Implementing an Open64-based Tool for Improving the Performance of MPI Programs

Anthony Danalis Lori Pollock Martin Swany John Cavazos

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

Abstract

While MPI parallel programming has become the primary approach to achieving performance gains in cluster computing, the communication overhead inherent in a cluster environment continues to be a major obstacle. A promising approach to improve performance is the use of computation-communication overlapping, which is enabled by communication libraries that utilize Remote Direct Memory Access (RDMA), either directly in the form of one-sided communication, or via two-sided communication over a low overhead rendezvous protocol. To spare the scientific programmer from learning how to utilize these libraries to effectively maximize computation-communication overlap, we have developed a tool that automatically transforms an MPI parallel program to a semantically equivalent program with selected data exchange calls in MPI replaced to leverage an RDMA-targeted communication library. In this paper, we describe the implementation of this MPI program transformer using the Open64 compiler.

1. Introduction

Domain scientists typically parallelize their programs to exploit cost-effective parallel clusters using a high-level message passing interface such as MPI [2], and rely on the MPI implementation to achieve high performance. Often, communication-intensive parallel programs suffer in such an environment from the layers of communication software between the sender processes and the receiver processes. Performance can often be improved through direct transfers from memory coupled with asynchronous communication primitives. A promising approach to improve performance is the use of computation-communication overlapping, which is enabled by communication libraries that utilize Remote Direct Memory Access (RDMA), either directly in the form of one-sided communication, or via two-sided communication over a low-overhead rendezvous protocol.

Many researchers have proposed optimizations for parallel software in both the compilers and the interconnecting networks [11, 13, 14, 21, 17, 3, 9, 6]. However, most of these optimizations target specialized hardware or programming languages, require specialized knowledge from the application programmer (engineer or domain scientist), or are not enough to provide a comprehensive solution on their own. Thus, many parallel applications exist and are still being written utilizing only simplistic message passing mechanisms (usually small subsets of *MPI*) although they could achieve significantly better performance with network-specific optimizations.

Our goal is to allow the scientific programmer to take advantage of the RDMA-based communication libraries while sparing them from learning how to utilize these libraries to effectively maximize computation-communication overlap. To this end, we have developed a tool that automatically transforms an MPI parallel program to a semantically equivalent program with selected message passing calls in MPI replaced to utilize an RDMA-targeted communication library. In contrast to most existing work on enhancing the performance of MPI applications, which generally concentrates on enhancing the MPI library itself, our effort focuses on optimizing the parallel application that uses the communication primitives to exchange messages between tasks. Specifically, we perform program analysis and transformation on the MPI parallel program in order to use the communication library more effectively.

In this paper, we describe the implementation of this MPI program transformer using the Open64 compiler. Our transformer takes an MPI parallel program as input and compiles it to a binary that exploits the underlying cluster's smart interconnect technology [12, 4, 20]. These interconnect technologies allow communication to take place concurrently and independently from computation by using RDMA. Our transformation selectively replaces key data exchange MPI calls to utilize our Gravel library, which provides direct access to the one-sided transfer capabilities of the network interconnect. Use of this library has been demonstrated to reduce communication overhead by enabling it to be overlapped with computation more effectively [8]. We leverage the Open64 compiler infrastructure to build a transformer that evenutally will also perform advanced program analysis

to further optimize the computation-communication overlap through code motion.

The main contribution of this paper is a description of the implementation of the basic program transformer for MPI parallel programs into a form that uses a library that enables computation-communication overlap, where most appropriate, while leaving other MPI calls in place. The importance of this paper's contribution is that it demonstrates a mechanism via which the compiler can improve the performance of existing parallel codes, and enable codes to be written in a simple and maintainable style while being transformed to perform well on modern cluster hardware. Therefore, the impact of this work is twofold. First, it enables newly developed parallel applications to utilize the advanced features of modern cluster interconnects without increasing time-to-solution by requiring the developers to learn how to program using complicated libraries or new language extensions. Second, it enables legacy codes written in MPI that do not communication-computation overlapping to be transformed into more efficient codes without requiring a human to understand these codes at the level necessary to transform them manually.

2. Transformation Challenges and Opportunities

The implementation of the MPI program transformation tool described in this paper is primarily orthogonal to the library choice. That is, several communication libraries [8, 19, 5, 18, 1, 15, 10] including MPI itself, could be used in principle by a compiler transformation phase that aims to achieve better communication-computation overlapping. We chose to target the Gravel library [8] for selected data exchange MPI calls due to several advantages of Gravel's design. Gravel (1) enables direct RDMA operations, (2) provides separate functions for meta-data exchange and application data exchange, (3) ports beyond a single type of interconnect, (4) does not perform any internal operations (such as queueing and copying unexpected messages) that would lead to performance penalties, (5) provides finer control of the communication process by the application layer than most other options, and (6) is directly usable in Fortran applications. In the remainder of this paper, we focus our discussion specifically on the implementation of our MPI program transformation tool which emits "gravel-ized" code.

The major challenges faced by an implementer of an MPI program transformation tool for increasing computation-communication overlap lie in the differences between MPI and the targeted communication library. Gravel differs from MPI in a number of ways:

- The application code must explicitly perform memory registration when Gravel is used for communication.
- In Gravel, data transfer functions are separate from meta-data exchange functions.
- Different functions are called to perform consumer-initiated versus producer-initiated data exchanges in Gravel.

Figures 1 and 2 illustrate these differences through two example code snippets. In each figure, we contrast the asynchronous MPI implementation with the Gravel implementation of the same simple communication operation.

```
mpi_irecv(rbuf[], size, ...)  } Memory registration
mpi_isend(sbuf[], size, ...)  } Meta-data exchange
                              } Data exchange

mpi_wait()
mpi_wait()
```

(a) MPI-based code

```
gravel_register_buffer(rbuf[], size)  } <-- Memory registration
gravel_register_buffer(sbuf[], size)  }

gravel_post_recv_buffer_rdma(rbuf[], size, ...)  } Meta-data exchange
gravel_wait_recv_buffer_rdma(...)                }

gravel_post_os_put(sbuf[], size, ...)  <-- Data exchange

gravel_wait()
gravel_wait()
```

(b) Equivalent Gravelized code

Figure 1. Contrasting MPI and Gravel consumer-initiated data exchange

```
mpi_isend(sbuf[], size, ...)  } Memory registration
mpi_irecv(rbuf[], size, ...)  } Meta-data exchange
                              } Data exchange

mpi_wait()
mpi_wait()
```

(a) MPI-based code

```
gravel_register_buffer(rbuf[], size)  } <-- Memory registration
gravel_register_buffer(sbuf[], size)  }

gravel_post_send_buffer_rdma(sbuf[], size, ...)  } Meta-data exchange
gravel_wait_send_buffer_rdma(...)                }

gravel_post_os_get(rbuf[], size, ...)  <-- Data exchange

gravel_wait()
gravel_wait()
```

(b) Equivalent Gravelized code

Figure 2. Contrasting MPI and Gravel producer-initiated data exchange

The most important challenges posed by these library differences are:

- **Memory registration** is a process dictated by the hardware and operating system mechanisms that enable the remote direct memory access (RDMA) transfer mode. In particular, for every RDMA data transfer, the memory where that data resides needs to be registered. Memory registering is a relatively time consuming process and it is limited by the physical memory resources of the system. Gravel requires that memory registration be performed explicitly by the application (unlike MPI where registration is implicitly handled by the data transfer routines).

Thus, the MPI program transformation tool must decide which memory regions to register, how much memory to register, and where in the code to insert the memory registration function calls.

- **Communication protocol mapping.** Gravel separates the protocol header exchange from the data exchange. The header exchange must complete before data transfer begins, thus Gravel provides a blocking operation to assure that it has [1]. For this reason, a symmetric SPMD application (where all tasks perform the same operations) will deadlock if all tasks start by executing the blocking operation followed by a non-blocking operation. Therefore, care needs to be taken to ensure that the right exchange protocol (consumer-initiated vs. producer-initiated) will be used to avoid deadlocks.

In addition to the challenges, the differences between MPI and Gravel enable several MPI program optimization opportunities:

1. Since memory registration is a time consuming process, the program transformation tool can hoist the memory registration code outside communication loops or across caller-callee edges, to limit its runtime overhead.
2. Often, meta-data messages do not have data dependencies with the actual data buffers or the data dependencies are limited dependencies. This enables the MPI program transformation tool to perform code motion on the meta-data exchange function calls and thus overlap the control communication with useful computation.
3. Different parallel applications exhibit varying computation and communication patterns even within different parts of a single application. Therefore, the MPI program transformation tool can choose for each data exchange a communication mode that best suits the context where the exchange takes place.

This paper focuses on the implementation of the basic MPI program transformation tool which addresses the posed challenges, as well as optimization (1). Optimizations (2) and (3), are left for future work.

3. MPI Program Transformation Tool Implementation

To address the challenges and exploit the optimization opportunities described in Section 2 without requiring the MPI programmer to perform these tedious and error prone tasks, we implemented a tool based on the Open64 compiler that automatically replaces selected MPI communication calls with equivalent functionality from the Gravel library. Our tool takes as input an MPI parallel program and a programmer's indication of the targeted communication to be replaced. The output is the same MPI parallel program with the indicated MPI communication primitives replaced by the corresponding Gravel primitives as well as additional, necessary Gravel code. We first describe the overall integration of our tool with existing Open64 technology, and then the details of how each challenge of automatic MPI program transformation for improving computation-communication overlapping is addressed in this environment.

[1] MPI, in contrast, performs this header exchange implicitly, but still must pay the same penalty.

3.1 Leveraging the Open64 Compiler

Open64 is a large and complex compiler system. To utilize it, a compiler developer can either alter the code of an existing phase of the system or create a distinct phase that exists as a stand-alone executable, operating on intermediate files generated by the system. The latter scenario is a viable option because of the way Open64 is designed to operate.

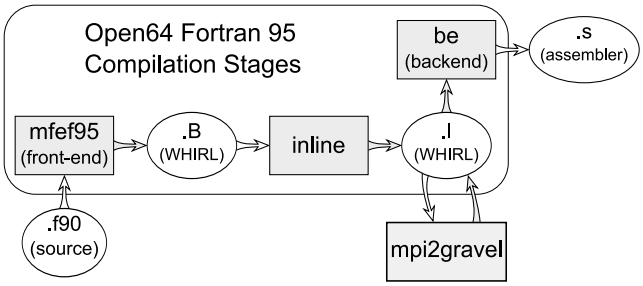

Figure 3. Open64 compilation process and our MPI program transformation tool

The normal compilation process of Open64 involves several standalone executables. Figure 3 shows how compiling a Fortran code begins with executing the front-end executable **mfef95**, which generates a "`.B`" file containing the original code represented as a WHIRL tree, along with the necessary symbol table information. Next, the stand-alone inliner, **inline**, performs potential inlining of function bodies at function call sites. The output of **inline** is again a WHIRL tree representation of the program in addition to symbol table information that is stored in a "`.I`" file. Finally, the executable **be** reads this file, performs program analysis, optimization and code generation, and produces a file containing assembly code. As we are focusing on existing Fortran applications, we do not perform pointer analysis. The generated code can be assembled and linked using external tools, such as *gcc*, *as*, and *ld*.

Our MPI program transformation phase exists as a standalone executable that operates on the "`.I`" file between the execution of the inliner and the Open64 backend, **be**. We considered integrating our tool into **be**; however, implementing a pass that operated before **be** has some benefits over a **be**-based tool. Some of the benefits of this design decision include rapid prototyping and better portability. In contrast to a **be**-based approach, implementing our tool as a separate code pass did not require thorough knowledge of

Open64 internals. This makes prototyping fast, as the MPI program transformer developer only has to learn the details of the WHIRL tree representation and corresponding symbol tables, rather than the details of different Open64 analysis and optimization phases and how they interact. In addition, a standalone MPI optimizer can work on any system that has a working binary version of Open64 that uses the same intermediate files and the same WHIRL. In contrast, an optimization phase integrated into the backend would be more difficult to use with different or future versions of the Open64 compiler.

Developing the MPI program transformer as a standalone phase has some negative consequences related to reimplementing existing functionality. Since our MPI program transformer is not inside the backend, none of the analysis and existing code transformations that exist in the backend can be exploited in the development of our transformer. To achieve the best of both worlds, we are planning to combine the two approaches. We will complement our existing standalone tool with further analysis and optimization phases of our tool, which will execute as part of the backend in order to take advantage of the existing analysis and transformation in the Open64 backend.

The next subsection describes how we currently enable programmer annotations to indicate the MPI communication call sites to be replaced by Gravel calls. We then describe our approaches and implementations for inserting memory registration operations and for mapping MPI send/recv operations to Gravel operations.

3.2 Identifying Selected MPI Communication Operations for Replacement

Our MPI program transformation tool replaces selected pairs of MPI synchronous and asynchronous send and receive operations and their corresponding wait operations such as: `mpi_send()`, `mpi_isend()`, `mpi_recv()`, `mpi_irecv()`, `mpi_wait()` and `mpi_waitall()`. Identifying matching send and receive operations in a parallel application with no annotations is a difficult data flow analysis problem and in general is statically undecidable [7, 16]. Thus, we currently assume that either the MPI programmer or a user-guided tool has identified the potential matching MPI operations of each desired communication replacement. Our MPI program transformer expects that this information is encoded in the form of PRAGMA directives annotating the targeted matching MPI send, receive and wait operations.

To create a quick prototype, our current implementation achieves the annotation by overloading the "`UNROLL`" directive. In particular, when the MPI program is examined by our transformation tool, only the call sites preceded by an `UNROLL` directive are considered to be candidates for gravelizing. We use the following syntax convention. An example use of the annotation is shown in Figure 4.

- A send or receive operation candidate for optimization should be annotated with the directive: "`!DIR$ UNROLL` $2*n$" such that $1 < n < 512$.
- The matching send or receive operation should be annotated with the directive: "`!DIR$ UNROLL` $2*n+1$" using the same value for n as above.
- The corresponding wait operations should be annotated with the directive: "`!DIR$ UNROLL` m" where m is the value ($m = 2*n$ or $m = 2*n+1$) used to identify the send or receive operation the given wait corresponds to.

```
!DIR$ UNROLL 12
call mpi_irecv(..., rreq, ...)
...
!DIR$ UNROLL 13
call mpi_isend(..., sreq, ...)
...
!DIR$ UNROLL 13
call mpi_wait(sreq)
!DIR$ UNROLL 12
call mpi_wait(rreq)
```

Figure 4. Example annotation to indicate targeted communication replacement

Clearly, the value of n must be unique for every matching pair that is targeted for communication replacement. In the case of `mpi_waitall()` where multiple send and receive operations are matched by a single wait, the call site of `mpi_waitall()` should be preceded by all the applicable directives, one after the other, in any order. The number used in the annotation of an MPI function call site is saved by our program transformation tool as the id of the given call site. The MPI program transformer keeps track of the directives used to annotate the code and removes them before saving the output file so that the subsequent phases of Open64 do not try to interpret them as loop unrolling directives. We will soon replace this overloaded UNROLL directive by a new directive (e.g., MPI2GRAVEL).

3.3 Analyzing the MPI Program to Insert Memory Registration Function Calls

Appropriate memory registration is required before a data transfer that uses RDMA. In particular, the memory region where the data will be read from (on the producer side) as well as the memory region where the data will be written into (on the consumer side) must be registered before the transfer takes place. As mentioned in section 2, unlike MPI where memory registration takes place implicitly inside the library data communication calls, Gravel requires that memory registration is performed explicitly at the application layer. Thus, the MPI program transformation tool needs to decide where to insert the memory registration function calls in the MPI program, and the location and amount of memory that needs to be registered. However, MPI program

analysis can utilize information about the program structure to change this requirement into a feature, by carefully selecting where and how to register the required memory regions to avoid unnecessary operations and reduce the performance penalties inherent in memory registration operations. This section describes the analysis we have implemented to achieve this goal.

A conservative choice for the location of the memory registration call for a given targeted communication replacement is the point immediately before the corresponding call that initiates a `send` or `receive` operation as shown in Figure 5. Although such a choice would produce correct code, it would often lead to redundant memory registrations and therefore suboptimal code.

```
call register_memory( rbuf[ expr1 ], length )
call recv( rbuf[ expr1 ], length, ... )

call register_memory( sbuf[ expr2 ], length )
call send( sbuf[ expr2 ], length, ... )
```

Figure 5. Naive placement of memory registration

To improve performance in the case of applications that invoke communication calls inside a loop, memory registration calls could be hoisted outside the loop as shown in Figure 6. Such a transformation would generate more efficient code, but the required program analysis to determine whether this will maintain semantically equivalent code is not always possible. For example, consider the case shown in Figure 7 where the starting location, and/or size of the message buffer are statically undecidable since the former depends on the potentially unknown values stored in array `index` and the latter depends on the return value of a potentially unknown function.

```
call register_memory( rbuf[ expr1 ], length )
call register_memory( sbuf[ expr2 ], length )
do i=1,nproc
    call recv( rbuf[ expr1 ], length, ... )
    call send( sbuf[ expr2 ], length, ... )
end do
```

Figure 6. Hoisted memory registration calls

In such a case, the compiler would not be able to statically generate a single memory registration call able to register the entire memory region enclosing all the message buffers, since that memory region cannot be statically determined. A conservative solution that would produce correct code is to generate a call that registers the entire memory region occupied by the array where the messages reside (or will be written to) as shown in Figure 8. To achieve this conservative solution, the compiler would still need to infer the size of the array. While this is always possible in Fortran for local arrays [2] it might not be so for arrays that are passed as arguments to a procedure, as shown in Figure 9.

```
do i=1,nproc
    length = f(i)
    call recv( rbuf[ index[i] ], length, ... )
    call send( sbuf[ index[i] ], length, ... )
end do
```

Figure 7. Undecidable memory region

```
call register_memory( rbuf[ rbufTotalSize ] )
call register_memory( sbuf[ sbufTotalSize ] )
do i=1,nproc
    length = f(i)
    call recv( rbuf[ index[i] ], length, ... )
    call send( sbuf[ index[i] ], length, ... )
end do
```

Figure 8. Registering the entire array

In such a case, the compiler needs to perform interprocedural analysis to trace the array definition. In particular, the analysis must traverse the call graph backwards, across callee-to-caller edges, starting from every call site of the function that performs the `send`/`receive` operations and proceeding to its caller until the "memory source" is found. Memory source, in this context, is the function that has a local declaration of the array that will be used by the communication operations. For example, in the code shown in Figure 9 function `exchange()` invokes communication primitives that use array `A` ($A_{exchange}$), but that array is local in function `main()`. Therefore, the memory source of $A_{exchange}$ is `main()`.

```
program main
  integer A(32), B(32)
  call foo( A, B )
end

subroutine foo( A, B )
  integer A(*), B(*), C(32)
  C(1:32) = B(1:32)
  call exchange( A, C )
end subroutine

subroutine exchange( A, B )
  integer A(*), B(*)

  do i= ...
      call recv( A(...), length )
      call send( B(...), length )
  end do
end subroutine
```

Figure 9. Array of unknown size as argument

To identify which arrays of each caller are passed to each callee, the call graph traversal needs to keep track of the formals and local arrays of the caller that are passed as

[2] Even if the array has a dynamic size evaluated at run-time based on the value of an expression, the call to the register function can use the same expression to register the correct amount of memory at run-time.

actuals to each call site. For example, in the case of Figure 9, the compiler must record that `main()` passes arrays `A` and `B` to `foo()` and `foo()` passes arrays `A` and `C` to `exchange`. Note that full control and data flow analysis is not necessary for finding the memory source. This is true because the data that will be in the message buffers is not what this analysis is trying to trace, but rather the memory region that will be involved in the data exchange. As an example, in Figure 9, the source of array `B` in `exchange()` is array `C` in `foo()` and not `B` in `main()`.

To perform this memory source tracing analysis, we implemented a phase that traverses the WHIRL tree of every program unit (PU) and saves in an output file the following information:

- the formal parameters of each function
- the call sites of each function, and for each call site:
  - each actual parameter
  - the number of the actual in the actual parameter list
  - whether the actual parameter is local to the caller [3], or passed as a formal parameter
  - the PRAGMA id of the call site

The information collected by this phase for the example program of Figure 9 is shown in Figure 10.

```
subroutine MAIN(  ){
  call id=0 foo:: A:0:T, B:1:T
}
subroutine foo( A, B ){
  call id=0 exchange:: A:0:F, C:1:T
}
subroutine exchange( A, B ){
  call id=0 recv:: A:0:F, LENGTH:1:T
  call id=0 send:: B:0:F, LENGTH:1:T
}
```

Figure 10. Collected information from memory source tracing analysis

```
traceActualsSrc(string name, id, actNum, callSiteList){
  list memSources;

  foreach( callSite=findCallSite(name,id,callSiteList) ){
    clr = callSite.caller;
    actual = callSite.actuals[actNum];

    if( isLocal( actual ) ){
      memSources.add( clr, actual );
    }else{
      callerFormal = findCorrespondingFormal( actual );
      s=traceActualsSrc(clr,0,callerFormal,callSiteList);
      memSources.add( s );
    }
  }
  return memSources;
}
```

Figure 11. Memory source tracing algorithm

[3] Global variables in *COMMON* are considered local to the caller.

When this memory source tracing information collection is collected for every file of the MPI parallel program, the collected information is used to construct a data structure we call the Memory Tracing Call Graph (MTCG). Figure 12 depicts the MTCG for the example code of Figure 9. The memory source for every array passed to each communication function can be located with a simple traversal of the MTCG. Figure 11 outlines the algorithm that takes as input a call site name and id, along with the actual parameter for which we are trying to identify its memory source. It produces as output a list of (memory source) callers and the names of the corresponding arrays.

When the memory source for each message buffer has been identified, the MPI program transformer inserts memory registration calls at the beginning of each corresponding procedure. This leads to code that is safe and efficient, since the memory registration cost is paid once per distinct memory region, regardless of the number of data exchanges performed from or to this region.

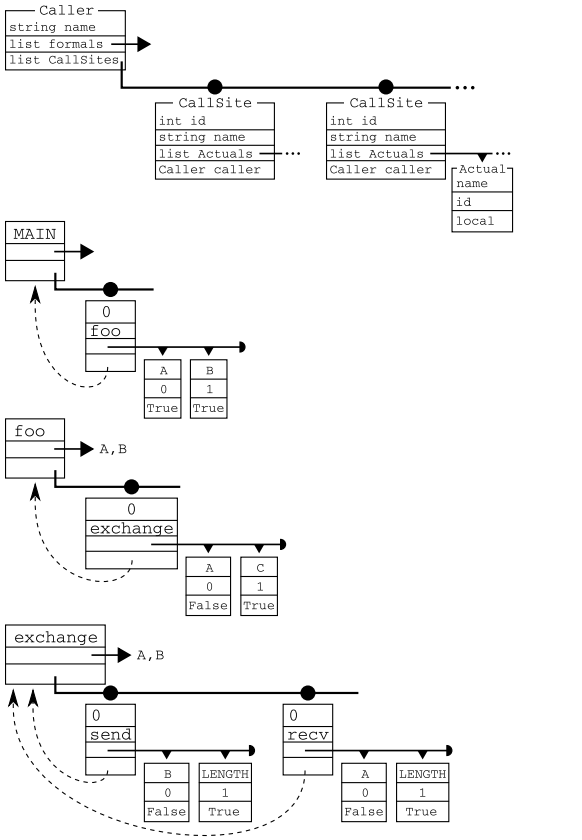

Figure 12. Memory Tracing Call Graph example

3.4 Mapping MPI `send/recv` Pairs to Gravel Protocols

Gravel uses two distinct pairs of send/receive operations to implement consumer-initiated versus producer-initiated data

exchanges. For consumer-initiated data exchanges, the consumer, i.e., the task that called `mpi_irecv()` in the original MPI parallel program, should now call the non-blocking call `gravel_post_recv_buffer_rdma()`. This function asynchronously sends a control message to the producer declaring that the consumer is ready to receive the data and specifying the memory location where the data should be put. In contrast, the producer starts by calling the **blocking** function `gravel_wait_recv_ buffer_rdma()` which returns only after the control message from the consumer has arrived. Afterwards, the producer proceeds with calling `gravel_post_os_put()` which is a non-blocking call that issues a one-sided RDMA write operation to asynchronously put the data into the consumer's memory as specified by the control message.

The rationale behind this design is that the function call that posts the receive request usually can be hoisted earlier in the execution path and therefore the blocking operation will not really have to wait. Despite the potential performance gains or losses of this design, there is a correctness issue that the MPI program transformer needs to take into account. Namely, if the application is a symmetric SPMD code where every task executes the same operations (or at least follows the same branches), then calling the blocking call before the non-blocking call would cause a deadlock.

To ensure deadlock avoidance, our MPI program transformer has to statically assess the relative order of execution of the original MPI send/receive operations and choose the correct corresponding set of Gravel functions (i.e., producer-initiated or consumer-initiated operations). Since the MPI-to-Gravel transformation happens in a phase implemented as a stand-alone tool that executes before the Open64 backend, we implemented some steps of control and data flow analysis inside our tool. Namely, we try to assess the execution order of the original MPI send/receive calls using the algorithm shown in Figure 13.

As can be seen in the algorithm, there are some cases (lines 23,26 and 49) where the order of execution is undecidable. If this is the case, our MPI transformer tool does not proceed with the transformation of the given code segment. In many cases, the execution order is either the same as the textual order, or reversed. The latter occurs in cases where both call sites are inside the body of a loop and the first call site (textually), is inside a one time **if** guard that checks if the loop induction variable is greater than the lower bound of the loop. This case is captured by the **if** statement on line 37 of the execution order discovery algorithm (shown in Figure 13).

If this analysis is successfully performed, the tool can proceed with the transformation and use the correct Gravel calls to ensure that the non-blocking calls will send the control messages before the blocking calls block waiting for them.

```
1  if( inSameBasicBlock( send, recv ) ){
2
3    return firstInTextualOrder( send, recv );
4
5  }else{
6
7    topBlock = findClosestEnclosingBlock( send, recv );
8
9    encSend = findTopNonSharedBlock( send, topBlock );
10   encRecv = findTopNonSharedBlock( recv, topBlock );
11
12   txtFirst = firstInTextualOrder( encSend, encRecv );
13   txtLast = lastInTextualOrder( encSend, encRecv );
14
15   if( isNotDominatedByBranch( txtFirst, topBlock ) ){
16
17     return txtFirst;
18
19   }else{
20
21     if( isDominatedByLoop( encSend, encRecv ) ){
22
23       if( isDominatedByUndecidableBranch(send) )
24         return ORDER_UNKNOWN;
25
26       if( isDominatedByUndecidableBranch(recv) )
27         return ORDER_UNKNOWN;
28
29       if( isDominatedByOneTimeGuard( txtFirst ) &&
30         isGTLoopLowerBoundComparison( guard, loop.lb )){
31
32         if( isDominatedByOneTimeGuard( txtLast ) &&
33           isGTLoopLowerBoundComparison( guard, loop.lb )){
34
35           return txtFirst;
36
37         }else{
38           return txtLast
39         }
40
41       }else{
42         return txtFirst;
43       }
44
45     }else{
46     }
47
48   }
49
50   return ORDER_UNKNOWN;
51 }
```

Figure 13. Send-Recv execution order discovery algorithm

In summary, the mapping of MPI send/recv calls to Gravel protocols includes memory source tracing and send/recv execution order determination. There are cases where it is not possible to determine this information statically. In those cases, we conservatively do not perform gravelization of the targeted MPI communication sites. However, in our preliminary work with benchmarks from well known suites, including the NAS benchmarks, we have typically found that this information can be determined statically.

4. Current Status

As mentioned in Section 3.1 our current implementation of the transformation tool works as a stand-alone executable that modifies the intermediate WHIRL files before the backend (**be**) of Open64 is invoked. While it is currently under

active development, our transformer is already able to handle complex application codes including several of the NAS benchmarks.

Our current plans are focused on extending our existing infrastructure as well as developing code that can execute inside the backend of Open64. Our goal regarding the standalone executable is to make it able to transform complex applications so that they use Gravel instead of MPI for key selected data transfers. Regarding the code integrated into Open64, our goal is to utilize the existing program analysis and transformation capabilities of the Open64 system to improve the performance of the transformed applications even further.

References

[1] Gm reference manual. `http://www.myri.com/scs/GM/doc/refman.pdf`.

[2] The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/.

[3] Francoise Baude, Denis Caromel, Nathalie Furmento, and David Sagnol. Overlapping communication with computation in distributed object systems. In *HPCN Europe*, pages 744–754, 1999.

[4] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[5] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

[6] Thomas Brandes and Frederic Despez. Implementing pipelined computation and communication in an HPF compiler. In *Euro-Par, Vol. I*, pages 459–462, 1996.

[7] Dale Shires and Lori Pollock and Sara Sprenkle. Program Flow Graph Construction for Static Analysis of MPI Programs. In *Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, pages 1847–1853, June 1999.

[8] Anthony Danalis, Lori Pollock, and Martin Swany. Introducing Gravel: An MPI Companion Library. In *NSF Next Generation Systems Workshop (NFS/NGS 2008) in conjunction with IPDPS 2008*, Miami, FL, Apr 2008.

[9] Frederic Desprez, Pierre Ramet, and Jean Roman. Optimal grain size computation for pipelined algorithms. In *Euro-Par, Vol. I*, pages 165–172, 1996.

[10] Ouissem Ben Fredj and Éric Renault. Performance analysis of rwapi on top of the myrinet-2000 interconnect. In *Communications and Computer Networks*, pages 40–45, Lima, Peru, 2006.

[11] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Supercomputing*, pages 86–100, 1991.

[12] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.

[13] Ken Kennedy. Telescoping Languages: A Compiler Strategy for Implementation of High-level Domain-specific Programming Systems. In *roceedings of International Parallel and Distributed Processing Symposium 2000 (IPDPS 2000)*, pages 297–306, May 2000.

[14] C. Kessler and W. Paul. Automatic parallelization by pattern matching. In *Proceeding of Second Int. Conference of the Austrian Center for Parallel Computation*, pages 166–181, 1993.

[15] Mellanox Technologies Inc. Mellanox IB-Verbs API (VAPI), 2001.

[16] Michelle Mills Strout and Barbara Kreaseck and Paul D. Hovland. Data-Flow Analysis for MPI Programs. In *International Conference on Parallel Processing (ICPP 2006)*, pages 175–184, Aug 2006.

[17] Matthias M. Muller. Compiler-generated vector-based prefetching on architectures with distributed memory. *High Performance Computing in Science and Engineering '01, Transactions of the High Performance Computing Center Stuttgart (HLRS)*, pages 527–539, 2001.

[18] Myricom Inc. Myrinet EXpress (MX): A High Performance, Low-level, Message-Passing Interface for Myrinet. http://www.myri.com/scs/, 2003.

[19] 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.

[20] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, 2002.

[21] A. K. Somani and A. M. Sansano. Minimizing overhead in parallel algorithms through overlapping communication/computation. Tech. Rep. 97-8, ICASE, Feb. 1997.