendezvous protocol when receiver arrives first

components: the non-blocking send call, the non-blocking receive call, and the progress engine.

3.2.1 Non-blocking Receive Operation

In the non-blocking receive call, the sender's protocol is predicted using the local message size. For Eager prediction, the regular receive communication path for Eager messages is followed. For Rendezvous, the receiver will start the communication with an RTR if it finds itself arriving earlier than the sender. On the other hand, in case of an early sender, the current RDMA-Read based Rendezvous protocol is performed using the RTS from the sender side.

Figure 4 depicts a detailed view of the protocol flow when an MPI non-blocking receive is called. Initially, a protocol prediction is performed based on the local message size at the receive side. In the case of Rendezvous protocol, an initial progress engine call is made. Then, for either case of the prediction outcome, the Unexq is checked for a matching message. If an Eager message is found, it will be placed into the user buffer and the communication will be finalized. If a matching RTS is found, the receiver will start the data transfer using RDMA Read. If no matching is found in the Unexq, the receiver side will post the Rreq to the Recvq. If the Rendezvous

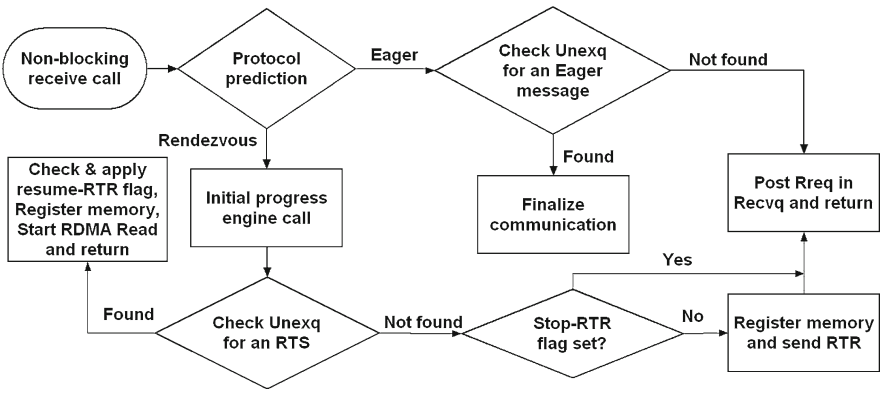

Fig. 4 Simplified protocol flow for a non-blocking receive operation

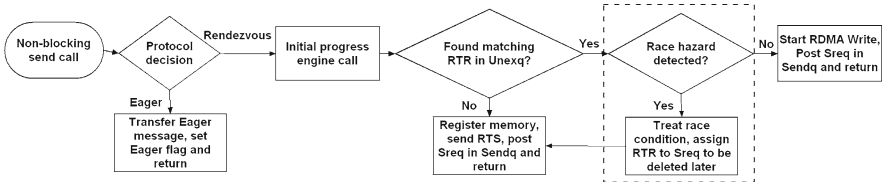

Fig. 5 Simplified protocol flow for a non-blocking send operation

protocol has been predicted, the local memory will be registered and an RTR message will be prepared and sent to the sender to initialize the Rendezvous protocol, unless producing RTR has been disabled (please refer to the adaptation mechanism in Sect. 3.6 and the stop-RTR flag in Sect. 3.3 for cases that RTR is stopped).

3.2.2 Non-blocking Send Operation

As stated earlier, protocol decision is made at the sender side. Eager messages are sent as before. In case of Rendezvous, and if the sender arrives earlier than the receiver, the current Rendezvous protocol is followed. Otherwise, the new receiver-initiated RDMA-Write based Rendezvous will be performed using an RTR message sent by the receiver. The sender finalizes the communication after performing an RDMA Write.

A detailed protocol flow for non-blocking send is shown in Fig. 5. A non-blocking send call will first decide the appropriate protocol, Eager or Rendezvous. In the case of Eager, the message is transferred eagerly. Since the receiver may mispredict the protocol, it may send an RTR for the current send operation. The mispredicted RTR may mistakenly be used for a future matching send call, posing a race hazard. To prevent such a miss-assignment, the mispredicted RTR should be deleted. Due to the possibility of misprediction at the receiver side some race conditions are possible. A complete description of treating race hazards is given in Sect. 3.3. The dashed region in Fig. 5 (regarding race treatment) is expanded in Fig. 11.

Similar to the case for the non-blocking receive, if the Rendezvous protocol is chosen, an initial progress engine call is invoked and then the Unexq is searched for any possibly arrived matching RTR from the receiver. In case a matching RTR is found, the sender will immediately initiate data transfer using RDMA Write, unless a race hazard is detected. The send request (Sreq) will be also enqueued into a send request queue (Sendq) for future reference. We have included a Sendq in our design for posted send calls that have not yet completed their communication. On the other hand, if no RTR is found, the sender will register the local memory and start the negotiation by sending an RTS to the receiver.

3.2.3 MPI Progress Engine

The progress engine in the proposed protocol handles the incoming RTR and RTS messages, and finalizes the communication (e.g. sending *done* packets and deregistering memory). The RTR processing unit is the new component in the proposed progress

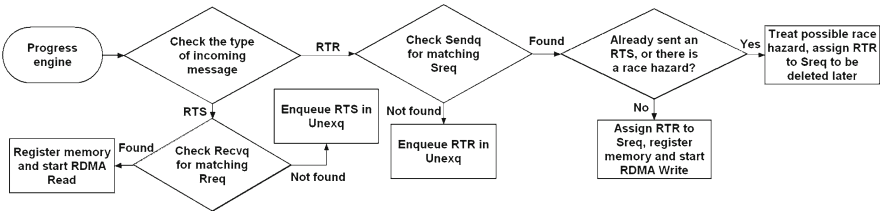

Fig. 6 Simplified progress engine flow for the proposed Rendezvous protocol

engine. The job of the new progress engine is to find a matching send/receive request for the arrived RTR/RTS message in the corresponding send/receive queues, and to start the RDMA-Write/RDMA-Read operations accordingly if a matching request is found.

Figure 6 shows a detailed behavior of the progress engine according to the proposed protocol. Handling an incoming RTS is as it was before. It is assigned to the first matching Rreq in the Recvq. After removing the matched Rreq from the Recvq, the corresponding data buffer is registered, and the data is transferred using RDMA Read. If no matching request is found in the Recvq, the RTS is placed in the Unexq for future use.

Dealing with RTR is a bit different, though. If no Sreq is matched against the arriving RTR, the RTR will be placed in the Unexq to be assigned to a future Sreq. Otherwise, the RTR will be assigned to the first matching Sreq in the Sendq that has no RTR assigned to it. The buffer registration and RDMA Write (for data transfer) will take place if no RTS has been sent to the receiver before, and the RTR is not marked for deletion due to race hazard conditions.

3.3 Race Hazards Prevention

Unlike the current Rendezvous negotiation model, in which only the sender is responsible for starting the Rendezvous, in the proposed protocol both sender and receiver are able to start the negotiation. Therefore, the new protocol should be checked for race and deadlock conditions. Appropriate modifications and constraints should be considered to avoid undesirable behavior.

Based on the timing of a send/receive call pair, one or both of the calls may start Rendezvous negotiation by sending RTS/RTR to the other side. It is also possible that a send/receive request does not produce RTS/RTR due to the reception of RTR/RTS from the other side. Therefore, the proposed protocol with its basic makeup cannot determine the actual source of a received RTS/RTR. Consequently, the actual target receive/send request cannot be determined. Another reason that makes the protocol undecided and ambiguous about the source/target of a received RTR is the possibility of protocol misprediction at the receiver side. Thus, in either case, the new protocol could cause incorrect assignment of RTR/RTS messages to the target send/receive requests. We call such a potential a race hazard. In the following, we enumerate different cases that could end up with a race condition. We then propose methods to prevent the hazards for each case.

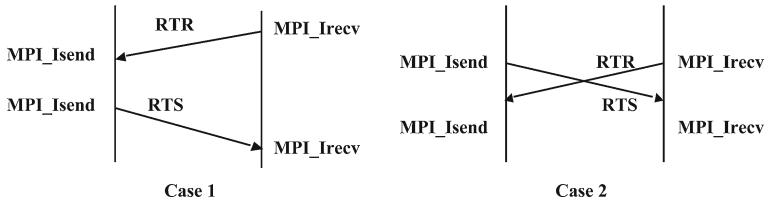

Fig. 7 An RTS race scenario

3.3.1 Ambiguous RTS/RTR Destination Problem

Consider the cases shown in Fig. 7. All depicted messages have matching message envelopes. In case 1, the first send call finds the arrived RTR and therefore does not send any RTS. As shown, the RTS is coming from the second send call for the second receive call. On the other hand in case 2, since the first send call has not yet received the RTR, it sends the RTS for the first receive call. From the receiver's point of view, both cases are identical. In both cases, an RTR has been sent by the first receive call, and an RTS is received. Therefore, the receiver cannot determine the target Rreq for the arrived RTS message. The same scenarios as shown in Fig. 7 for RTS messages can generate race hazards for RTR messages as well.

The main reason that the above conditions are hazardous is that the RTS/RTR message assignment policy is not yet defined in our protocol. In the current Rendezvous protocol, the RTS messages are assigned to the posted Rreqs in the Recvq in order. We keep this policy in the proposed Rendezvous protocol, and extend it to the RTR assignment as well. Therefore, each incoming RTR or RTS/Eager message will be assigned to the first send or receive request in the posted queues, respectively. In addition to this rule, and to avoid the race conditions shown in Fig. 7, the following rule is amended to the protocol: *a Sreq/Rreq that finds an RTR/RTS does not need to send an RTS/RTR, and should start the data transfer*. However, to prevent the race condition, an *acknowledgement* (ACK) message will be sent to the other side. With this ACK (to be assigned to its peer Rreq/Sreq), the other side will no longer expect any RTS/RTR from that Sreq/Rreq, and will assign the subsequent incoming RTS/RTRs to the next posted matching Rreq/Sreq. It should be noted that an Sreq/Rreq, which has already sent an RTS/RTR, does not need to send an ACK for an RTR/RTS that it receives later.

Figure 8 illustrates how the race conditions in Fig. 7 can be avoided using the proposed policy. Similar to Fig. 7, all messages depicted in Fig. 8 have a matching message envelope. In case 1, the RTR-ACK will be assigned to the first receive call, removing it from the Recvq. Consequently, the RTS will be assigned to the second receive call (as it is now the first matching Rreq in the Recvq). In case 2, the first send call will not send an RTR-ACK, because it has already sent the RTS. The arrived RTS at the receiver side will be assigned to the first receive call (which is the first Rreq in the Recvq). As we can see, the ambiguity surrounding both RTS and RTR assignments are now resolved with the acknowledgement technique.

Exceptions: All send/receive calls can be expected to send a control message (either RTS/RTR or an acknowledgment), except a receive call that predicts the Eager

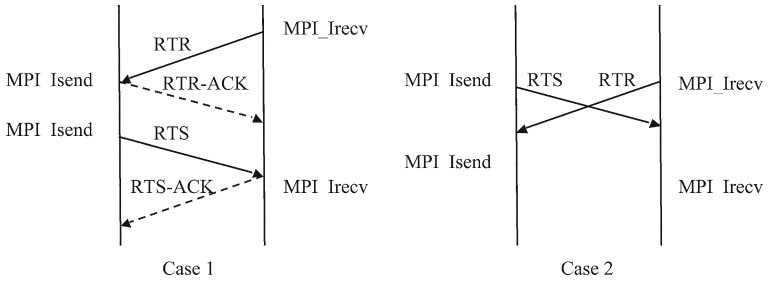

Fig. 8 Sending acknowledgements to avoid race hazard

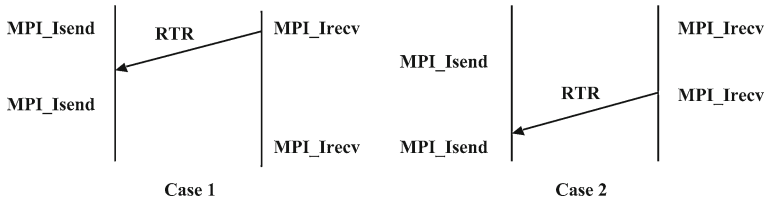

Fig. 9 Race hazard related to RTR and Eager prediction by receiver

protocol, or a receive call that its message source is indicated as MPI_ANY_SOURCE. For neither case, an RTR is applicable. If any of these situations happens, an RTR race may occur. This race is related to the case that an Rreq with no RTR can be posted into the Recvq. Consider the scenarios in Fig. 9. In case 1, the first receive call predicts a Rendezvous protocol and sends an RTR. In the other case, the first receive call is not sending an RTR because one of the aforementioned exceptions happens. In both cases, the target Sreq for the matching RTR arrived at the sender side is not known for the sender. This is because the send side cannot verify if the RTR is coming from the first Rreq (case 1) or the second one (case 2), as it is not aware of the outcome of their prediction.

To cover this race condition, the following policy is proposed: *no matching RTR will be sent if there is any matching Rreq in the Recvq that has not sent an RTR*. In such a case, all the incoming matching receive requests have to wait for the send side to start the communication, until the first Rreq (which has not sent RTR) is removed from the queue by receiving RTS or Eager message. Essentially, the race conditions are over then.

3.3.2 Mispredicted RTR Race Hazard

Mispredicted RTR messages form another source of a race hazard, as shown in Fig. 10. Based on the MPI specification, the size of the receiver buffer can be different (larger) than the actual sender buffer. In addition, the sender may decide to use a protocol for a specific communication mode (e.g. ready or synchronous modes), regardless of the message size. Therefore, the receiver protocol prediction may be different than the sender decision. As shown in case 2 of Fig. 10, the receiver may mispredict the Rendezvous protocol and send an RTR to the sender, while the sender has decided to use

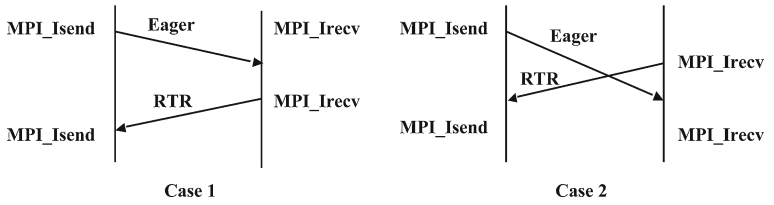

Fig. 10 A mispredicted RTR race scenario

the Eager protocol. On the other hand, in case 1, the receiver predicts the Rendezvous protocol correctly and the RTR is targeted for the second send call. Hence, there is a potential race condition here, because both cases look identical from the sender's perspective, and thus the target Sreq for the RTR is ambiguous at the sender side.

To avoid the hazard of mispredicted RTR messages, a flag is introduced, to be called *Eager-flag*. This flag, which is separate for each message envelope, is set by each send call transferring an Eager message. This is a warning for the subsequent matching Rendezvous send calls from the same process to be aware of the possibility of mispredicted RTRs. Since we cannot exactly determine the mispredicted RTRs, the safest method to avoid the race is to drop all RTRs matching with the current Rendezvous Sreq, and start the communication using RTS. Because the current send call cannot make sure whether all previously sent matching RTRs have already arrived or not, the following Eager flag treatment policy is proposed:

- The first Sreq that finds a matching set Eager-flag will start the communication using an RTS and piggyback a *stop-RTR* flag to the receiver side.
- As soon as the receiver side receives the RTS with the stop-RTR flag, it will set a local flag that stops producing RTR for this specific message envelope.
- Once an RTS-ACK or a *done* packet (showing the end of the RDMA Read) has been received by the sender, the sender will make sure that all on-the-fly matching RTRs have arrived and no more matching RTRs is forthcoming. Therefore, the sender can now remove all matching RTRs. Once all RTRs are removed by subsequent Rendezvous Sreqs, a *resume-RTR* flag will be piggybacked on one of the matching RTS packets targeted to the other side. This means that all ambiguous RTRs are now safely removed and the receiver side can resume sending RTR messages.

Figure 11 shows the completed protocol flow for the send operation. The dashed area in Fig. 11 shows the expansion of the dashed area in Fig. 5.

3.3.3 Accumulation of RTR Messages

A less serious issue associated with the mispredicted RTR messages is their accumulation at the sender side. If the sender continues generating Eager messages while the receiver is predicting the wrong protocol, the generated RTRs will be accumulated in the Unexq at the sender side. To prevent this to happen, and also to increase the adaptability of the protocol (in order to decrease extra RTR messages to be generated), the

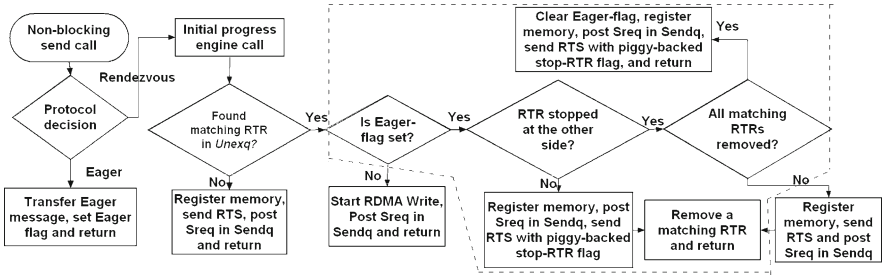

Fig. 11 Complete protocol flow for a non-blocking send with Eager-flag management

receiver should watch for mispredictions and temporarily disable RTR generation for that specific message envelope. In Sect. 3.6, we will discuss the adaptation mechanisms in the proposed protocol in more detail.

3.4 Deadlock Hazards Prevention

For a message transfer, deadlock happens when neither the sender nor the receiver proceeds with the communication. In the proposed protocol, there are certain cases that the receiver can post a request into the Recvq without sending an RTR. There are also cases that the sender drops the arrived RTR. However, based on the protocol description in Sect. 3.2 for the send side, either a previously arrived RTR is used or an RTS is sent to the receiver. Therefore, even in the case that an RTR is not sent or is dropped, we will have the current Rendezvous protocol case in place, in which the communication will continue by the receiver using the RTS information and performing an RDMA Read. This leaves no chance for communication deadlock for any messages.

Figure 12 shows a coarse view of the Rendezvous protocol flow at the sender and receiver sides. It can be seen that there might be a potential for a deadlock in the receiver-side protocol flow, implying that the receiver can be trapped in a cycle without making progress. However, at the sender side, we do not see such a cycle. The sender will either end up with transferring data using RDMA Write that proceeds with the communication, or with sending an RTS that breaks the receiver side cycle. Therefore, in either case, the communication will successfully complete.

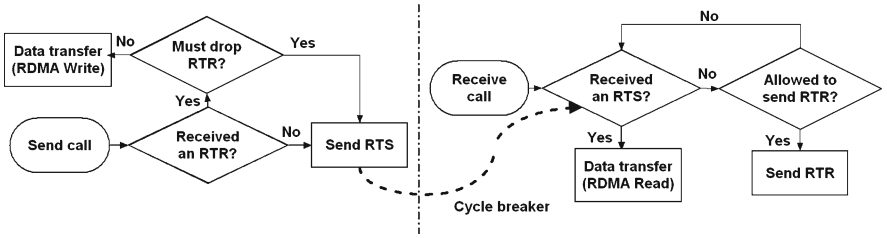

Fig. 12 Deadlock freedom in send/receive Rendezvous protocol flow

3.5 Protocol Usability

Obviously, only applications that employ non-blocking calls and use the Rendezvous protocol for communication will benefit from the improvements made by this protocol. In addition, the benefit is mostly in cases where the non-blocking receive calls are posted earlier than their peer send calls. Since the new protocol tries to initiate the communication using the early non-blocking calls rather than the MPI wait calls, appropriately interleaving communication calls (i.e. non-blocking send and receive and MPI wait calls) with computation phases will maximize the gain by increasing the computation/communication overlap. On the other hand, there are some cases that an application may not benefit from the proposed protocol and may even suffer from its overhead:

- If the application timing and synchronization is such that the receive calls are rarely invoked before their peer send calls, the proposed protocol will not help the application's communication progress much. In such a case, the application performance may even suffer from the protocol's speculation overheads (e.g. race and deadlock prevention, acknowledgement messages, etc.).
- In some applications, due to very tight synchronizations, it is possible that the peer send and receive calls arrive almost at the same time, and therefore both RTR and RTS messages are generated. These messages will then cross each other, and based on the protocol design the RTR will be dropped. While an application with such a timing does not need the RTRs, the overhead of RTR handling is inevitable.
- When an application does not overlap its computation phases with communications, any effort to improve MPI library for communication progress and overlap is useless, and in fact it may adversely affect the application performance due to implementation overheads. It should be noted though that the discussion of the utilization of overlap in modern scientific applications is beyond the scope of this research.

3.6 Protocol Adaptation

To reduce the amount of overhead imposed on applications that do not benefit from the new protocol, an adaptation mechanism is embedded in the proposed protocol to stop the speculation and revert the protocol to the current RDMA-Read based protocol. Basically, there are two adaptation techniques: *static* and *dynamic*. In the static adaptation, the execution of an application is profiled, and based on the profiling outcome one can determine whether the new protocol is useful or not. We can permanently disable the speculation for such an application if the protocol is not useful. In dynamic adaptation methods, we monitor the behavior of the speculative protocol during the application runtime, and dynamically disable the speculation when it is not useful.

In this paper, we propose a method for dynamic adaptation. We opt for a metric that could reflect the behavior of the speculative protocol and its likely impact on application performance. The new protocol generates an RTR message when a Rendezvous protocol is predicted. However, when the prediction is incorrect, or when an RTR message is crossing a peer RTS message, the transferred RTR is discarded

and consequently the system resources are wasted for its generation and transmission. Therefore, we use the *RTR usage factor* (the percentage of the received RTR messages that have been used by the sender) in our study, as it is a metric that shows how accurate and useful the speculation is. The adaptation mechanism compares the RTR usage factor for each message envelope with a threshold value, and stops the speculation if the RTR usage is lower than the threshold. Here is a complete description of the algorithm:

- Each process monitors the RTR generation and usage statistics. A search table (a hash table for large systems and a linear queue for small systems) is used to store the RTR statistics for each message envelope.
- Before preparing and sending an RTS, the sender will check the RTR usage factor. If it is lower than a certain threshold, the speculation is stopped. In addition, as no RTR will be generated the send-side initial progress engine call will also be stopped. A flag will be piggy-backed on the outgoing RTS to inform the receiver about the decision, and so the RTR generation for that message envelope will be stopped at the receiver.

4 Experimental Results and Analysis

We have implemented the proposed Rendezvous protocol on MPICH2 over NetEffect 10-Gigabit iWARP Ethernet. We use micro-benchmarks, some applications from NPB [13], and the RADIX application [14] to evaluate the proposed protocol's ability for overlap and independent progress in comparison to the current MPI Rendezvous protocol. In Sect. 4.1, we describe our experimental platform. Section 4.2 discusses the micro-benchmark results. Section 4.3 presents and analyzes the application results, and Sect. 4.4 covers the overhead of the proposed protocol.

4.1 Experimental Framework

The experiments were conducted on four Dell PowerEdge 2850 SMP servers, each with two 2.8 GHz Intel Xeon processors (with 1 MB L2 cache) and 2 GB of DDR-2 SDRAM. The machines run Linux Fedora Core 4 SMP with kernel version 2.6.11. Our iWARP network consists of the NetEffect NE020 10-Gigabit Ethernet RNICs [24], each with a PCI-Express x8 interface and CX-4 board connectivity. A Fujitsu XG700-CX4 12-port 10-Gigabit Ethernet switch connects the nodes together. We use MPICH2-iWARP, based on MPICH2 1.0.3 [10] over NetEffect verbs 1.4.3. MPICH2-iWARP uses the Rendezvous protocol for messages larger than 128 KB.

4.2 Micro-benchmark Results

In this section, we use micro-benchmarks to evaluate how the new protocol may affect overlap and communication progress. The micro-benchmarks are similar to what is described in [5]. However, we have revised these micro-benchmarks so that we can run them in two timing scenarios: when the sender is always forced to arrive earlier (to

use RTS), and when the receiver is forced to arrive earlier (to use RTR). Assessment is done at both sender and receiver sides.

4.2.1 Receiver Side Overlap and Progress

Enhancing the receiver side Rendezvous overlap and progress ability has been the main goal of this work. Figures 13 and 14 compare the overlap and progress ability of the proposed Rendezvous protocol with the current protocol for the two timing scenarios. The performance results show that the new protocol has been highly successful in its objectives. Note that Fig. 14 presents the results only for 1 MB messages. Similar results have been observed for other long messages.

Improving receiver side overlap ability from less than 10% to more than 80% (more than 90% for most message sizes) in both timing scenarios can be clearly observed in Fig. 13. In our judgment, this is a significant achievement. In the case that the sender arrives first, the initial progress engine call at the receiver side has helped to find the already arrived RTS message. Thus, we have now achieved the expected level of receiver side overlap and progress ability. The overlap is more than 80% and the progress benchmark latencies are not affected by the inserted delay, confirming a complete and independent progress.

The main achievement of the proposed Rendezvous protocol is for the other scenario, in which the receiver arrives earlier. Since the early-arrival receiver speculatively sends an RTR message, the late-arrival sender will find the RTR and start the communication (unlike the current protocol where the early-arrival receiver would start

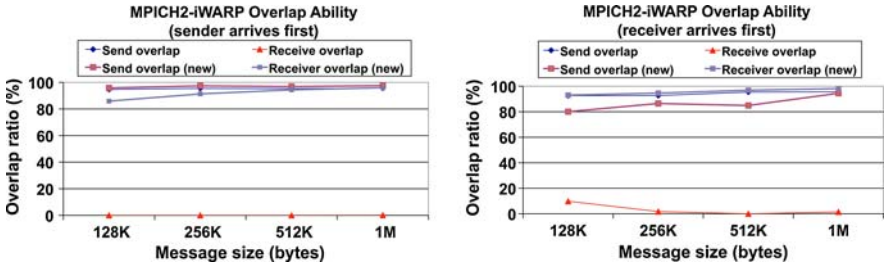

Fig. 13 Current and new Rendezvous overlap ability for two timing scenarios

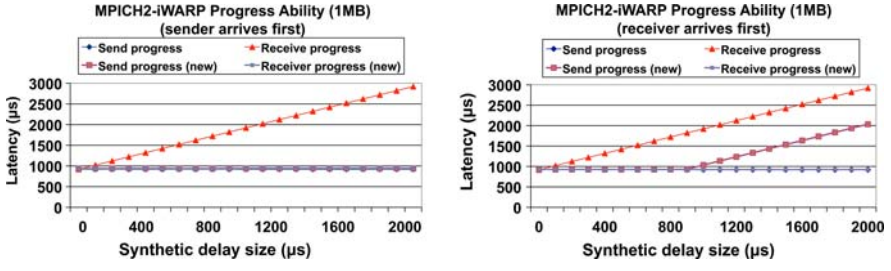

Fig. 14 Current and new Rendezvous progress ability for two timing scenarios

communication when it receives the RTS from the sender after its computation phase). Therefore, complete progress and almost full (more than 92%) overlap is achieved during the computation phase.

4.2.2 Sender Side Overlap and Progress

Based on the protocol details, as we have targeted the receiver side overlap and progress, we do not expect a major change in the send side overlap or progress ability. The results in Figs. 13 and 14 for the send overlap and progress comply with our expectations.

Comparing the current and the new send overlap results, we observe that except for some message sizes when the receiver arrives first, the overlap is almost at the same level or even better. This is a good achievement given our protocol has added some extra overhead at the send side. For the case that the receiver arrives earlier, the sender side overlap has dropped slightly (2–14% based on the message size). This can be due to the required RTR processing in the non-blocking send operation before launching the RDMA Write, as well as some post processing after the delay for sending the done packet.

The send side progress results comply with the overlap observations. When the sender arrives first, the same negotiation scenario occurs as in the current protocol. Therefore, we see similar progress results. However, just like the overlap results for the case that the receiver arrives earlier, the new send side progress is a bit worse. Although the corresponding results presented in Fig. 14 suggest that the latency has been shifted by the inserted delay (after a certain point), examining the details of latency results suggests that only 4–15% of the whole communication latency (depending on the message size) remains after the inserted delay. These results confirm that MPI has been able to make progress in more than 85% of the communication, which is in harmony with the corresponding overlap results in Fig. 13. The reason behind this can be attributed to the fact that we measure the latency at the receiver side to make sure the message has completely arrived. When the receiver arrives earlier and sends an RTR, the sender will start the RDMA Write after finding the RTR. Thus, the RDMA Write will be overlapped with the sender-side synthetic delay (inserted after the non-blocking call). However, the receiver does not finish the receive operation until receiving a done packet from the sender, which is sent after the delay. This affects the latency in the cases that the inserted delay is more than the message latency, which can clearly be observed in Fig. 14.

4.2.3 Exchange Model Micro-benchmarks

In this section, we evaluate the effect of our proposed protocol on a data exchange computation model that is used in many scientific applications. In each iteration of this model, two (or more) processes produce their data, and then exchange the data with the designated neighbors (one or two neighbors in our implementation), and finally consume the exchanged data. The pseudo-code for the exchange model micro-benchmarks is shown in Fig. 15. In this code, the computation phases are interleaved with the non-blocking communication calls in order to maximize the communication/computation

Exchange model 1	Exchange model 2
Timer_start *Produce* ($data_1$) *Loop i =1 to iterations* *MPI_Irecv* ($data_i$) *Pack* ($data_i$) *MPI_Isend* ($data_i$) *Produce* ($data_{i+1}$) *Wait (i, Isend)* *Wait (i, Irecv)* *Consume* ($data_i$) *End Loop* *Timer_stop*	*Timer_start* *Produce* ($data_1$) *Loop i =1 to iterations* *Pack* ($data_i$) *MPI_Isend* ($data_i$) *If(i >1) Consume* ($data_{i-1}$) *MPI_Irecv* ($data_i$) *Produce* ($data_{i+1}$) *Wait (i, Isend)* *Wait (i, Irecv)* *End Loop* *Consume* ($data_{iterations}$) *Timer_stop*

Fig. 15 Psuedo-code for data-exchange micro-benchmarks

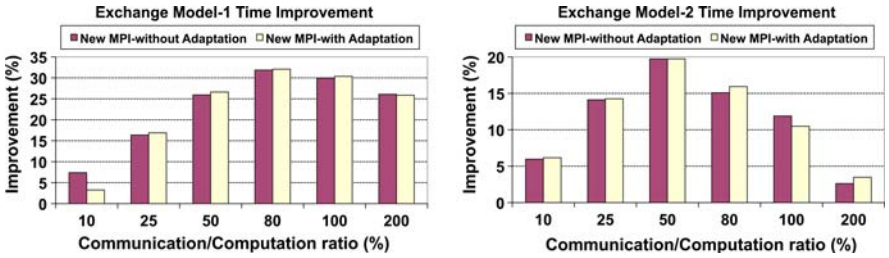

Fig. 16 Exchange model micro-benchmarks: improvement over current rendezvous protocol

overlap at the MPI code level. Two different code alternatives are used to examine the effect of the new protocol when the order of send and receive calls is changed. In the pseudo-codes, $Pack(data_i)$ represents the pre-processing of data usually performed in some applications to pack non-contiguous data buffers to be sent together to the receiver.

We have measured the iteration time when using the current Rendezvous protocol and compared it to the new protocol with and without using the adaptation mechanism. The results are shown in Fig. 16, for a number of communication/computation ratios. Communication time is calculated as the iteration time when no computation is performed. For the overall computation time, we add up the times associated with the produce and consume operations. For the sake of simplicity, we consider identical produce and consume times in our tests.

As shown in Fig. 16, for the benchmark Model 1 that the non-blocking receive is called before the non-blocking send, due to benchmark timings most of the generated RTRs are used because the RTRs are recognized by the sender before it generates any RTS. This makes the late sender to transfer data using RDMA Write, while the produce operation is in progress, which helps to improve the overlap and reduce the whole iteration time. With small communication time, the overlap opportunity is low and so is the improvement. As the ratio of communication time to computation time

increases, the amount of overlap also increases, leading to up to 32% improvement in the benchmark time. This is because when we increase the communication/computation ratio, in fact a larger ratio of communication (compared to the computation time) is overlapped. On the other hand, with very high communication, the whole computation can be overlapped. However, because most of the iteration time is spent in communication, the latency hiding does not cause significant improvement in the overall benchmark time. Clearly, since in benchmark Model 1 the RTR usage is high, the adaptation method does not take effect and the results for adaptive and non-adaptive cases are very close.

In benchmark Model 2 where the send call is before the receive call, in all cases, the RTS messages are recognized in non-blocking receive calls, because of using early progress engine call inside the non-blocking receive operations. Therefore, data transfer is started before the produce phase and the communication is overlapped with computation. However, with different communication times, different improvement ratios are obtained. With similar reasons discussed for the exchange Model 1, the improvement is low when communication/computation ratio is either very high or very low. The maximum improvement is about 20% when the communication time is 50% of the computation time.

Although in both cases we are using a fixed 128 KB message size, in benchmark Model 2 the maximum overlap occurs in a lower communication/computation ratio (50% compared to 80% in benchmark Model 1). The reason is that in Model 2, more computation is overlapped with the communication phase due to the fact that the data transfer from both sides is performed using RDMA Read (initiated using RTS messages). On the other hand, in most of Model 1 iterations, RTRs are used and therefore data is transferred using RDMA Write. Thus, with the same message size, in Model 2 a longer communication phase (RDMA Read compared to RDMA Write) is available. Moreover, because we use the whole pre-measured communication time to calculate the communication/computation ratio, the maximum improvement occurs in a smaller communication/computation ratio, compared to Model 1.

4.3 Application Results

In the previous section, we presented and analyzed the behavior of our proposed Rendezvous protocol with a number of micro-benchmarks, including the exchange model benchmarks that can represent the communication core of some scientific applications. To see the effect of the proposed protocol on actual applications, in this section we elaborate on our experimentation with CG, BT, and LU applications from the NPB benchmark suite [13], and the RADIX integer sorting application [14]. We have selected RADIX, and those applications in NPB that utilize MPI non-blocking calls, except for the SP benchmark that has a similar communication and computation pattern with BT. These applications all have different communication characteristics.

We have run the NAS benchmarks with class A and B, and the RADIX application with a larger workload to fit the messages into the Rendezvous range, on our 4-node cluster and measured the overall application wait time (including *MPI_Wait, MPI_Waitall* and *MPI_Waitany*) for the current protocol and the proposed Rendezvous

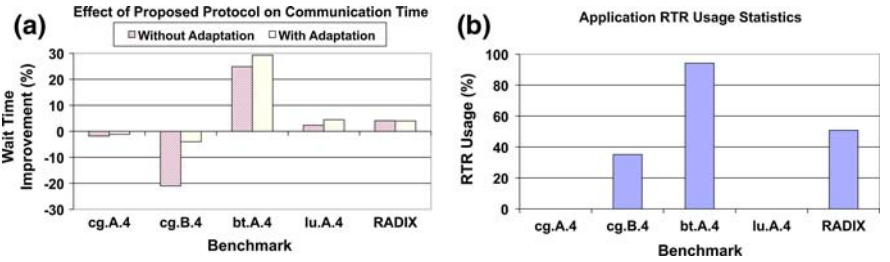

Fig. 17 Effect of the proposed Rendezvous protocol on applications: **a** wait time improvement, **b** global RTR usage

protocol (with and without adaptation). For the adaptation, we have used 80% RTR usage as the threshold for stopping/resuming RTR generation. Figure 17a shows the improvement that the new protocol has made to the communication time of each application. Figure 17b shows the global RTR usage percentage for each application. The results presented are the average of five runs.

As shown in Fig. 17, the communication time in some applications can significantly benefit from the protocol, while in others the wait time may increase to a great extent when adaptation is not used. For example, up to 30% reduction in the overall wait time for BT.A benchmark can be observed. BT.A heavily uses the Rendezvous protocol with a very high RTR usage percentage. Adaptation slightly affects the BT.A benchmark. However, LU.A is a benchmark that cannot use the proposed protocol because it is using *MPI_ANY_SOURCE* in its receive calls. The slight (∼5%) improvement seen for LU.A is mostly due to the impact of the early progress engine call. The RADIX application is also gaining around 4% improvement, mostly because of more overlapping communication processing in back-to-back communication calls, which is due to using both RTR and initial progress engine call in Rendezvous communications.

On the other hand, CG is negatively affected by the proposed protocol. CG.A, which does not have any Rendezvous communication is just affected by a small overhead (less than 2%). On the other hand, CG.B that heavily uses the Rendezvous protocol is affected by 21% increase in wait time, when adaptation is not in use. Because the RTR usage percentage of CG.B falls below the threshold, the adaptation switches to the old Rendezvous protocol, incurring only 4% overhead.

Characteristics of the applications under study are the main reasons behind the reported results. Figure 18 briefly illustrates the computation/communication pattern for part of each application that is involved in non-blocking communications. For the CG.B benchmark, the code does not have any computation interleaved with the non-blocking communication. Therefore, there is not much opportunity for the application to benefit from the receiver-initiated Rendezvous protocol. This confirms the results presented in Fig. 17.

For BT.A benchmark, in each iteration, producing data for a non-blocking send call is overlapped with a non-blocking receive call. Along with LU.A, BT.A has a communication pattern that overlaps with computation. Therefore, similar to what we observed in Sect. 4.2.3 with the exchange model benchmark, if the receive call is issued earlier (in the new protocol) it will initiate the Rendezvous (using RTR) and the non-blocking send call at the other side will start the data transfer. On the other

CG.B benchmark	BT.A benchmark	LU.A benchmark	RADIX application
Produce Loop i = 1 to #neighbors MPI_Irecv (from neighbor i) MPI_Send (to neighbor i) MPI_Wait () End Loop i Consume	Produce ($data_0$) MPI_Isend ($data_0$) Loop i = 1 to iterations MPI_Irecv ($data_i$) Produce ($data_i$) MPI_Wait (i-1, Isend) MPI_Wait (i, Irecv) Consume ($data_i$) MPI_Isend ($data_i$) End Loop i	Loop i = 1 to #neighbors MPI_Irecv (any_source) Produce ($data_i$) MPI_Send (neighbor) MPI_Wait (i, Irecv) Consume ($data_i$) End Loop i	Loop i =1 to iterations Loop j = 1 to 4 MPI_Irecv ($data_i$, src) End Loop j MPI_Waitall () End Loop i Loop i = 1 to iterations MPI_Send (dest) End Loop i

Fig. 18 NAS benchmarks and RADIX non-blocking communication patterns

hand, using the current Rendezvous protocol, the early receive call should wait until the MPI wait call to initiate the data transfer.

For the LU.A benchmark, despite interleaving computation with non-blocking calls, speculation cannot take place simply because the receiver does not know its peer process (recall the use of *MPI_ANY_SOURCE* in the receive calls).

The RADIX application has a high communication-to-computation ratio that is independent of the problem size and the number of processes [14]. Communication is performed in the form of a number of non-blocking receive calls, each four of them followed by an *MPI_Waitall* call, and then followed by a number of *MPI_Send* calls. In some cases, the send phase precedes the receive phase. A simplified form of its communication pattern is shown in Fig. 18.

4.4 Communication Latency and Volume Overhead

As discussed earlier, the proposed Rendezvous protocol has some overhead associated with its design, especially when the adaptation mechanism is not in effect. Certain parts of the protocol that are used for race hazard prevention are the main sources of overhead. Examples are RTR and RTS acknowledgements, Eager-flag manipulation and extra Recvq searches. Other sources of overhead are the dropped RTR messages that will consume processing and communication resources at both ends of the network.

In this section, we show the overhead of the new protocol on MPI message latency. We also present the amount of communication volume overhead for the applications, due to exchanging the newly introduced negotiation packets (RTR and ACK messages). For the message latency overhead, we have used a basic ping-pong benchmark, with a timing that all Rendezvous receive calls will produce RTRs, but the RTRs will be dropped due to crossing with simultaneously generated RTSs from the peer process. The results for this benchmark are shown in Fig. 19a, presenting the overhead with and without the adaptation mechanism. No remarkable overhead (less than 1%) is observed for Eager messages, while a small overhead (less than 3%) exists for Rendezvous messages, only if the adaptation is not in effect.

For the communication volume overhead, we count the amount of extra data generated due to the introduction of RTR and acknowledgement packets. We consider all dropped (unused) RTR messages and also all RTR-ACK and RTS-ACK messages as the sources of the new protocol's overhead. For each benchmark, we report the ratio of the extra data volume to the total application message volume. As shown in Fig 19b, less than 0.04% extra data is generated in the worst case among these benchmarks.

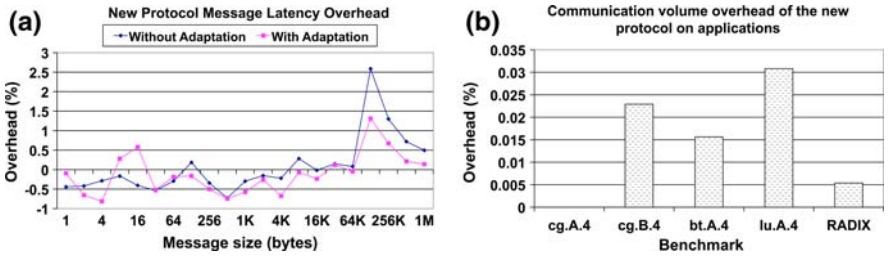

Fig. 19 Overhead of the proposed protocol: **a** latency overhead, **b** communication volume overhead

These results confirm that the new protocol introduces a low level of overhead on latency and data volume of the MPI applications.

5 Conclusions

In this work, we analyzed the overlap and communication progress ability and shortcomings of a current polling and RDMA-Read based MPI Rendezvous protocol on top of RDMA-enabled interconnects. To address its shortcomings, we proposed a novel speculative MPI Rendezvous protocol for RDMA-enabled interconnects and implemented it on MPICH2 over NetEffect 10-Gigabit iWARP Ethernet. Our experimental results show that the new protocol is able to effectively improve the receiver side progress and overlap ability from almost zero to nearly 100%, at the expense of only 2–14% degradation for the send side overlap and progress, and less than 3% latency overhead, when the receiver arrives first. Our implementation has also been able to improve the receiver-side overlap for more than 85%, and also make the receiver-side progress independent without reducing the sender-side overlap and progress ability. We also evaluated our proposal using exchange model micro-benchmarks. The evaluation confirms that the new protocol can have a significant improvement for applications with this kind of core computation model. Obviously, the amount of improvement depends on the ratio of communication over computation.

At the application level, we have compared the results using four NPB benchmarks and the RADIX application. The results show up to 30% improvement in the application wait time. The results also showed that the protocol may impose overhead on some applications due to their communication patterns and timing. To address this issue, we introduced a dynamic adaptation mechanism to switch back to the current Rendezvous protocol when the RTR usage statistics fall below a certain threshold. The adaptation mechanism has been quite successful in minimizing the amount of overhead on the applications.

Last but not least, we would like to add that although we used the 10-Gigabit iWARP Ethernet as the interconnect in this study, we believe the proposed MPI Rendezvous protocol has implications beyond this network, and can be applied directly to any other RDMA-enabled networks such as InfiniBand.

Acknowledgements This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through grant RGPIN/238964-2005, Canada Foundation for Innovation (CFI), grant

#7154, and Ontario Innovation Trust (OIT), grant #7154. The Authors would like to thank NetEffect Inc., for the resources and their technical support.

References

1. MPI: A Message-Passing Interface standard, MPI Forum (1997)
2. Goumas, G., Sotiropoulos, A., Koziris, N.: Minimizing completion time for loop tiling with computation and communication overlapping. In: Proceedings of 15th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS'01) (2001). doi:10.1109/IPDPS.2001.924976
3. Fishgold, L., Danalis, A., Pollock, L., Swany, M.: An automated approach to improve communication-computation overlap in clusters. In: 2006 NSF Next Generation Software Workshop (NSFNGS 2006). Proceedings of 20th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS'06) (2006). doi:10.1109/IPDPS.2006.1639590
4. Brightwell, R., Riesen, R., Underwood, K.D.: Analyzing the impact of overlap, offload, and independent progress for Message Passing Interface applications. Int. J. High Perform. Comput. Appl. **19**(2), 103–117 (2005). doi:10.1177/1094342005054257
5. Rashti, M.J., Afsahi, A.: Assessing the ability of computation/communication overlap and communication progress in modern interconnects. In: Proceedings of 15th Annual IEEE Symposium on High-Performance Interconnects (Hot Interconnects 2007), pp. 117–124 (2007). doi:10.1109/HOTI.2007.12
6. Wagner, A., Jin, H., Panda, D.K., Riesen, R.: NIC-based offload of dynamic user-defined modules for Myrinet clusters. In: Proceedings of 6th IEEE International Conference on Cluster Computing (Cluster'04), pp. 205–214 (2004). doi:10.1109/CLUSTR.2004.1392618
7. Sitsky, D., Hayashi, K.: An MPI library which uses polling, interrupts and remote copying for the Fujitsu AP1000+. In: Proceedings of International Symposium on Parallel Architectures, Algorithms, and Networks, pp. 43–49 (1996). doi:10.1109/ISPAN.1996.508959
8. 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 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2006), pp. 32–39 (2006). doi:10.1145/1122971.1122978
9. Trahay, F., Denis, A., Aumage, O., Namyst, R.: Improving reactivity and communication overlap in MPI using a generic I/O manager. In: Proceedings of Euro PVM/MPI 2007, LNCS 4757, pp. 170–177 (2007)
10. MPICH2: http://www-unix.mcs.anl.gov/mpi/mpich2/
11. Rashti, M.J., Afsahi, A.: 10-Gigabit iWARP Ethernet: comparative performance analysis with InfiniBand and Myrinet-10G. In: 7th IEEE Workshop on Communication Architecture for Clusters (CAC'07). Proceedings of 21st IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS'07) (2007). doi:10.1109/IPDPS.2007.370480
12. Rashti, M.J., Afsahi, A.: Improving communication progress and overlap in MPI Rendezvous protocol over RDMA-enabled interconnects. In: Proceedings of 22nd International Symposium on High Performance Computing Systems and Applications (HPCS 2008), pp. 95–101 (2008). doi:10.1109/HPCS.2008.10
13. National Aeronautics and Space Administration (NASA): NAS Parallel Benchmarks (NPB) for MPI, http://www.nas.nasa.gov/Resources/Software/npb.html/
14. Shan, H., Singh, J.P., Oliker, L., Biswas, R.: Message passing and shared address space parallelism on an SMP cluster. J. Parallel Comput. **29**(2), 167–186 (2003). doi:10.1016/S0167-8191(02)00222-3
15. Petrini, F., Coll, S., Frachtenberg, E., Hoisie, A.: Performance evaluation of the Quadrics interconnection network. J. Cluster Comput. **6**(2), 125–142 (2003). doi:10.1023/A:1022852505633
16. Brightwell, R., Doerfler, D., Underwood, K.D.: A comparison of 4X InfiniBand and Quadrics elan-4 technologies. In: Proceedings of 6th IEEE International Conference on Cluster Computing (Cluster'04), pp. 193–204 (2004). doi:10.1109/CLUSTR.2004.1392617
17. InfiniBand Trade Association, InfiniBand Architecture Specification, vol. 1, October (2004)
18. Beecroft, J., Addison, D., Hewson, D., McLaren, M., Roweth, D., Petrini, F., Nieplocha, J.: QsNetII: Defining high-performance network design. IEEE Micro **25**(4), 34–47 (2005). doi:10.1109/MM.2005.75
19. Doerfler, D., Brightwell, R.: Measuring MPI send and receive overhead and application availability in high performance network interfaces. In: Proceedings of EuroPVM/MPI 2006, LNCS 4192, pp. 331–338 (2006)

20. Liu, J., Chandrasekaran, B., Wu, J., Jiang, W., Kini, S., Yu, W., Buntinas, D., Wyckoff, P., Panda, D.K.: Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics. In: Proceedings of 2003 ACM/IEEE Conference on Supercomputing (SC 2003) (2003). doi:10.1109/SC.2003.10007
21. Zamani, R., Qian, Y., Afsahi, A.: An evaluation of the Myrinet/GM2 two-port networks. In: 3rd IEEE Workshop on High-Speed Local Networks (HSLN 2004). Proceedings of 2004 International Conference on Local Area Networks (LCN 2004), pp. 734–742 (2004). doi:10.1109/LCN.2004.20
22. Mellanox Technologies, Inc.: http://www.mellanox.com/
23. Myricom. http://www.myricom.com/
24. NetEffect, Inc.: NetEffect NE020 10Gb iWARP Ethernet channel adapter. http://www.neteffect.com/
25. RDMA Consortium: iWARP protocol specification, http://www.rdmaconsortium.org/
26. Amerson, G., Apon, A.: Implementation and design analysis of a network messaging module using virtual interface architecture. In: Proceedings of 6th IEEE International Conference on Cluster Computing (Cluster'04), pp. 255–265 (2004). doi:10.1109/CLUSTR.2004.1392623
27. MVAPICH: http://mvapich.cse.ohio-state.edu/index.shtml/
28. Kumar, R., Mamidala, A.R., Koop, M.J., Santhanaraman, G., Panda, D.K.: Lock-free asynchronous Rendezvous design for MPI point-to-point communication. In: Proceedings of EuroPVM/MPI 2008, LNCS 5205, pp. 185–193 (2008)
29. Pakin, S.: Receiver-initiated message passing over RDMA networks. In: Proceedings of 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008) (2008). doi:10.1109/IPDPS.2008.4536262
30. Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell Broadband Engine architecture and its first implementation—a performance view. IBM J. Res. Develop. **51**(5), 559–572 (2007)