

# Introducing Gravel: An MPI Companion Library

Anthony Danalis, Aaron Brown, Lori Pollock and Martin Swany  
Department of Computer and Information Sciences  
University of Delaware, Newark, DE 19716  
{danalis, brown, pollock, swany}@cis.udel.edu

## Abstract

*A non-trivial challenge in high performance, cluster computing is the communication overhead introduced by the cluster interconnect. A common strategy for addressing this challenge is the use of communication-computation overlapping. In this paper, we introduce Gravel, a portable communication library designed to inter-operate with MPI for improving communication-computation overlapping. Selected MPI calls are semi-automatically replaced by Gravel calls only in key locations in an application where performance is critical. Gravel separates the data transfers from the handshake messages, enabling application developers to utilize Remote Data Memory Access (RDMA) directly, whether the data exchange scheme of their application is pure one-sided, or a more traditional two-sided. The Gravel API is much simpler than existing low level libraries and details (i.e., pointers) are hidden from the application layer so Gravel is usable in FORTRAN applications. This paper presents an overview of Gravel.*

## 1. Introduction

A well known approach to obtaining increased performance of parallel applications in clusters is communication-computation overlapping. The communication library and cluster interconnect technology both play a key role in the achieved performance improvements. There exist many available communication libraries. Some of these libraries (e.g., GASNet [3], ARMCi [8]) are meant to be used by compilers of Global Address Space (GAS) languages. Others are provided by the hardware vendor as a means to implement higher-level libraries, such as MPI, for the given interconnect (e.g., VAPI [6], GM [1]), or are usable only on specific hardware interconnects (e.g., MX [7]). Furthermore, most of these libraries require either C pointer manipulation, or use of memory returned by C library functions

(i.e., `lib_specific_malloc()`), both of which are impossible directly from within FORTRAN programs.

In this paper, we present a new communication library, *Gravel*, aimed at assisting MPI programs to achieve higher performance through improved overlapping of communication and computation. *Gravel* works in conjunction with MPI and is designed to replace only key data exchange calls in MPI programs to increase the communication-computation overlapping and hide communication latency.

*Gravel* is designed to provide the following features:

*Explicit use of Remote Direct Memory Access (RDMA).* *Gravel* provides functions that can initiate an RDMA write, or read operation without setting any restrictions regarding their use. Correctness is handled at the application layer.

*High level API.* *Gravel* does not expose low level information (such as queues, events, special memory regions, etc) to the application layer. Using *Gravel* to perform simple data exchanges is as easy as it is when using MPI.

*Portability beyond a single type of interconnect.* *Gravel* is portable as it does not depend on the special features of any given interconnect technology. It can be implemented over any modern cluster interconnect that supports RDMA operations.

*No hidden performance costs.* *Gravel* does not perform any internal operations (such as queuing and copying unexpected messages) that would lead to performance penalties. All the internal operations of *Gravel* are of book-keeping nature and have constant cost, regardless of the size of the exchange messages.

*Function separation.* *Gravel* provides separate API functions for meta-data exchange and application data exchange. Applications can implement simple, or complex communication protocols such that the data exchange best matches the characteristics of the application.

*Directly usable in FORTRAN applications.* *Gravel* does not

expose any information to the application layer that would require pointers in order to be handled, and *Gravel* does not demand that the exchange messages be stored in memory returned by library functions, since such a demand would make *Gravel* unusable in FORTRAN programs.

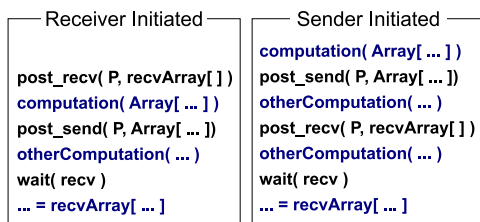
We argue that no existing communication library provides all these features simultaneously. The main goal behind *Gravel* is to fill a gap and complement, or fast-path, MPI by speeding up selected data exchanges in parallel applications that are written using MPI.

## 2. The *Gravel* Communications Library

**Context.** *Gravel* can be utilized in several ways. Our ultimate goal is to integrate it into the bigger, automated optimization system we are developing (ASPhALT). Nevertheless, we developed *Gravel* such that it can be used stand-alone and independently of the rest of the optimization system.

**Overview.** Figure 1 shows two abstract forms of a code segment that can benefit from using *Gravel*. The key characteristic is that the code can interleave computation and asynchronous communication, in order to overlap the data transfer with useful computation and therefore hide the communication latency.

Unlike MPI, *Gravel* uses a different communication mode in the case of a receiver-initiated exchange than in the case of a sender-initiated exchange. In each case, the first call transmits meta-data to its peer, the second call performs the actual data transfer and the `wait` call blocks until the data transfer has completed. In both cases, the achievable overlapping is maximized when there is enough independent computation to overlap both the initial handshake and the actual data transfer.



**Figure 1. Code targeted by *Gravel***

The main differences between MPI and *Gravel* are:

- No copying of messages takes place inside the *Gravel* library. This disables the possibility for an unexpected message queue. Additionally, messages of all sizes are treated the same way requiring a receive buffer to be posted by the application on the consumer side before a transfer can start on the producer side. In other

words, small messages are not exchanged using an eager protocol that would require library-maintained receive buffers.

- Buffers that will be involved in data transfers must be explicitly registered at the application layer before any transfer take place. This way no speculation needs to take place inside the library regarding the frequency of use of different buffers, and no registered-memory cache needs to be maintained.

These *Gravel* design decisions lead to programs that can achieve better performance than their MPI versions, at the cost of not supporting some of MPI's abstractions. Thus, *Gravel* cooperates with MPI rather than replacing MPI.

<code>gravel_init(NPROC, RANK, ERR)</code> INTEGER NPROC, RANK, ERR
<code>gravel_finalize(ERR)</code> INTEGER ERR
<code>gravel_register_buffer(BUF, SIZE, ERR)</code> <code>gravel_unregister_buffer(BUF, SIZE, ERR)</code> <type> BUF(*) INTEGER SIZE, ERR
<code>gravel_post_recv_buffer_rdma(DEST, BUF, SIZE, TAG, REQ, ERR)</code> <code>gravel_post_send_buffer_rdma(DEST, BUF, SIZE, TAG, REQ, ERR)</code> <type> BUF(*) INTEGER DEST, SIZE, TAG, REQ, ERR
<code>gravel_post_send_request_rdma(DEST, SIZE, TAG, ERR)</code> INTEGER DEST, SIZE, TAG, ERR
<code>gravel_wait_recv_buffer_rdma(SRC, TAG, ERR)</code> <code>gravel_wait_send_buffer_rdma(SRC, TAG, ERR)</code> INTEGER SRC, TAG, ERR
<code>gravel_wait_send_request_rdma(TAG, IPROC, ERR)</code> INTEGER TAG, IPROC, ERR
<code>gravel_post_os_put(DEST, BUF, SIZE, TAG, RMT_OFF, REQ, ERR)</code> <type> BUF(*) INTEGER DEST, SIZE, TAG, RMT_OFF, REQ, ERR
<code>gravel_post_os_get(SRC, BUF, SIZE, TAG, RMT_OFF, REQ, ERR)</code> <type> BUF(*) INTEGER SRC, SIZE, TAG, RMT_OFF, REQ, ERR
<code>gravel_send_fin(DEST, TAG, ERR)</code> INTEGER DEST, TAG, ERR
<code>gravel_test_and_pop(REQ, FLAG, TYPE, STATUS, ERR)</code> INTEGER REQ, FLAG, TYPE, STATUS(MPI_STATUS_SIZE), ERR
<code>gravel_wait(REQ, ERR)</code> INTEGER REQ, ERR
<code>gravel_gettime(TIME)</code> DOUBLE PRECISION TIME

**Figure 2. *Gravel* FORTRAN API**

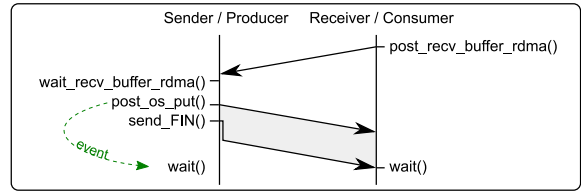
**API.** From an application developer's perspective, *Gravel*

provides a small, simple and high level API. Depending on how it is used, *Gravel* can appear almost as abstract as MPI. Figure 2 depicts the FORTRAN API of *Gravel*. The entire *Gravel* API for FORTRAN is implemented as subroutines that are invoked via the FORTRAN CALL statement. In the rest of this document we will be using the terms “procedure”, “function” and “method” interchangeably, since in this context they unambiguously have the same meaning. All the sizes passed as arguments to *Gravel* functions are in bytes.

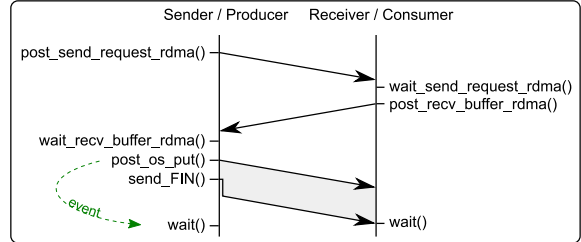
**Implementation.** From a performance standpoint, *Gravel* is a thin layer over an existing low level communication library. Currently, *Gravel* is implemented on top of uDAPL [2], the User Direct Access Programming Library, defined by the DAT Collaborative. We chose uDAPL because it works on our Infiniband cluster test bed, it is an open standard and it works on various modern cluster interconnects. Nothing in the design of *Gravel* enforces the use of uDAPL as the underlying layer. If someone wants to re-implement *Gravel* over a different low level library they would be free to choose any such library as long as it provides reliable one-sided operations, and of course some mechanism (such as events) for a process to know if a past operation that the process issued has finished.

To keep the API small and simple, we hid all pointers and hard to handle meta-data information in hash tables maintained by *Gravel*. Storing all the the low level information concerning a message into a hash table and associating that information with the user provided “request”, enables the application developer to send, receive, or wait for each message by handling just that request. Storing and retrieving the meta-data from the hash tables is a fast process and does not increase with the message size.

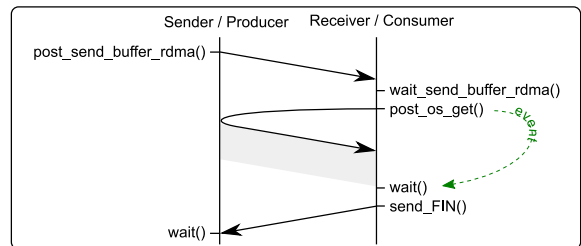
**A Brief Look at *Gravel* Rendezvous Protocols.** Several communication libraries, including MPI implementations, use implicit handshakes to synchronize communication. An in-depth study of the implications and performance of different data exchange protocols in the MPI implementation *MVAPICH*, can be found in [9]. *Gravel* in contrast, provides the application programmer with distinct functions that transfer data only, and functions that transfer meta-data only. This forces any exchange protocol to be explicitly implemented at the application layer. However, all the necessary data structures and machinery needed for implementing handshakes are hidden behind the library API, so that the application programmer does not need to worry about 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 that application. At the same time, the high-level, abstract API enables programmers to easily implement highly efficient exchange proto-



(a) Consumer Initiated RDMA write protocol



(b) Producer Initiated RDMA write protocol



(c) Producer Initiated RDMA read protocol

**Figure 3. Timing schematics of RDMA\_write and RDMA\_read based rendezvous protocols**

cols, even in languages such as FORTRAN that does not support C pointers (pointers to arbitrary memory locations, or pointers returned by C library functions).

The three simplest protocols that can be implemented with *Gravel* are presented in Figure 3. Due to space limitations, we only describe the consumer-initiated RDMA-write protocol here. We use the term “producer” for the node that will be the sender of the application data message, and “consumer” for the node that will be the receiver of the application data message. We do so to increase clarity in cases where the “consumer” sends *Gravel* meta-data messages to the “producer”, and therefore each side both sends and receives messages.

The consumer-initiated RDMA-write protocol case is presented in Figure 3(a) and is optimal for two-sided communication in applications that are ready to receive data from their peers, earlier than they are ready to send data to their peers.

In this case, the consumer of the application data initiates the handshake. It does so by invoking `gravel_post_rcv_buffer_rdma()`. This function is non-blocking and will asynchronously use RDMA to send a small meta-data message to a predefined location in

the consumer’s memory (initialized by `gravel_init()`) that we refer to as the *receive-info* ledger. At the application level, the only information that needs to be provided to the call is the communication peer (i.e. the producer), the start of the local memory where the data will be stored, the size of the expected data (not the size of the local buffer), an application defined tag, and a request handle. Clearly, the size of the local buffer must not be smaller than the expected data. The caller of `gravel_post_recv_buffer_rdma()` (i.e., the consumer) can now proceed with independent computation in order to hide the communication delay, but it must assume that the memory region specified by the parameters of this call can be altered at any time between this call and the return of `gravel_wait()` when the latter is given as argument that the request handle filled by `gravel_post_recv_buffer_rdma()`.

On the other side, the producer starts by calling `gravel_wait_recv_buffer_rdma()`. This blocking function will internally poll the *receive-info* ledger until the meta-data message from the consumer arrives. Upon arrival, the high level information of the meta-data message (consumer, buffer, size, etc) is copied into another predetermined structure (*curr\_remote\_buffer*), along with any necessary low level information required by the underlying library in order to perform the following application data transfer. Clearly, since this function is blocking, it needs to be called after `gravel_post_recv_buffer_rdma()` in cases where the communication is symmetric and all tasks are both consumers and producers. After `gravel_wait_recv_buffer_rdma()` has returned, the producer knows that the consumer is ready to receive the data (since the consumer has already posted the receive buffer meta-data) and also has all the information needed for the transfer in *curr\_remote\_buffer*. At that point, the producer initiates a non-blocking RDMA-write operation to transfer the application data to the consumer. It does so by calling `gravel_post_os_put()`. This function provides the RDMA engine of the network interface with the necessary information about the transfer and returns without waiting for the transfer to take place. Afterwards, the producer invokes the non-blocking call `gravel_send_fin()` which will asynchronously use RDMA to send a small meta-data message to a pre-defined location in the consumer’s memory (*RDMA-FIN* ledger)

After this step, the producer can proceed with independent computation, in order to hide the communication latency. Finally both sides will wait for the completion of the transfer. Although at the application layer both the producer and the consumer invoke the same blocking function (`gravel_wait()`), the producer waits on the underlying hardware to signal the completion of the RDMA-write op-

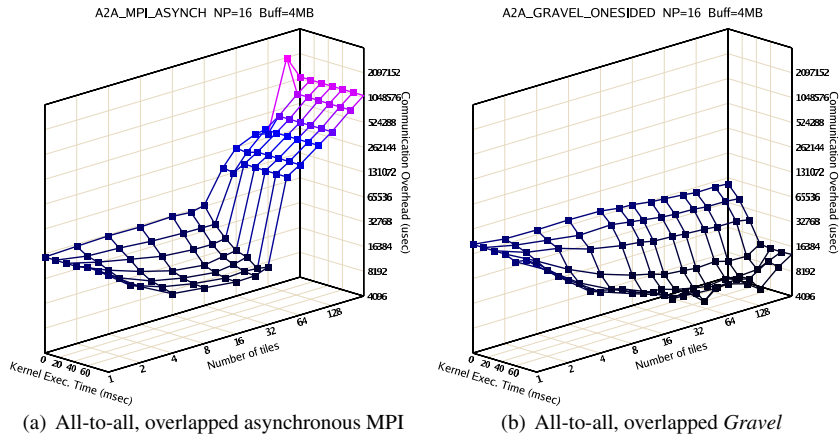
eration, while the consumer polls the *RDMA-FIN* ledger for the completion of the transfer.

This case is expected to perform well in applications that adhere to the following rationale. In SPMD applications, data to be sent by a producer to a consumer are generated by the producer before they can be sent. Therefore the send operation cannot be initiated earlier than the code that produces the data. On the consumer’s side on the other hand, data has to be received before it can be used (i.e., consumed). Therefore, the receive operation could be hoisted so that it is initiated earlier. In particular, as far as the consumer is concerned the receive request can be posted as early as the beginning of the program (unless the receive buffer is reused earlier in the application.) Therefore, the handshake overhead can be entirely overlapped with independent computation and the actual data transfer can be performed by a single RDMA-write operation without any delays or copying, regardless of the size of the transfer, leading to a very efficient data exchange.

### 3. Experimental Study

To study the performance that can be achieved with *Gravel*, we conducted experiments using a synthetic benchmark on an Infiniband cluster. Two versions of the benchmark were studied; one that uses the MVAPICH implementation of MPI, and another that uses the former library, but relies on *Gravel* for the time-critical data transfers. Both cases exchange data in a clique topology (all-to-all). In both cases, the code starts by initiating all the necessary receive operations using the asynchronous function `MPI_Irecv()`, or `gravel_post_recv_buffer()` respectively. Then, both cases enter a tiled loop. At each iteration of this loop one computation tile is executed and then the data produced by that computation is communicated. This is implemented via a call to `MPI_Isend()`, or `gravel_wait_recv_buffer()` followed by `gravel_post_os_send()`. Finally, all the asynchronous transfers are waited for after the loop.

By varying the number of tiles needed to process and communicate the data buffer, we control the message size and therefore study the effect of message size on the performance of *Gravel*. Figure 4 shows the performance witnessed by the benchmark. The subgraph on the left demonstrates the performance of the code that uses asynchronous MPI calls and the subgraph on the right demonstrates the performance of the code that uses *Gravel*. The vertical axis of each sub-graph represents the communication overhead witnessed by the benchmark. This is calculated by subtracting the time it took to execute the computation kernel from the total time it took to perform the computation and the (overlapped) communication. The long horizontal axis represents the number of tiles (which is reversely proportional



**Figure 4.** *Gravel* performance compared to *MPI* through a synthetic Benchmark

to the message size) and the short horizontal axis represents the execution time of the compute kernel.

Several observations can be made based on these results:

- As the computation time increases the communication overhead decreases for both cases. This is an intuitive result, since when there is more computation to overlap the transfers with, more of the transfers' delay will be hidden.
- The case using *Gravel* outperformed the case using *MPI* for most values of the independent variables.
- The design decision that we made for *Gravel*, to use a protocol that utilizes RDMA for all message sizes, has positive results when communication and computation are executed in a way that enables overlapping. Namely, while *MPI* switches to an eager protocol for small message sizes and fails to hide the communication latency, *Gravel* exhibits a smooth and uniform performance.

#### 4. Current Status

*Gravel* is currently a fully working library that can achieve significant performance, if used properly. Our current work is focused on evaluating the performance of *Gravel* more thoroughly through synthetic benchmarks and scientific applications, as well as testing and debugging the library so that we can release a first, stable version of *Gravel* soon.

Orthogonally to that, we are working on an Open64 based tool, aimed at automatically transforming *MPI* applications such that they utilize *Gravel* to achieve better communication-computation overlapping. *Gravel*, although usable as a stand-alone library, was developed to be a part of a bigger compiler system able to optimize *MPI*

codes. In our previous work [4, 5] we have shown that scientific applications can achieve lower communication overhead if appropriate transformations are applied to them. In our current and future work we plan to integrate the process of generating *Gravel* code into our optimization system and extend the optimization components of the system such that complex scientific codes can be automatically transformed to achieve more efficient communication.

#### References

- [1] Gm reference manual. <http://www.myri.com/scs/GM/doc/refman.pdf>.
- [2] uDAPL: User Direct Access Programming Library. [http://www.datcollaborative.org/uDAPL\\_doc\\_062102.pdf](http://www.datcollaborative.org/uDAPL_doc_062102.pdf).
- [3] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.
- [4] A. Danalis, K. Kim, L. Pollock, and M. Swany. Transformations to Parallel Codes for Communication-Computation Overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.
- [5] A. Danalis, L. Pollock, and M. Swany. Automatic *MPI* application transformation with ASPHALT. In *Performance Optimization for High-Level Languages and Libraries (POHLL '07) in conjunction with IPDPS 2007 (also in NSFNGS '07)*, Long Beach, CA, Mar 2007.
- [6] Mellanox Technologies Inc. Mellanox IB-Verbs API (VAPI), 2001.
- [7] Myricom Inc. Myrinet EXpress (MX): A High Performance, Low-level, Message-Passing Interface for Myrinet. <http://www.myri.com/scs/>, 2003.
- [8] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *RTSPP IPPS/SDP'99*, 1999.
- [9] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. RDMA read based rendezvous protocol for *MPI* over InfiniBand: design alternatives and benefits. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 32–39, 2006.