# Enhancing Performance Portability of MPI Applications Through Annotation-Based Transformations

Md. Ziaul Haque
*Dept. Computer Science,*
*U. Texas at San Antonio,*
*San Antonio, Texas, USA*
*mzh.olive@gmail.com*

Qing Yi
*Dept. of Computer Science,*
*U. Colorado at Colorado Springs,*
*Colorado Springs, CO, USA*
*qyi@uccs.edu*

James Dinan Pavan Balaji
*Math. and Comp. Sci. Div.,*
*Argonne National Lab.,*
*Chicago, IL, USA*
{*dinan,balaji*}*@mcs.anl.gov*

***Abstract*—MPI is the de facto standard for portable parallel programming on high-end systems. However, while the MPI standard provides functional portability, it does not provide sufficient performance portability across platforms. We present a framework that enables users to provide hints about communication patterns used within MPI applications. These annotations are then used by an automated program transformation system to leverage different MPI operations that better match each system's capabilities. Our framework currently supports three automated transformations: coalescing of operations in MPI one-sided communications; transformation of blocking communications to nonblocking, which enables communication-computation overlap; and selection of the appropriate communication operators based on the cache-coherence support of the underlying platform. We use our annotation-based approach to optimize several benchmark kernels, and we demonstrate that the framework is effective at automatically improving performance portability for MPI applications.**

***Keywords*-high performance computing, parallel programming, automatic programming**

## I. Introduction

MPI [9] is a de facto standard for parallel programming in scientific domains. *Performance portability* for MPI programs, however, is challenging because of the hard-to-predict relative cost of MPI operations across systems, due to unknown variations in hardware parameters, system capabilities, and features provided by MPI implementations.

To elaborate, the rich communication semantics of MPI provide users with multiple algorithmic choices so that the same application functionality can be implemented using a variety of different operations in MPI. For instance, Fourier transform can either be implemented in a bulk synchronous model where different processes compute and collectively exchange data at regular synchronization points (using MPI collective communication operations) or through a more asynchronous model using MPI one-sided communication where data is moved when it is ready. While the two models are functionally equivalent, the performance they achieve varies on different systems. For example, InfiniBand based clusters can utilize hardware-supported asynchronous progress capabilities, making them ideal platforms for using MPI one-sided communications. On the other hand, platforms such as IBM Blue Gene provide hardware support for collective acceleration, making them ideal for using group communication operations. Which MPI communication routines perform better is highly platform specific, making it hard for the application developers to determine the right algorithmic choice a priori in a portable manner. Unfortunately, state-of-the-art MPI implementations, in spite of their many aggressive dynamic optimizations, only see each MPI operation individually and focus on optimizing it. As a result, they cannot always handle such performance differences internally due to the lack of a holistic view of the application algorithms.

This paper presents an annotation-based program transformation framework to help MPI applications enhance their performance portability across different platforms. The central idea is to build automated program transformations that utilize user-supplied hints about their underlying algorithmic models as well as system-specific information to automatically transform user applications to utilize the best MPI operations for each platform. By modulating the behaviors of both the application and MPI implementations, our approach enables applications to automatically leverage system-specific capabilities that enhance the efficiency of some MPI operations over others. Consequently, it provides a portable means for accessing system-specific features beyond the MPI standard.

The workflow of our framework is shown in Figure 1. Specifically, we allow developers to annotate their applications with concise information about the MPI communication mechanisms used in varying blocks of statements in their applications. Based on these user annotations combined with additional information of the underlying runtime platform, the optimization analysis component of our framework determines possible program transformations to enhance the efficiency of MPI operations in the user application. The transformation decisions are then fed into a *program transformation* component, which automatically specializes the annotated communication blocks for the underlying platform before sending the user application to the vendor compiler to generate executables.

Our framework currently supports a number of user annotations that provide simple hints on the opportunities of applying three program transformations: coalescing of MPI one-sided communications, overlapping communications with computations, and automatic selection of the appropriate communication operators based on the cache-coherence support of the underlying platform. We present experiment results using the Graph500 benchmark [3], a

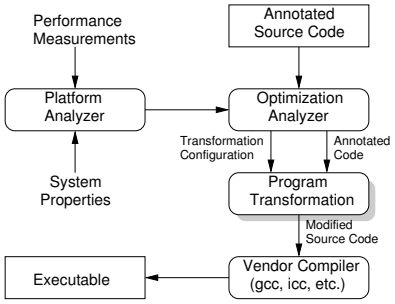



Figure 1: Annotation-based optimization workflow.

```
(1) start          ::= annot_block start | UNKNOWN_UNTIL_EOL start | ϵ
(2) annot_block ::= "#" "pragma" "mpi" pragma BLOCK
    pragma         ::= dc_annot|cco_annot|rma_annot|ldst_annot|indep_annot
(3) dc_annot       ::= "osc_coalesce" win_buf_list overlap
    overlap        ::= "no_overlap" | ϵ
    win_buf_list::= win_buf win_buf_list | ϵ
    win_buf        ::= "(" ID "," ID "," TYPE "," osc_spec ")"
    osc_spec       ::= ID | ID "," osc_spec
(4) cco_annot      ::= "cco" comm_grp_list
    comm_grp_list::= comm_grp comm_grp_list | ϵ
    comm_grp       ::= mpi_comm "(" ExpList ")"
    mpi_comm       ::= "MPI_Send" | "MPI_Recv" | "MPI_Wait" | "MPI_Win"
    ExpList        ::= EXP "," ExpList | EXP
(5) rma_annot      ::= "rma" win_buf_list
(6) ldst_annot     ::= "local_ldst" win_buf_list overlap
(7) indep_annot ::= "indep" comm_grp_list
```

* UNKNOWN_UNTIL_EOL: parse the current line as a list of unstructured strings;
* ID : an identifier; EXP: an expression of the underlying language;
* TYPE: a type expression in the underlying language;
* BLOCK: a single block of statements in the underlying language

Figure 2: BNF definitions of the annotation language.

stencil kernel, a synthetic benchmark, and the NAS FT benchmark [4]. Our results demonstrate that our framework can consistently achieve high performance across different platforms better than what users can typically achieve manually while retaining application portability.

Our contributions include: (1) we present and demonstrate the effectiveness of a light-weight annotation-based framework to enhance the performance portability of MPI applications; and (2) we demonstrate the sensitivity of several important application-level MPI optimizations to varying runtime configurations and draw insights into how to make careful optimization decisions.

The rest of the paper is organized as follows. Section II presents our annotation language. Section III presents our overall framework to support the associated program transformations. Section IV presents performance results demonstrating the benefits of our framework. Section V presents related work, and Section VI concludes.

## II. The Annotation Language

The syntax of our annotation language is summarized in Figure 2 and includes the following components.

- *Overall structure of the input program.* As specified at Line 1 of Figure 2, our framework parses the whole input program as a sequence of annotated statement blocks (*annot_block*) or single lines of unknown strings (*UNKNOWN_UNTIL_EOL*). Therefore, only the annotated code segments of a large MPI application will be parsed while the rest of the application will simply be saved as unstructured strings.
- *The structure of each annotated block.* Line 2 of Figure 2 specifies that each annotated block must start with *#pragma mpi* followed by a specific annotation and a block of statements. Our framework currently supports five annotations, defined at Lines 3–7.
- *The data coalescing annotation* (*dc_annot* at Line 3), which starts with the keyword "osc_coalesce" followed by a list of the windows and buffers of one-sided communications ($win\_buf\_list$) whose data could be coalesced. An optional specification, $no\_overlap$, can be used to indicate that the data of different MPI operations are never sent to the same addresses (by default, overlapping is assumed when MPI_Accumulate is involved).
- *The cco annotation (cco_annot* at Line 4), which starts with the keyword "cco" followed by a list of the MPI communications, each specified as an MPI operator (e.g., MPI_Send) followed by an optional list of parameters or as an MPI window (MPI_Win) followed by a window name, that could be moved to better overlap with independent computations.
- *The remote-memory access annotation* (*rma_annot* at Line 5), which includes a keyword "rma" followed by a list of the windows and buffers of one-sided communications ($win\_buf\_list$) that may be converted to local loads and stores when appropriate.
- *The local load/store annotation* (*ldst_annot* at Line 6), which includes a keyword "local_ldst" followed by a list of the windows and buffers ($win\_buf\_list$) being operated by the annotated block through local load/store operations. An optional specification, $no\_overlap$, can be used to specify that there is no aliasing or overlapping among these buffers.
- *The independent annotation* (*indep_annot* at Line 7), which includes the keyword "indep" followed by a list of the MPI communications that are independent of the statement block being annotated; that is, the annotated statements do not interfere with or use any data received from the listed MPI communications.

Our *Optimization Analyzer* in Figure 1 uses the above annotations to ensure both the safety and the profitability of the relevant program transformations, through the analysis algorithm shown in Figure 3. The algorithm takes as input the underlying platform configuration ($config$) and the input MPI code to optimize. It traverses the input program, searches for occurrences of user-annotated code segments, and then invokes the corresponding optimizing transformations to modify the input code accordingly based on current platform configurations. Each of the annotations at Lines 3–6 of Figure 2 is used to enable an automated transformation currently supported within our framework, and the independent annotation at Line 7 is combined with the other annotations to ensure the correctness of the program transformations.

In essence, our framework requires developers to manually insert annotations to automate the necessary transformations of porting their applications to varying execution platforms. The annotations themselves are minimally

```
optimization_analysis(input, config)
  input: input MPI program to optimize;
  config: architecture configurations of the underlying system;
foreach annotated MPI block (annot, body) in input:
(1) if (is_data_coalesce_annot(annot)):
      foreach win ∈ win_buf_list(annot):
       mpi_osc_data_coalesce(win, has_overlap(annot), body);
(2) if (is_cco_annot(annot)):
      foreach comm ∈ comm_groups_of(annot):
       mpi_comp_comm_overlap(comm, innermost_body_of(body));
(3) if (is_rma_annot(annot)):
      foreach win ∈ win_buf_list_of(annot):
       if (cache_coh(config)): mpi_rma_2_ldst(win,body);
(4) if (is_ldst_annot(annot)):
      if (cache_coh(config)): mpi_ldst_coh(win,has_overlap(annot),body);
      else mpi_ldst_incoh(win,has_overlap(annot),body);
```

Figure 3: Optimization analysis algorithm.

intrusive to the original source code. While developers need to deliberately consider alternative implementations to insert annotations correctly, the annotations allow their applications to automatically attain performance portability without having to be manually specialized for each MPI platform. In particular, our framework allows developers to specify a single, possibly simplest, algorithm implementation and provides developers with the ability to automatically synthesize alternative implementations from the original one based on different platform configurations, thereby significantly improving the productivity of porting MPI applications to varying platforms.

## III. Annotation-Based Framework

As shown in Figure 1, the workflow of our overall framework includes the following three key components.

- *Platform analysis*, which collects information about the underlying platform, e.g., the number of available nodes and processing cores, the cache coherence protocol, and the MPI library installed, by querying the operating system or by empirically evaluating varying MPI operations using different system configurations.
- *Optimization analysis*, which identifies opportunities for modifying an MPI application to use more efficient communications based on configurations generated by the platform analyzer and annotations inserted by software developers in their applications.
- *Program transformations*, which include a collection of program transformation routines invoked by the optimization analyzer to modify the input application for better performance. The modified MPI application is then fed into a vendor compiler (e.g., icc or gcc) to generate a machine executable.

Currently we have implemented the platform analyzer using simple shell scripts that automatically detect the relevant system configurations. Both the optimization analysis and transformations are implemented using POET [14], an interpreted program transformation language designed to support the programmable control and parameterization of source-to-source compiler optimizations. Our POET scripts are extensively parameterized, and variations of alternative implementations can be flexibly generated via different configurations supplied by the platform analyzer.

Our framework currently supports the following optimizing transformations for MPI communications.

```
MPI_Win_fence(win);
MPI_Accumulate(x[0], target, win);
MPI_Accumulate(x[1], target, win);
foo();
MPI_Put(y[0], target1, win);
MPI_Put(y[1], target2, win);
MPI_Put(y[2], target1, win);
MPI_Win_fence(win);
```

Figure 4: Example: MPI one-sided communication

### A. Coalescing of One-Sided Communication Operations

MPI remote memory access (RMA) or one-sided communication follows an epoch-based model for memory consistency. Specifically, a group of processes can expose a part of their memory as public memory, which is referred to as a "window". Then, a process can open an epoch within which it can access such public memory using RMA operations such as MPI_Get, MPI_Put, and MPI_Accumulate. Figure 4 shows an example of such an RMA epoch, which operates inside window $win$ using both MPI_Put and MPI_Accumulate enclosed by a pair of *MPI_Win_fence(win)* synchronizations.

Applications often issue multiple RMA operations per epoch. When compatible, these operations can be combined. For example, the first and third MPI_Put operations in Figure 4 can be combined into a single MPI_put. The end result is fewer messages being sent, thus reducing communication overheads.

Note that the coalescing optimization could be rather complex to apply manually for non-trivial codes, and the degree to which it is beneficial depends on the system and software architectures (e.g., network latency or eager message protocol threshold). It is difficult for the MPI library to apply this optimization due to the lack of application context and the knowledge of when to prioritize latency, bandwidth, or communication/computation overlap. It is also not advisable for developers to hardwire this optimization inside their applications as the transformation could seriously detriment the readability of their applications, and implementations specialized for one platform may perform poorly on a different platform.

Our framework supports the coalescing of all MPI operators (get, put, and accumulate) within a single epoch, driven by the "osc_coalesce" annotation. The transformation algorithm is summarized in Figure 6. Figure 5 illustrates the result of applying the algorithm to *BFS*, a benchmark from Graph500 [3]. The annotation at Line 1 of Figure 5(a) specifies that the MPI one-sided communications in windows $p2\_win$ and $q2\_win$ can be coalesced when appropriate. These communications start with the two invocations of *MPI_Win_fence* at Lines 3–4, contain two *MPI_Accumulate* calls at Lines 9–10, and end with two *MPI_Win_fence* calls at Lines 11–12. The transformed code is shown in Figure 5(b), which contains the following modifications to the original code for each of the annotated windows ($p2\_win$ and $q2\_win$).

- Declare and allocate arrays to save the information for coalescing each communication operator, at Lines 1–2 of Figure 5(b). In particular, four arrays (_data,_cntr,_tgt_disp, and _ctree) are used to track

```
1:#pragma mpi osc_coalesce(p2_win,pred2,int64_t,MPI_INT64_T,
  MPI_COMM_WORLD) (q2_win,queue_bitmap2,unsigned long,
  MPI_UNSIGNED_LONG,MPI_COMM_WORLD)
2:while (1) {......
3:MPI_Win_fence(MPI_MODE_NOPRECEDE, p2_win);
4:MPI_Win_fence(MPI_MODE_NOPRECEDE, q2_win);
5:for (i = 0; i < queue_nwords; ++i) { ......
6: for (bitnum = 0; bitnum < ulong_bits; ++bitnum) {......
7:  for (qelem_idx = 0; qelem_idx < elts_per_queue_bit;
      ++qelem_idx) { ......
8:   for (ei = g.rowstarts[v_local]; ei < ei_end; ++ei) {...
9:    MPI_Accumulate(&local_vertices[v_local],1,MPI_INT64_T,
VERTEX_OWNER(w),VERTEX_LOCAL(w),1,MPI_INT64_T,MPI_MIN,p2_win);
10:   MPI_Accumulate(&masks[((VERTEX_LOCAL(w)/elts_per_queue
_bit)%ulong_bits)],1,MPI_UNSIGNED_LONG,VERTEX_OWNER(w),VERTEX
_LOCAL(w)/elts_per_queue_bit/ulong_bits,1,MPI_UNSIGNED_LONG,
MPI_BOR,q2_win);
   } }  } }
11:MPI_Win_fence(MPI_MODE_NOSUCCEED, q2_win);
12:MPI_Win_fence(MPI_MODE_NOSUCCEED, p2_win);  .......}
```

**(a) Original code with annotation**

```
1:int64_t** p2_win_MIN_data=alloc(int64_t*,comm_world_sz,NULL);
  int **p2_win_MPI_MIN_tgt_disp=alloc(int*,comm_world_sz,NULL)
  int *p2_win_MPI_MIN_cntr = alloc(int*,comm_world_sz,0);
  ctree* p2_win_MPI_MIN_ctree=alloc(ctree,comm_world_sz,NULL);
2:... similar declarations for q2_win \& MPI_BOR...
  while (1) {......
3:MPI_Win_fence(MPI_MODE_NOPRECEDE, p2_win);
4:MPI_Win_fence(MPI_MODE_NOPRECEDE, q2_win);
5:for (i = 0; i < queue_nwords; ++i) {......
6: for (bitnum = 0; bitnum < ulong_bits; ++bitnum) {......
7:   for (qelem_idx = 0; qelem_idx < elts_per_queue_bit;
      ++qelem_idx) {......
8:    for (ei=g.rowstarts[v_local]; ei<ei_end; ++ei) {...
9:      /* optimized MPI_Accumulate call from p2_win */
9.1:    if (p2_win_MIN_data[VERTEX_OWNER(w)]==NULL)
          { ...p2_win_MIN_data[VERTEX_OWNER(w)]= ... }
9.2:    if(p2_win_MIN_data[VERTEX_OWNER(w)]==NULL)
          /*no buffer; send it now using original MPI call*/
          MPI_Accumulate(&local_vertices[v_local],...) ;
        else {
9.3:      if (p2_win_MIN_cntr[VERTEX_OWNER(w)]+1>=CL_FACTOR)
            { ... buffer is full; send it now ...}
9.4:      /* now pack data*/
          int idx = ctree_insert(p2_win_MPI_MIN_ctree,...);
          if( idx==0) {...no conflict; simply pack data...}
          else{ ... merge conflict before packing .... }  }
10:    ... similar MPI_Accumulate on q2_win_BOR_data ...
  } } } }
11:{..send all data in the coalescing buffers for q2_win..}
   MPI_Win_fence(MPI_MODE_NOSUCCEED, q2_win);
12:{..send all data in the coalescing buffers for p2_win..}
   MPI_Win_fence(MPI_MODE_NOSUCCEED, p2_win);
13 ...free the coalescing buffers for p2\_win & q2\_win...;
 ...}
```

**(b) Outline of transformed code**

Figure 5: Example: applying data coalescing to Graph500.

the address, size, destination process, and overlapping of destination addresses, respectively, of the coalescing buffer allocated for each MPI operator.

- Postpone communications until a coalescing buffer is full, at Lines 9–10 of Figure 5(b). The code at Line 9.1 first identifies p2_win_MIN_data[rank], where $rank$ identifies the destination process, as the address of the coalescing buffer and allocates space if the address is NULL. Lines 9.2 and 9.3 send the buffered data if no space is available or if the buffered data size exceeds a preset limit (CL_FACTOR); otherwise, Line 9.4 packs the data into the buffer while using an AVL tree [1] (p2_win_MIN_ctree[rank]) to track conflicting addresses.
- Modify each final synchronization of the communication epoch to send any data that have been buffered but not yet sent, at Lines 11–12 of Figure 5(b).
- Free the coalescing buffers so that their spaces can be used for other purposes, at Line 13 of Figure 5(b).

The above transformations are applied at Steps 5, 8, 9, and 6 of the algorithm in Figure 6, respectively. Finally,

mpi_osc_data_coalesce($win$, $ovlap$, $body$)
win: info. of current one-sided communication window;
$ovlap$: whether different data accesses of $win$ may overlap;
$body$: input code to optimize
(1) extra_comm="";
(2) foreach MPI remote-memory access call $rma$ in $body$:
(3) $coal\_buf = gen\_name(win\_name(win), operator(rma))$;
(4) if $(not\_already\_processed(coal\_buf))$:
remember_processed(coal_buf);
(5) $insert\_new\_decl(body, gen\_dc\_buf\_decl(coal\_buf, win, ovlap))$;
(6) $append\_new\_cleanup(body, gen\_dc\_buf\_free(coal\_buf, ovlap)$;
(7) $extra\_comm \cup = gen\_dc\_extra\_comm(coal\_buf, win)$;
(8) replace_stmt(rma, gen_dc_rma(rma,coal_buf, win, ovlap));
(9) foreach $syn\_end \in find\_osc\_synch\_end(win, body)$:
$insert\_before(syn\_end, extra\_comm)$;
(10) foreach statement $s \in find\_unsafe\_stmts(body, win)$:
$insert\_before(s, extra\_comm)$;

Figure 6: Data coalescing transformation algorithm.

Step 10 of the algorithm searches the annotated block for any statement (e.g., unknown function calls) that may interfere with the communication windows being coalesced. If found, it inserts statements to send off the coalesced data to make sure that any new MPI communications that may be triggered by the unsafe statement do not interrupt the original flow of MPI communications. Developers can use the *independent* annotation at Line 7 of Figure 2 to declare statements as independent of the MPI windows being optimized. These independent statements will be treated as safe statements regarding the windows, so no extra communications will be inserted before them.

The algorithm in Figure 6 adopts the following strategies to guarantee the correctness of the transformation.

- *Grouping of MPI communications*. The algorithm allocates a dedicated buffer for each group of MPI communications that belong to the same window, have the same destination process, and communicate by using the same MPI_Put/MPI_Get or the same reduction operator in MPI_Accumulate. This strategy ensures that all the coalesced data will arrive at their original destinations. Since MPI standard maintains that an epoch does not enforce any sequential ordering of communications, the coalesced communications have the same semantics as the original ones.
- *Overlapping of destination addresses*. When using MPI_Put and MPI_Get, the addresses to place the communicated data on the destination process are not allowed to overlap by MPI standard. When invoking MPI_Accumulate, however, the displacement of the data can indeed overlap, and values sent to the same location need to be accumulated by using the reduction operator of MPI_Accumulate. When the developer does not rule out such overlapping using a "no_overlap" clause, our transformation uses an AVL tree [1] to keep track of the displacements of all the data being communicated via MPI_Accumulate. And whenever an overlapping is detected, local reduction of the data within the coalescing buffer is invoked.
- *Handling unknown function calls*. When an MPI epoch invokes unknown function calls, the functions could include MPI synchronizations that end the epoch being optimized. Our transformation algorithm considers each unknown function call as an unsafe

```
1:#pragma mpi cco MPI_SendRecv(ew_comm,ns_comm)
2:for( i=0; i<niter; i++ ){
3: fill_nssnd_buf( data, nsnd_buf, ssnd_buf, nx, ny, nz );
4: fill_ewsnd_buf( data, esnd_buf, wsnd_buf, nx, ny, nz );
5: if (ns_id > 0 )
6:  MPI_Send(nsnd_buf,ny*nz,MPI_DOUBLE,ns_id-1,0,ns_comm);
7: if (ns_id < ns_comm_size-1)
   MPI_Recv(ns_buf+ny*nz,ny*nz,MPI_DOUBLE,ns_id+1,0,ns_comm,&s);
8: if (ns_id < ns_comm_size-1)
    MPI_Send(ssnd_buf,ny*nz,MPI_DOUBLE,ns_id+1,0,ns_comm);
9: if (ns_id > 0 )
     MPI_Recv(ns_buf,ny*nz,MPI_DOUBLE,ns_id-1,0,ns_comm, &s);
10: ... similar send/recv operations along ew_comm...
11: #pragma mpi indep MPI_SendRecv(ew_comm, ns_comm)
       { compute_inner_stencil(data_next, data, nx, ny, nz);
          update_front_rear(data_next, data, nx,ny,nz); }
12: #pragma mpi indep MPI_SendRecv(ew_comm)
       { update_north_south(data_next,data,nx,ny,nz,ns_buf);}
13: #pragma mpi indep MPI_SendRecv(ns_comm)
       { update_east_west( data_next,data,nx,ny,nz,ew_buf); }
14: update_corners( data_next,data,nx,ny,nz,nsrcv_buf,ew_buf);
15: tmp = data_next; data_next = data; data = tmp;
   }
```

**(a) Original code with annotation**

```
1: for( i=0; i<niter; i++ ){
2: ...declarations for new variables...
3: fill_nssnd_buf( data, nsnd_buf, ssnd_buf, nx, ny, nz );
4: fill_ewsnd_buf( data, esnd_buf, wsnd_buf, nx, ny, nz );
5: if (ns_id > 0 )
   MPI_Isend(nsnd_buf,ny*nz,MPI_DOUBLE,ns_id-1,0,ns_comm,&_r1);
6: if (ns_id < ns_comm_size-1)
   MPI_Irecv(ns_buf+ny*nz,ny*nz,MPI_DOUBLE,ns_id+1,0,ns_comm,&_s1);
7: if (ns_id < ns_comm_size-1)
   MPI_Isend(ssnd_buf,ny*nz,MPI_DOUBLE,ns_id+1,0,ns_comm,&_r2);
8: if (ns_id > 0 )
   MPI_Irecv(ns_buf,ny*nz,MPI_DOUBLE,ns_id-1,0,ns_comm,&_s2);
9:  ... similar Isend/Irecv operations along ew_comm...
10:#pragma mpi indep MPI_SendRecv(ew_comm,ns_comm)
     { compute_inner_stencil(data_next, data, nx, ny, nz);
        update_front_rear(data_next, data, nx,ny,nz); }
11: ... MPI_Wait operations for ns_comm Isend/Irecv ...
12:#pragma mpi indep MPI_SendRecv(ew_comm)
      { update_north_south(data_next,data,nx,ny,nz,ns_buf);}
13: ... MPI_Wait operations for ew_comm Isend/Irecv ...
14:#pragma mpi indep MPI_SendRecv(ns_comm)
      { update_east_west( data_next,data,nx,ny,nz,ew_buf); }
15:update_corners( data_next,data,nx,ny,nz,nsrcv_buf,ew_buf);
16:tmp = data_next; data_next = data; data = tmp;
 }
```

**(b) Outline of transformed code**

Figure 7: Example: comp/comm overlapping.

statement and automatically inserts communications to send the coalesced data before the invocation, unless these calls have been annotated by developers as independent of the MPI window being optimized.

### *B. Overlapping of Computation and Communications*

MPI provides several nonblocking operations, e.g., `MPI_Isend` and `MPI_Irecv`, to overlap computation and communication. However, the right amount of computation to be perfectly overlapped with communication is specific to a given platform, the destination process of the communication (e.g., whether the process can be reached over shared memory or the network), and the kind of data being communicated (e.g., contiguous data vs. noncontiguous data). Such constraints make it hard to portably decide on the right amount of computation that needs to be interleaved with communication operations.

Our framework allows developers to implement their algorithms using simple blocking MPI_send/recv operations and then simply insert a few annotations to enable the transformations to be automatically applied when desired.

Within our framework, the computation-communication overlapping transformation is driven by the *cco* annotation, illustrated at Line 1 of Figure 7(a), which specifies that the MPI_Send and MPI_Recv operations with either

mpi_comp_comm_overlap($com\_grp$,$input$)
  $com\_grp$: communication group specification;
  $input$: innermost block containing $com\_grp$;
(1) $comms = find\_matching\_comms(com\_grp, input)$;
(2) $indep\_stmts = find\_indep_stmts(com\_grp, input)$;
(3) foreach MPI send/recv call $c \in comms$:
(4)  (decl,ncomm,wait) = mpi_blocking_2_nonblocking(c);
(5)  (before,in1,in2,after) = split_stmt_block(input, ncomm,wait);
(6)  insert_new_decl(input, decl);
(7)  move_comm_up(in1, indep_stmts, before);
(8)  move_comm_down(in2, indep_stmts, after);

Figure 8: Algorithm: overlapping comp & comm.

*ew_comm* or *ns_comm* as handles in the annotated block could be optimized for a 2-D stencil kernel. Figure 7(b) outlines the result of the transformation by invoking the algorithm in Figure 8 to perform the following steps.

1) Find all the MPI operations that match the specifications included in the *cco* annotation. In Figure 7(a), these include all the MPI_Send/Recv invocations with $ew\_comm$ or $ns\_comm$ as handles.
2) Find all the statements that are independent of each communication based on the *independent* annotations contained within $input$. In Figure 7(a), they include statements at Lines 11 and 12 for $MPI\_SendRecv(ew\_comm)$ and statements at Lines 11 and 13 for $MPI\_SendRecv(ns\_comm)$.
3) For each MPI communication $c$ found at Step 1, perform transformation Steps 4–8.
4) Replace $c$ with an equivalent combination of an asynchronous operation ($ncomm$) and a MPI_Wait operation ($wait$). Then, create declarations ($decl$) for the new request and handle variables.
5) Break up the statements in $input$ into four groups, $before$, $in1$, $in2$, and $after$, where $before$ and $after$ contain statements that appear before $ncomm$ and after $wait$ respectively, and $in1$ and $in2$ contain the $ncomm$ and the $wait$ operations respectively, so that $in1$ and $in2$ can be easily interchanged with statements immediately before or after them later.
6) Insert new variable declarations. In Figure 7(b), these declarations are inserted at Line 2, the beginning of the innermost body of the annotated block.
7) Safely move up $in1$, which contains the asynchronous operation $ncomm$, so that it can be evaluated as early as possible. In Figure 7(b), the asynchronous MPI operations for $ew\_comm$ and $ns\_comm$ are placed at Lines 5–9, since they cannot be moved further up in the original code.
8) Safely move down $in2$, which contains the MPI $wait$ operation for $ncomm$, so that it can be evaluated as late as possible. In Figure 7(b), the wait operations for $ns\_comm$ and $ew\_comm$ are moved to Line 11 and Line 13, respectively.

The correctness of the transformation algorithm in Figure 8 hinges on careful implementations of Steps 5, 7, and 8. Note that in the optimization analysis algorithm in Figure 3, the innermost body of the annotated block is used as the input parameter when invoking the *cco* transformation algorithm. Therefore, the transformation rearranges only the relative execution order of the statements within the

```
#pragma mpi rma(win,buf,int,MPI_INT,wsize,wrank)
 {
    MPI_Win_lock(MPI_LOCK_SHARED, i, 0, win);
    for (j = 0; j < BUF_PER_PROC ; j++) {
       MPI_Put(&wrank,1,MPI_INT,i,base+j,1,MPI_INT,win);
     }
    MPI_Win_unlock(i, win);
 }
```

**(a) Remote memory access with annotation**

```
#pragma mpi local_ldst(win,buf,int,MPI_INT,wsize,wrank)
            no_overlap
  {
     MPI_Win_lock( MPI_LOCK_EXCLUSIVE, i, 0, win );
     for (j = 0; j < BUF_PER_PROC ; j++) {
         buf[base+j] = wrank;
     }
     MPI_Win_unlock(i, win);
  }
```

**(b) Local load/store with annotation**

Figure 9: Example: RMA and local load/store operations.

innermost loop body without changing the iteration order of their surrounding loops. The algorithm requires that all the annotated MPI communications be immediately nested inside the innermost body. Specifically, they may be inside if-conditionals but not surrounded by any additional loops. Consequently, no dependence is violated by separating out these communication operations from other statements that may lie in the same if-conditional. Our algorithm currently relies on the *independent* annotations inserted by developers to determine the safety of moving the communication operations. The algorithm, however, does examine the MPI operations and determine that these operations are independent of each other if they operate on distinct buffers and use different communicators (handles).

### C. Transformations for Cache-coherent Architectures

MPI allows direct load/store accesses to the buffers associated with one-sided communication windows, illustrated in Figure 9(b). However, as shown in this example, these operations often need to be enclosed inside an exclusive lock to avoid conflict with the other nonlocal accesses. This conservative strategy enables portability across a wide variety of systems but imposes an unnecessary overhead on systems with cache-coherent hardware, where concurrent load/store and RMA operations are permitted as an extension to the MPI standard.

Figure 9 illustrates two equivalent MPI epochs that modify the memory ($buf$) associated with an MPI one-sided communication window. Both versions work correctly on all platforms irrespective of their cache-coherence support, but both need to pay a performance penalty to ensure their portability. On a cache-coherent system, to improve performance, the MPI_Put operations in (a) should be replaced with local load/stores, and the exclusive lock in (b) with a shared lock. On a non-cache-coherent system, the version in (a) is likely to perform better than (b) because the extra overhead of the exclusive lock in (b). Based on the *rma* and *local_ldst* annotations in both versions, our framework can automatically specialize these implementations based on the requirement of the underlying platform.

The transformation algorithms in Figure 10 are straightforward and include the following components.

```
mpi_rma_2_ldst(win,input)
  win: info. of current one-sided communication window;
  input: input code to optimize
(1) foreach MPI remote-memory access call rma in input:
(2)    cond = is_equal(get_comm_rank(win),get_des_rank(rma));
(3)    repl = gen_ldst_stmt(get_src(rma), get_des(rma), get_op(rma));
(4)    replace_stmt(rma, gen_if(cond, repl, rma));

mpi_ldst_coh(win,ovlap,input);
  win: info. of current one-sided communication window;
  ovlap: whether different data access of win may overlap;
  input: input code to optimize
(1) if (ovlap) return;
(2) foreach MPI_Win_lock call l in input:
(3)    if (get_lock_type(l) == MPI_LOCK_EXCLUSIVE):
         change_lock_type(l, MPI_LOCK_SHARED);

mpi_ldst_incoh(win,ovlap,input)
  win: info. of current one-sided communication window;
  ovlap: whether different data access of win may overlap;
  input: input code to optimize
(1) foreach assignment stmt s: l=r in input:
      if (is_array_access(l) && get_array(l) == get_buf(win)):
        replace_stmt(s, gen_rma_call("MPI_Put", win,addr_of(r));
      else if (is_array_access(r) && get_array(r) == get_buf(win)):
        replace_stmt(s, gen_rma_call("MPI_Get", win,addr_of(l));
(2) if (ovlap is false):
      foreach MPI_Win_lock call l in input:
        if (get_lock_type(l) == MPI_LOCK_EXCLUSIVE):
          change_lock_type(l, MPI_LOCK_SHARED);
```

Figure 10: Algorithm: translating RMA & local load/store.

1) *Translating RMA operations to local loads/stores (mpi_rma_2_ldst)*, which is driven by the *rma* annotation and applied when the underlying platform supports cache coherence. The transformation searches for each RMA operation and then replaces it with an if-conditional so that when the rank of the local process equals the rank of the destination process, a local load/store operation is used instead.
2) *Eliminating exclusive locks (mpi_ldst_coh)*, which is driven by the *local_ldst* annotation and invoked to replace the exclusive locks surrounding local load/store operations with shared locks if the underlying platform does support cache coherence and if the memory accessed by the operations never overlaps.
3) *Translating local load/store operations to RMA operations (mpi_ldst_incoh)*, which is driven by the *local_ldst* annotation and applied when the underlying platform does not support cache coherence. The transformation includes two steps: (1) for each local assignment involving the communication buffer, replace the assignment with an equivalent invocation of MPI_Put or MPI_Get; and (2) if the buffers being accessed in the MPI epoch never overlap, change all the lock types from exclusive locks to shared locks.

### D. Generality of the Framework

Among the three groups of optimizations supported by our framework, data-coalescing and computation-communication overlapping have been well-acknowledged as important optimizations for MPI applications, while the conversion between RMA and local load/store operations targets at specializing MPI applications for varying platforms in order to improve their performance portability. Manually applying these optimizations is cumbersome

Table I: Benchmark details

| Name | Benchmark | Description | Transformation Applied |
|---|---|---|---|
| bfs | graph500 | breadth-first search of an undirected graph | OSC data coalescing |
| rma-ldst | synthetic | random communications using MPI_Put | RMA vs. local ld/st translation |
| stencil | synthetic | 3D stencil using MPI send/recv | comp-comm overlapping |
| FT | NAS | 3D PDE using MPI all-to-all | collective vs. one-sided communication |

and compromises both the readability and the portability of applications. By allowing developers to single out important regions of code that implement epochs of MPI communications, a lightweight program transformation system has been used to automatically specialize MPI applications and thus significantly enhance their performance portability across different systems. With the help of developer supplied algorithmic hints, the overhead of applying the necessary program transformations becomes negligible (linear to the size of code regions being transformed). The scalability of the overall approach can be enhanced by utilizing optimizing compilers to automatically determine the dependence constraints within the application and by utilizing performance modeling techniques to automatically identify hot regions of code to specialize, both of which are topics of our future work.

## IV. Experimental Results

Attaining performance portability is the ultimate goal of high performance computing. While the small collection of program transformations currently supported within our framework is far from addressing the full range of performance portability issues in MPI programming, our proposed lightweight annotation-based approach serves to demonstrate a migration path that may eventually lead to fully portable MPI applications. We have studied four benchmarks, shown in Table I, to validate the potential of this annotation-based approach and to demonstrate the sensitivity of our application-level optimizations to varying runtime configurations. Three of these benchmarks, *bfs*, *stencil*, and NAS *FT*, represent well-acknowledged important scientific computations, while *rma-ldst* is a synthetic kernel we developed to study the sensitivity of MPI RMA vs local load/store operations. We have optimized *bfs*, *rma-ldst,*, and *stencil*, by manually inserting annotations into selected functions and then applying our framework to automatically optimize their MPI communications for varying platforms. The NAS FT benchmark, however, was transformed manually without using our framework, as its MPI communications are scattered across several procedures and thus beyond the capacity of our existing framework. Since supporting automated inter-procedural optimization of MPI communications is a topic of our future work, we present our performance study of FT.

We evaluated our benchmarks on two supercomputers at Argonne National Laboratory: *Fusion*, a cluster with 320 compute nodes, each with two Intel Nehalem Quad-Core 2.6 GHz processors and 36 GB of memory, interconnected via InfiniBand QDR at 4 GB/s per link; and *Surveyor*, a Blue Gene/P system with 1024 compute nodes, each with a quad-core 850 MHz PowerPC 450 processor and 2 GB memory. On both machines, we compiled the benchmarks using the default *mpicc* compilers, which use *mvapich2* 1.4.1 and gcc 4.1.2 on the Fusion machine and use gcc 4.4.6 on Surveyor. We compiled all benchmarks using the -O2 optimization flag, which enables all the relevant advanced optimizations within gcc but avoids some overly aggressive optimizations in -O3 which might actually slow down the applications. For Graph500 and FT, we present the default performance metrics reported by the benchmarks, specifically the average time of running the Graph500 *bfs* kernel 64 times and the total elapsed time of running 6 iterations of the FT computation. For both the *stencil* and the *rma-ldst* benchmarks, we report their average performance across 10 different runs.

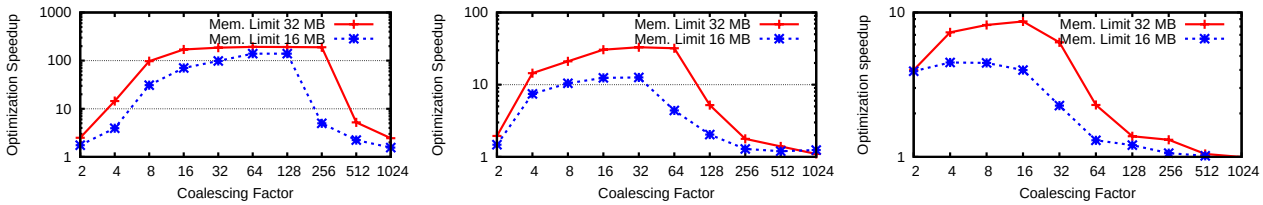
### A. Optimizing the Graph500 Benchmark

The Graph500 benchmark suite [3] currently includes one reference graph algorithm, a breadth-first search (*bfs*) of undirected graphs. We inserted an *osc_coalesce* annotation inside an implementation of the search kernel that uses MPI one-sided communications, an outline of which is shown in Figure 5(a). The original implementation of *bfs* in Figure 5(a) has two MPI_Accumulate invocations, each sending a single data item, at the innermost loop. Although these communications are overlapped with other computations (omitted as ..... in Figure 5), the communication overhead may be too high. Our framework automatically coalesced the many small messages so that an actual communication is triggered only when the size of the coalesced message exceeds a predetermined threshold (CL_FACTOR at Line 9.3 of Figure 5(b)).

An overhead of applying data-coalescing is the extra memory required. Our framework allows the user to specify a limit on the overall size of memory used for the coalescing buffers. For example, if this limit is 64 MB when using 64 processes, each process is allowed to use at most 1 MB for message coalescing. If the process runs out of memory, it stops coalescing.

Figure 11 shows the overall execution time of Graph500 when running both the original and the coalesced implementations of *bfs*, using a variety of different coalescing factors and memory limit, on the Fusion machine using 64, 128, and 256 processes, respectively, with 8 processes allocated to each node of the cluster. The input is an undirected graph of $2^{16}$ vertices and $2^{16} * 12$ edges. Because Graph500 uses some Assembly intrinsics that work only on Intel architectures and because the Surveyor machine uses PowerPC processors, we were not able to collect results for *bfs* on the Surveyor machine.

From Figure 11, the best optimization speedup from data coalescing is around 190x when using 64 processes. Because the graph size is constant in all the evaluations, as the number of processes increase, each process has a smaller amount of work, resulting in fewer remote memory accesses and thus less optimization benefit.




(a) $p = 64$ (b) $p = 128$ (c) $p = 256$

Figure 11: Result of applying data coalescing to one-sided communications of Graph500 *bfs* on machine *Fusion*.

From all three graphs in Figure 11, having a larger amount of memory available for coalescing does increase the effectiveness of the optimization, especially when using a large number of processes. In order to obtain the best speedup, the coalescing factor needs to be adjusted based on the amount of memory available. In particular, when the memory demand is high, it is better to make coalescing buffers smaller (i.e., using smaller coalescing factors) so that the coalesced data can be sent sooner and thus free up more available memory for other needs.

In summary, the performance benefit of data coalescing depends on the number of small messages that are sent to the same destinations and thus could be coalesced; the speedup could be orders of magnitude. It is difficult to automatically determine the best memory limit and coalescing factors a priori because their best configurations are sensitive to the internal memory demands of the application, although a reasonable default configuration can be used. Note that while data coalescing appears to be beneficial for Graph500, it can be rather platform-sensitive when considered together with overlapping computation with communication, discussed in Section IV-D.

### B. Optimizing Stencil Computations

Stencil computations are among the most important kernels of scientific computing and are good candidates for exploring overlapping of computation with communications since each process needs to communicate only with its neighbors about their boundary values. An outline of the original stencil code we used is shown in Figure 7(a), which uses MPI blocking primitives for better readability. Our optimizations can similarly work if the original code is written using MPI Isend/Irecv operations.

Figure 12 shows the performance of both the original 3-D stencil and the transformed code, which uses asynchronous MPI operations to better overlap the communications with the inner stencil computation of each process. The stencil size is 2048*2048*4096 on Fusion and 2048*2048*1024 on Surveyor, since each node on Surveyor has smaller memory than the nodes on Fusion.

The transformed code has consistently performed better, with 15% to 2.4x speedup, on both machines. However, one anomaly exists when using 256 processes on Surveyor, where the original blocking communications result in less stress on the high memory demand of each process. This demonstrates that while overlapping computation with communication typically results in better performance of MPI applications, using blocking communications can sometimes perform better.

Since the stencil size stays constant on each machine, as more processes participate in the computation, the ratio between the amount of computation and communication per process decreases, and it becomes increasingly important to hide the latency of the communications in order to reduce their overhead, therefore resulting in more significant performance benefit from the overlapping transformation.

In summary, while it is profitable to overlap communications with computations in most situations, blocking synchronizations can sometimes outperform asynchronous operations. It is advantageous for developers to use annotations to facilitate the optimization instead of directly modifying their applications since MPI blocking communications are easier to debug and maintain and allows the logistics of local computations to be better grouped together. Further, this optimization could become highly platform-sensitive when considered together with the data coalescing transformation, discussed in Section IV-D.

### C. Cache-Coherence-Aware Transformations

To compare the efficiencies of different MPI operations on varying platforms, we developed a synthetic benchmark, illustrated in Figure 9(a), which randomly sends data to other processes using the MPI_Put operation. The code is then annotated with our *rma* annotation so that it can be automatically translated to equivalent local load/store operations on platforms that support cache coherence.

Figure 13 shows the performance of running our synthetic benchmark to perform 1 million RMA accesses (99% are local memory accesses), with and without our cache-coherence-aware transformation using 8–1024 processes. Since both machines support cache coherence, in all cases the local-load/store version automatically generated by our framework performed significantly better than the RMA version.

Among the three groups of optimizations currently supported in our framework, the cache-coherence-aware optimizations are the most platform sensitive and therefore need to be automated via application-level transformations. Our annotation-based framework is lightweight and can be supported as a preprocessor of MPI applications within a *mpicc* compiler before the vendor compiler (e.g., gcc) is invoked.

### D. Optimizing The NAS FT Benchmark

While both the data-coalescing and computation-communication overlapping optimizations are profitable in most situations, combining them often results in a tradeoff in performance. In particular, while frequently

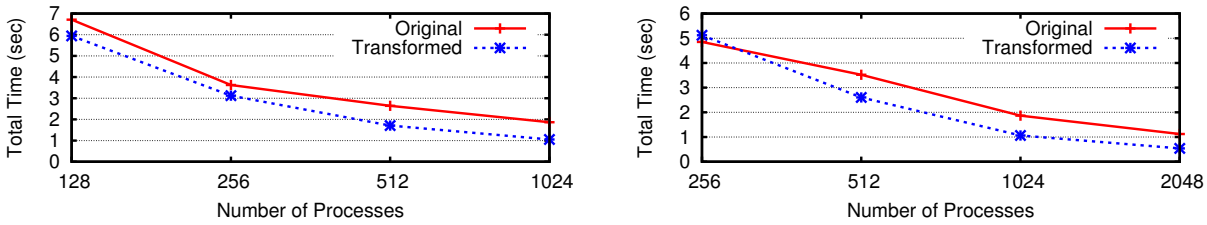



Figure 12: Result of applying computation-communication overlapping to *stencil* on (left) Fusion and (right) Surveyor.

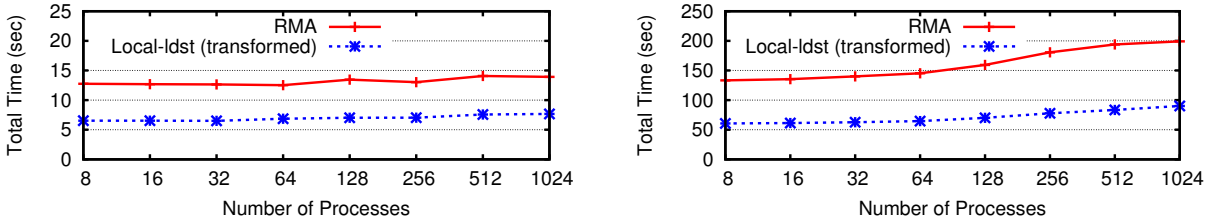



Figure 13: Result of translating RMA to local load/store operations on (left) Fusion and (right) Surveyor.

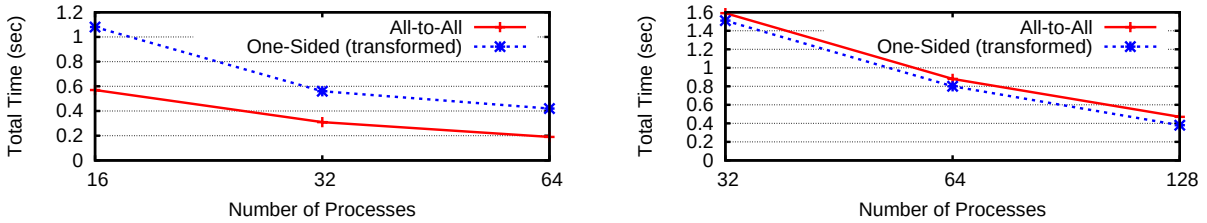



Figure 14: Result of optimizing the NAS FT benchmark on (left) Fusion and (right) Surveyor.

```
for(...){
   for(each i=...)
      for(each j=...)
         computation using array(i,j);
    exchange array using blocking communications
}
```

**(a) Original code using all-to-all communications.**

```
for(...){
   for(each i=...)  {
      for(each j=...)
         computation using array(i,j);
      exchange array(i) using MPI one-sided communications
}  }
```

**(b) Optimized code using one-sided communications.**

Figure 15: Optimizing MPI communications in NAS FT.

sending small messages does not fully utilize the network bandwidth, it is easy to hide the latency of frequent small communications by overlapping them with independent computations. On the other hand, while coalescing small messages may reduce the overhead of communication, the coalesced messages may not overlap well with local computations due to the delays in sending the messages. We demonstrate the issues in making optimization decisions using FT from the NAS parallel benchmarks [4]. Because MPI communications within FT are scattered across a number of subroutines, we have manually applied the transformations for the performance study.

Figure 15(a) illustrates the original communication pattern of NAS FT, which solves a 3-D partial differential equation (PDE) using forward and inverse FFTs. It first computes an entire 3D array and then scatters the data across all processes using MPI_Alltoall. In contrast, the manually transformed code in (b) scatters a single row of the 3D array immediately after each row of the array has be computed, using MPI one-sided communications. In essence, the original code coalesces all the communications and wait until the end before sending all the data using a single MPI_Alltoall operation, while the transformed code breaks up the communications into smaller messages to enable better overlapping with local computations.

Figure 14 shows the performance of the original and manually transformed code on both the Fusion and Surveyor machines, where the transformed code was able to attain 5%-25% speedup on Surveyor but resulted in almost 2x slowdown on Fusion. The results clearly indicate that careful optimization decisions need to be made based on the underlying node structures and network connections of each parallel platform, as the efficiencies of different MPI operations vary significantly across platforms.

## V. Related Work

Existing work on optimizing MPI codes mostly focused on producing efficient implementations of MPI libraries. Gropp et al. [8] studied options and the associated cost of implementing the synchronization mechanisms of MPI one-sided communications. Almási et al. [2] optimized implementations of MPI collectives targeting microbenchmarks on Blue Gene/L. Sur et al. [12] exploited RDMA read and selective interrupt-based asynchronous progress in order to provide better computation/communication overlap on InfiniBand clusters. Faraj [7] presented a system that produces efficient MPI collective communication implementations customized for each platform by automatically generating topology-specific routines and then

empirically selecting the best implementations. Xu and Kuang [13] developed a systematic method to estimate the communication overhead of three message-passing operations (point-to-point communication, collective communication, and collective computation). Our work focuses on utilizing developer annotations and automated program transformations to improve the use of MPI operations within user applications instead of MPI libraries.

Similar to our work, Danalis et al. investigated program transformations directed toward improving communication-computation overlap in MPI applications that use collective operations [5] and proposed the development of MPI-aware compilers that exploit the knowledge of MPI call effects [6]. Preissl et al. [10] explored automated identification of communication patterns from dynamically generated MPI traces to support debugging of communications and enable performance optimizations [11]. Our work also focuses on optimizing MPI applications but emphasizes automatically supporting their performance portability through a light-weight annotation-based program transformation framework.

## VI. Conclusions

This paper presents an annotation-based program transformation framework where users can annotate MPI applications with concise information about the communication mechanisms used inside varying blocks of statements, so that these blocks can be modified to use alternative communication mechanisms in MPI in order to achieve portable high performance on different platforms. Our framework currently supports three optimizations: data coalescing for MPI one-sided communications, overlapping communications with independent computations, and automatic selection of communication operators based on the cache-coherence support of the underlying platform.

## Acknowledgments

This work was supported through resource grants from the Argonne Leadership Computing Facility, the Argonne Laboratory Computing Resource Center, the Oak Ridge National Center for Computational Sciences, and the National Energy Research Scientific Computing Center, by the NSF through grants CCF-0747357, CCF-0833203, and CNS-0855247, and by the U.S. Department of Energy under grant DE-SC0001770 and contracts DE-AC02-06CH11357, DE-AC05-00OR22725, and DE-ACO6-76RL01830.

## References

[1] G. M. Adelson-Velskiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathmatics*, 3(5):1259–1262, 1962. Translated from Russian *Doklady, Akademiĭ Nauk SSSR* **146**:263–266.

[2] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of mpi collective communication on bluegene/l systems. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 253–262, New York, NY, USA, 2005. ACM.

[3] D. Bader, J. Berry, S. Kahan, R. Murphy, E. Riedy, and J. Willcock. Graph 500 list, 2010. www.graph500.org/specifications.

[4] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber an dH. Simon, V. Venkatakrishnan, and S. Weeratunga. Nas parallel benchmarks, 1994. http://www.nas.nasa.gov/publications/npb.html.

[5] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to parallel codes for communication-computation overlap. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos. Mpi-aware compiler optimizations for improving communication-computation overlap. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 316–325, New York, NY, USA, 2009. ACM.

[7] Ahmad Faraj and Xin Yuan. Automatic generation and tuning of mpi collective communication routines. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 393–402, New York, NY, USA, 2005. ACM.

[8] William Gropp and Rajeev Thakur. An evaluation of implementation options for mpi one-sided communication. In *Proceedings of the 12th European PVM/MPI users' group conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI'05, pages 415–424, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] MPI Forum. MPI: A message-passing interface standard version 2.2. Technical report, Univ. of Tennessee, Knoxville, September 2009.

[10] Robert Preissl, Thomas Köckerbauer, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. Detecting patterns in mpi communication traces. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 230–237, Washington, DC, USA, 2008. IEEE Computer Society.

[11] Robert Preissl, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. Transforming mpi source code based on communication patterns. *Future Gener. Comput. Syst.*, 26:147–154, January 2010.

[12] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. Rdma read based rendezvous protocol for mpi over infiniband: design alternatives and benefits. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 32–39, New York, NY, USA, 2006. ACM.

[13] Zhiwei Xu and K. Hwang. Modeling communication overhead: Mpi and mpl performance on the ibm sp2. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 4(1):9–24, 1996.

[14] Qing Yi. POET: A scripting language for applying parameterized source-to-source program transformations. *Software: Practice & Experience*, pages 675–706, May 2012.