Gravel: A Communication Library to Fast Path MPI

Anthony Danalis, Aaron Brown, Lori Pollock, Martin Swany, and John Cavazos

Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716
{danalis,brown,pollock,swany,cavazos}@cis.udel.edu

Abstract. Remote Direct Memory Access (RDMA) technology allows data to move from the memory of one system into another system's memory without involving either one's CPU. This capability enables communication-computation overlapping, which is highly desirable for addressing the costly communication overhead in cluster computing. This paper describes the consumer-initiated and producer-initiated protocols of a companion library for MPI called Gravel. Gravel works in concert with MPI to achieve increased communication-computation overlap by separating the meta-data exchange from the application data exchange, thus allowing different communication protocols to be implemented at the application layer. We demonstrate performance improvements using Gravel for a set of communication patterns commonly found in MPI scientific applications.

1 Introduction

The communication overhead of cluster computing continues to challenge MPI programmers trying to maximize the performance of their applications. Remote Direct Memory Access (RDMA) technology holds the promise of hiding these overheads by facilitating the overlap of communication operations with computation. To exploit the RDMA for communication-computation overlap, the communication library must provide support for one-sided communication and two-sided communication with low-overhead rendezvous protocols, and the application must contain communication and computation patterns that are amenable to overlap.

There already exist communication libraries that provide for asynchronous communication and have the goal of exploiting RDMA support; however, none provides the set of features that the proposed library, *Gravel*, provides. MPI provides asynchronous communication operations (e.g., Isend, Irecv, and Wait) and even one-sided communication support, although it enforces strict rules regarding the use of the latter. The User Direct Access Programming Library, uDAPL [1], provides functionality necessary to enable RDMA from applications, with support for memory registration and connection establishment, but does not provide a "message" abstraction nor does it provide a high level and intuitive interface for domain scientists and engineers to embrace. ARMCI [3] aims to be a portable library for RDMA communication. However, it requires the use of a custom memory allocator, which makes it unsuitable for substituting arbitrary MPI operations in Fortran applications without major restructuring of the application buffers. GASNet [4] provides similar capabilities and is intended to be used internally by a compiler or transformation system, but as its name implies, is targeted to Global Address Space parallel languages. Other communication libraries are provided by the hardware vendor as a

A. Lastovetsky et al. (Eds.): EuroPVM/MPI 2008, LNCS 5205, pp. 111–119, 2008.

means of implementing higher-level libraries, such as MPI, for the given interconnect (e.g., VAPI [5], GM [2]), or are usable only on specific hardware interconnects (e.g., MX [6]). Additionally, most of these libraries require either C pointer manipulation, or the use of memory returned by C library functions (i.e., `lib_specific_malloc()`), both of which are not possible directly from within FORTRAN programs.

In contrast to these approaches, our goal was to design a communication library that provided minimal messaging protocol, maximal overlap potential, support for Fortran and support for further communication optimization through code motion and transformation. Fortran support implies several requirements, most notably explicit memory registration functionality rather than special memory allocation routines. This paper describes a portable communication library, *Gravel*, which works in conjunction with MPI and is designed to (1) replace only key data exchange calls in MPI programs (2) separate the meta-data exchange from the application data exchange, and (3) improve the potential of code motion to increase the communication-computation overlapping and hide communication latency.

2 Communication Library

Gravel is a minimal library designed to be used in conjunction with MPI by replacing only key data exchange calls in MPI programs to exploit the potential communication-computation overlap in applications. It implements a simple messaging abstraction designed for RDMA-capable networks. The Gravel system is currently built on top the uDAPL [1] in the OpenFabrics software suite. However, it is not dependent on uDAPL, therefore can be ported to additional network interconnects.

Gravel is neither aimed to be a replacement for MPI, nor a low level library that one should use for implementing MPI. Rather, Gravel is designed to be used in MPI applications to improve performance by replacing selected MPI calls. While systems like ARMCI provide synchronization mechanisms, Gravel relies on those provided by MPI. Gravel's API is designed to be similar to that of MPI to facilitate automatic replacement (e.g., within a compiler) of performance-critical MPI calls with Gravel calls. Gravel places a number of restrictions on when it can be used. Gravel never copies messages, instead all message exchanges are "rendezvous" style, in MPI parlance. Further, as is the case with all RDMA layers, memory must be registered to be a source or destination for messages.

Gravel provides minimal RDMA-based messaging functionality that is decomposed to maximize potential for overlapping communication with computation. In general, a message can be seen as the exchange of message metadata (e.g., a message header) and the exchange of the data itself, followed by some indication of completion. Gravel separates this functionality into independent parts by providing distinct functions for implementing explicit registration of memory, communication rendezvous (exchanging message metadata), and performing the actual data transfer.

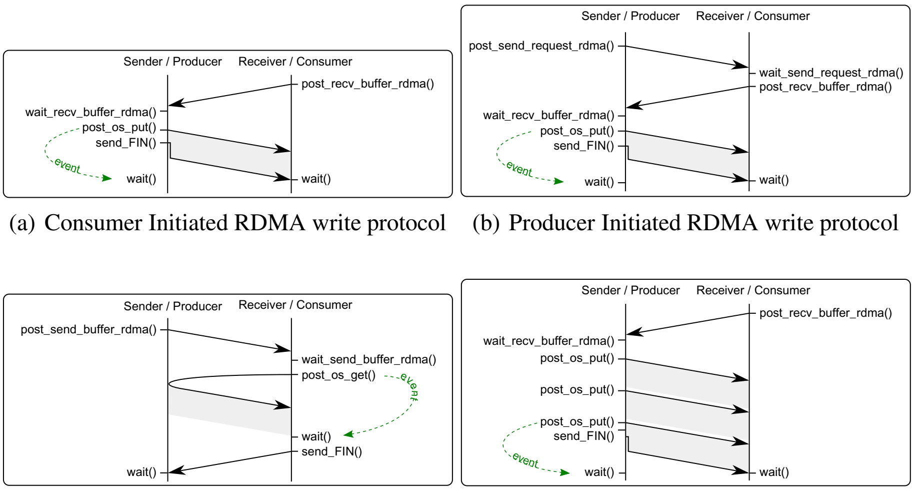
2.1 Rendezvous Protocols

Sur et al. [7] discuss the behavior and performance of the simple rendezvous found in a typical MPI implementation. They present a more advanced alternative which they implemented within *MVAPICH*, the Ohio State University implementation of MPI. Both rendezvous protocols are "sender initiated". The legacy protocol uses `RDMA-write` for the transfer, and the advanced protocol uses `RDMA-read` to eliminate unnecessary round-trip delays. Shipman et al. [8] also discuss advanced protocols used in MPI, but for the OpenMPI implementation of the MPI standard.

Gravel provides distinct functions for transferring data and metadata in order to allow an automatic program transformation tool (e.g., a compiler) or programmer to use the most efficient model given the constraints of the application. Efficiency in this case comes from appropriate overlapping of orthogonal computation.

Unlike highly abstracted libraries such as MPI, Gravel does not provide fixed data exchange protocols hidden inside calls such as `MPI_Isend`. It rather provides the necessary API and infrastructure for implementing any protocol at the application layer. Details are hidden behind the library API, so that the application does not need to specify anything other than which task should do what and when. By doing so, Gravel enables "function separation", i.e., the handshake is separated from the data transfer. As a result, each application can implement the appropriate exchange protocols that best fit the structure of each data exchange in that application. At the same time, the high-level, abstract API enables programmers to easily implement highly efficient exchange protocols, even in languages such as FORTRAN that does not support C pointers (pointers to arbitrary memory locations, or pointers returned by C library functions). Four protocols that can be implemented with Gravel are presented in Figure 1 and described below. In the following text, we use the term "producer" for the node that will send the **application data** message, and "consumer" for the node that will receive the **application data** message.

Consumer Initiated RDMA-Write Protocol. In this case (shown in Figure 1(a)), the consumer initiates the handshake (`post_receive_buffer_rdma()`) by sending a



(a) Consumer Initiated RDMA write protocol (b) Producer Initiated RDMA write protocol

(c) Producer Initiated RDMA read protocol (d) Advanced RDMA write based protocol

Fig. 1. Timing schematics of `RDMA_write` and `RDMA_read` based rendezvous protocols

metadata message to a predefined location in the producer's memory (initialized by `gravel_init()`) referred to as the *receive-info* ledger. The details concerning the ledger are hidden from the application and are automatically handled by Gravel. Thus, the application does not need to allocate the ledger, register it, exchange its address between all the peers, etc. Furthermore, an application programmer, or compiler that uses Gravel does not even need to know that there is such an entity as a ledger. With this handshake message, the consumer passes to the producer the start of the local memory where the data will be received (receive buffer), the size of the expected data, an application defined tag, and a request handle to the call. After initiating the non-blocking transfer of the handshake, the consumer can proceed with independent computation, but it must assume that the receive buffer can be altered at any time between this call and the return of the corresponding wait operation.

When the producer is ready to post a send, it reads the next metadata message from its *receive-info* ledger (or blocks until the metadata message from the consumer arrives). The producer then initiates a non-blocking `RDMA-write` (`post_os_put()`) operation to transfer the message data to the consumer, followed by an additional non-blocking `RDMA-write` to send a small metadata message (into the *RDMA-FIN* ledger)[1]. Then, the producer can proceed with independent computation. Finally, both sides will wait for the completion of the transfer.

This protocol is well-suited for cases when the receive operation can be posted early because the computation does not have data dependencies with the receive buffer. In this scenario, the handshake overhead will be entirely overlapped with the independent computation and the actual data transfer will be performed by an RDMA-write operation without any delays or copying, regardless of the size of the transfer, leading to an efficient data exchange.

Producer Initiated RDMA-Write Protocol. Many MPI programs use the wildcard `MPI_ANY_SOURCE` such that any peer could be the producer of the data. Supporting this requires an extended version of the previous protocol, that is producer initiated, as shown in Figure 1(b). In this case, the producer first sends an asynchronous metadata message to the consumer notifying it of an upcoming data send operation. This message is written into the predetermined *send-info* ledger in the consumer's memory.

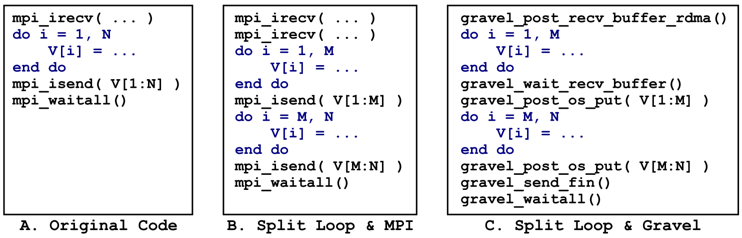
When the consumer is ready to post the receive, it will read the first "send request" from its *send-info* ledger, (or block until at least one peer sends a "send request"). At this point, the consumer can continue with the exact same steps taken by the consumer initiated RDMA-write protocol.

Producer Initiated RDMA-Read Protocol. This protocol (shown in Figure 1(c)) is very different from the previous two protocols. Here, the producer initiates the handshake, but only after the data is ready to be sent to the consumer. The producer sends a small metadata message to a predefined location in the consumer's memory (*send-info* ledger). This message contains the location of the application data in memory (send buffer), the size of the data to be sent, and the tag. Then, the producer can proceed with independent computation and then block, waiting for the completion of the transfer.

[1] If the underlying network does not support message ordering, appropriate measures need to be taken for the FIN to arrive at the consumer after the application data.

On the other side, the consumer reads the next entry from the *send-info* ledger, or blocks waiting for the arrival of a metadata message from the producer. At this point, the consumer initiates the transfer with a non-blocking `RDMA-read` operation. This provides the RDMA engine with the necessary information about the transfer and returns immediately. Thus, the consumer can execute independent computation during the data transfer. Before the consumer can notify the producer about the completion of the transfer, the consumer must wait for the `RDMA-read` to complete. When the transfer is completed, the consumer writes to the *RDMA-FIN* ledger of the producer, signaling the completion of the data transfer. Clearly, the producer must assume that the data is being used at any point between the initiation of the rendezvous and the corresponding `wait()` function, and cannot alter the data.

This protocol is expected to perform less efficiently than the "consumer initiated" protocol described earlier, but is necessary for cases that meet all the following criteria: 1) the communication is symmetric and every node is both consumer and producer, 2) the send operation takes place before the receive operation, and 3) the data or control dependencies prevent the receive operation from being hoisted above the send operation.



```
mpi_irecv( ... )
do i = 1, N
    V[i] = ...
end do
mpi_isend( V[1:N] )
mpi_waitall()
```

A. Original Code

```
mpi_irecv( ... )
mpi_irecv( ... )
do i = 1, M
    V[i] = ...
end do
mpi_isend( V[1:M] )
do i = M, N
    V[i] = ...
end do
mpi_isend( V[M:N] )
mpi_waitall()
```

B. Split Loop & MPI

```
gravel_post_recv_buffer_rdma()
do i = 1, M
    V[i] = ...
end do
gravel_wait_recv_buffer()
gravel_post_os_put( V[1:M] )
do i = M, N
    V[i] = ...
end do
gravel_post_os_put( V[M:N] )
gravel_send_fin()
gravel_waitall()
```

C. Split Loop & Gravel

Fig. 2. Communication overlapping before and after splitting computation

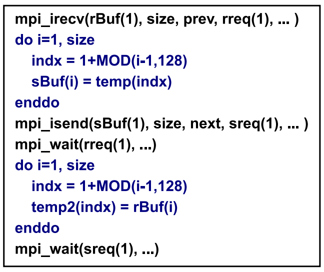
Advanced RDMA-Write Based Protocol. Figure 2(a) shows an example of a parallel program where every task computes, stores into an array `V` and transfers some data to one or more neighbors. This simple case is common and does not provide an opportunity for overlapping the communication generated by the `send` operation with useful computation. However, if there are no dependencies on array `V` throughout the iteration space of the loop, it can be transformed to enable overlapping. In Figure 2(b), we see that the loop is split and therefore the `mpi_isend(V[1:M])` call that transfers the first part of the array can be overlapped with the computation of the second part of the array. Although this could lead to performance benefits, due to overlapping, it also has two major drawbacks. Namely, smaller messages experience lower throughput and every additional message adds contention and overhead through multiple handshakes. The latter concern can be alleviated with Gravel through a more advanced rendezvous scheme, specialized for pipelined transfers. In particular, only one handshake message is necessary, even if the transfer takes place in multiple segments. Indeed as is shown in Figure 2(c) only one call to the metadata transfer functions (`post_recv_buffer()`, `wait_recv_buffer()`, `send_fin()`) is performed, but the data is transferred in two steps by calling `post_os_put()` twice, achieving overlap without the overhead of additional control messages. Clearly, when `post_os_put()` is called multiple times, each call needs to be given an “offset” equal to the cumulative amount of data transferred by the previous calls. Figure 1(d) shows a schematic of a similar communication pattern, where the data transfer takes place in three steps.

This example of an advanced protocol demonstrates the difference between MPI and Gravel. MPI provides implicit protocols hidden behind calls such as `MPI_Isend` and `MPI_Irecv`, designed without any knowledge of a particular application. In contrast, Gravel provides the appropriate API and infrastructure for implementing the exchange protocols at the application layer. This way, the particular characteristics of each application can be exploited for maximizing communication-computation overlap.

3 Experimental Study

Experiment Design. For evaluation, we designed our experiments to explore how a program’s performance is affected when key MPI calls are replaced by the equivalent Gravel library calls and how the previously mentioned different Gravel protocols affect performance.

We experimented with different communication libraries, message sizes, and strip-mining to enable more overlapping. Our experiments were performed on an infiniband cluster with 24 nodes running Linux 2.6.18. We used *mvapich-1.0* and *OpenMPI-1.2.5*[2] implementations built on top of the infiniband layer provided by the OpenFabrics Alliance’s OFED-1.3. We also used our Gravel implementation on top of uDAPL-2.0 which is also provided by OFED-1.3. We experimented with different tuning parameters (`mpi_leave_pinned=1` and `btl_openib_use_eager_rdma` for OpenMPI and `LAZY_MEM_UNREGISTER` for mvapich) to achieve good performance with each MPI implementation. Each benchmark is implemented such that it starts with 1,000 cold runs that are not timed, so that the MPI engine is warmed up and exhibits “steady state” performance before the timing begins. After the timer is started the code segment that performs the computation and communication is also executed 1,000 times so that the timing errors are amortized and anomalous behavior is averaged out.



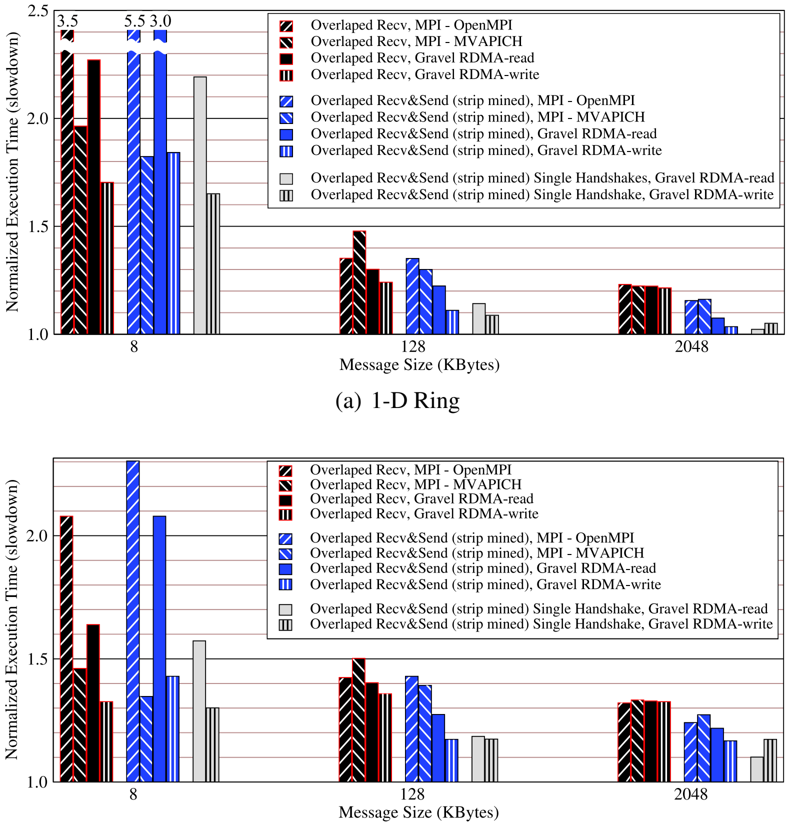
```
mpi_irecv(rBuf(1), size, prev, rreq(1), ... )
do i=1, size
   indx = 1+MOD(i-1,128)
   sBuf(i) = temp(indx)
enddo
mpi_isend(sBuf(1), size, next, sreq(1), ... )
mpi_wait(rreq(1), ...)
do i=1, size
   indx = 1+MOD(i-1,128)
   temp2(indx) = rBuf(i)
enddo
mpi_wait(sreq(1), ...)
```

Fig. 3. 1-D Ring micro-benchmark critical part

[2] We also experimented with the trunk version of OpenMPI-1.3 and found that it performs significantly faster than OpenMPI-1.2.5, but compares with Gravel the same way. Since the trunk does not constitute a stable version of the library that others can use to replicate our experiments, we do not report those results.

We compared the runtime performance of a set of MPI Fortran micro-benchmarks which represent communication-computation patterns found in many scientific applications. We used two micro-benchmarks: (1) a *1D-ring* pattern where every task receives data from the previous one and sends data to the next one and (2) a *2D-wavefront* where the tasks are organized in a 2-D grid and every task receives data from “north” and “west” and sends data to “south” and “east”. The *1D-ring* pattern appears in well known codes including *SP*, *BT* and *LU* in the *NAS* suite; the *2D-wavefront* pattern appears in *LU* of the *NAS* suite and *Sweep3D*. The critical part of each micro-benchmark performs some minimal computation (array to array copy) that fills the message buffer to be sent, transfers the message and then performs some computation (copy) with the message it received from its peer. Figure 3 demonstrates the critical part of the *1-D Ring* micro-benchmark in pseudocode.

Results. Figure 4(a) and Figure 4(b) show the performance results for the *1D-ring* and *2D-wavefront* micro-benchmarks, respectively. The Y axis of both graphs presents execution time normalized to the ideal case, where communication is infinitely fast and



(a) 1-D Ring

(b) 2-D Wavefront

Fig. 4. Experimental evaluation of Gravel and MPI for 1-D Ring and 2-D Wavefront data exchanges

causes no overhead. This case is simulated by a version of the benchmark that does not perform any communication, just the computation loops. All values above 1 designate the slowdown factor caused by the communication. Each micro-benchmark was run several times, and the graph plots the minimum execution time for each scenario. We found very little variation across several runs of the same micro-benchmark using the same configuration. Each graph shows three different clusters of bars corresponding to the three different message sizes we evaluated. Within each cluster, there are three subgroups using dark, medium and light shadings, respectively. For all subgroups, the "computation" is a simple buffer to buffer copy.

The first subgroup to the left end of each bar cluster demonstrates the scenario where the transmission of data is not overlapped with computation and only the `recv` operation has potential for overlapping. Here, we attempt to answer the question of how the program's performance is affected when MPI calls are simply replaced by equivalent Gravel library calls. By looking at the bars within this subgroup and in particular the bar with the vertical stripes representing the non-transformed code using Gravel's consumer initiated RDMA-write protocol, one can see that the execution time of this version is lower than that achieved with either MPI version across sizes and data exchange patterns.

The second subgroup within a cluster demonstrates the scenario where the computation is strip-mined into a double nested loop where the inner loop operates on a section, or tile, of the buffer and the outer loop iterates over the consecutive tiles. In this scenario, the communication is broken into smaller messages and inserted into the outer loop of the loop nest such that after each tile of the buffer is computed, the transfer of that tile is initiated and overlapped with the computation of the next tile. Here, one can see that the transformed versions of the application that use Gravel experienced lower overhead than the transformed versions that use MPI for large message sizes and comparable overhead for small message sizes.

The third subgroup within a cluster demonstrates the scenario where we compare the same overlapped code as the second subgroup, but Gravel's ability to perform several data transfers with only one handshake is utilized, to minimize protocol overhead. This corresponds to the advanced RDMA based protocol shown in Figure 1(d). By comparing the bars of this subgroup with the corresponding medium shaded bars, the reader can see that across message sizes and communication patterns, the advanced Gravel protocols that use a single handshake for multiple data transfers perform better than the MPI-like protocols where every data transfer requires a handshake. For small message sizes, strip-mining to achieve overlapping might cause the application to run slower than the original version (when either MPI or Gravel is used) due to increased protocol overhead and significantly reduced throughput. For larger sizes, when either library is used, overlapping through strip-mining benefits the communication performance.

The results show that for all but the very large sizes, the consumer-initiated RDMA-write based protocol outperforms the producer-initiated RDMA-read based protocol. The reason for the reversal of this behavior witnessed for large sizes will be further investigated in the future. Also, by studying the graphs of Figures 4(a) and 4(b), one can see that for every bar cluster, there is a trend going from more to less overhead as we move from the left-most bar to the right-most bar. This is due to moving from a simpler form of the code to an optimized form as well as moving from pure MPI code, to code that combines MPI and Gravel, and finally to code that combines MPI and an advanced use of Gravel.

4 Conclusions and Future Work

In this paper we presented Gravel, a communication library designed to inter-operate with MPI to fast-path key data transfers of parallel applications. We have described rendezvous protocols that can be implemented at the application layer when using Gravel as opposed to MPI's exchange protocols that are fixed and do not exploit the structure of each particular application. In addition, we have demonstrated the performance improvements that a parallel application can achieve with Gravel through communication-computation overlapping. Currently, we are working on using Gravel with our compiler transformation tool [9], to enable communication optimization of parallel applications without the need for user intervention.

References

1. uDAPL: User Direct Access Programming Library, http://www.datcollaborative.org/uDAPL_doc_062102.pdf
2. GM reference manual, http://www.myri.com/scs/GM/doc/refman.pdf
3. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In: RTSPP IPPS/SDP 1999 (1999)
4. Bonachea, D.: GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley (October 2002)
5. Mellanox Technologies Inc.: Mellanox IB-Verbs API (VAPI) (2001)
6. Myricom Inc.: Myrinet EXpress (MX): A High Performance, Low-level, Message-Passing Interface for Myrinet (2003), http://www.myri.com/scs/
7. Sur, S., Jin, H.W., Chai, L., Panda, D.K.: RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In: PPoPP 2006: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 32–39 (2006)
8. Shipman, G.M., Woodall, T.S., Bosilca, G., Graham, R.L., Maccabe, A.B.: High performance RDMA protocols in HPC. In: Proceedings, 13th European PVM/MPI Users' Group Meeting, Bonn, Germany. LNCS. Springer, Heidelberg (2006)
9. Danalis, A., Pollock, L., Swany, M., Cavazos, J.: Implementing an open64-based tool for improving the performance of mpi programs. In: Open64 Workshop in conjunction with IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Boston, MA (April 2008)