MPI-aware Compiler Optimizations for Improving Communication-Computation Overlap

Anthony Danalis[*]
University of Delaware
Newark, DE 19716
danalis@cis.udel.edu

Lori Pollock
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu

Martin Swany
University of Delaware
Newark, DE 19716
swany@cis.udel.edu

John Cavazos
University of Delaware
Newark, DE 19716
cavazos@cis.udel.edu

ABSTRACT

Several existing compiler transformations can help improve communication-computation overlap in MPI applications. However, traditional compilers treat calls to the MPI library as a black box with unknown side effects and thus miss potential optimizations. This paper's contributions enable the development of an MPI-aware optimizing compiler that can perform transformations exploiting knowledge of MPI call effects to increase communication-computation overlap. We formulate a set of data flow equations and rules to describe the side effects of key MPI functions so an MPI-aware compiler can automatically assess the safety of transformations. After categorizing existing compiler transformations based on their effect on the application code, we present an optimization algorithm that specifies when and how to apply these optimizing transformations to achieve improved communication-computation overlap. By manually applying the optimization algorithm to kernels extracted from HYCOM and the NAS benchmarks, we show that even when transforming these highly optimized codes, execution time can be decreased by an average of over 30%.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*; B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems)—*Parallel I/O*

General Terms

Algorithms, Theory

[*]Currently with the University of Tennessee and the Oak Ridge National Laboratory.

This research was funded by NFS grant CSR ASE 0509170.

1. INTRODUCTION

The use of explicitly parallel, MPI applications has been the most popular way of harnessing high performance computing power for over a decade. The widespread acceptance of MPI as well as the high performance it offers, make it likely that MPI will remain the dominant parallel programming paradigm in the near future, despite the existence of alternative approaches such as Partitioned Global Address Space (PGAS) languages [15, 19, 29]. However, unlike PGAS languages, MPI offers its benefits only at the library and system layer, without utilizing information from the application layer. Furthermore, all MPI implementations are libraries including binary objects. As a consequence, when a compiler translates and optimizes a parallel application with calls to MPI, it will treat the MPI calls as a black box and avoid optimizing the calls and in some cases the code that surrounds them.

Keeping MPI at the library and system layer simplifies the work of MPI implementers and makes MPI compiler-independent. Unfortunately, many optimization opportunities are lost. Most modern compilers can perform the necessary control and data flow analysis to determine the earliest legal point to use or define an array, and the latest legal point before the array is used or redefined. If the array is a message buffer, that analysis can reveal the earliest location the data transfer involving the message can be initialized, and the latest location the transfer has to be finalized. PGAS languages use this type of analysis to achieve communication-computation overlap and to reduce communication cost [7, 8]. However, traditional compilers do not manipulate calls to library functions, therefore, no such optimizations will take place in an MPI program by any mainstream compiler, however trivial, or beneficial the optimization. However, the behavior of MPI functions, and their effects on the code that calls them, are well defined by the MPI standard. A witness to that is the fact that a human programmer, with knowledge of the MPI standard, can safely transform an MPI application either by applying traditional compiler transformations to code snippets that include MPI calls, or by transforming the MPI calls themselves. We argue that an MPI-aware compiler could perform safe transformations on MPI function calls and thus reduce the communication delays by enabling communication-computation overlap.

In this paper, we examine how compiler transformations can be applied to MPI calls in parallel application codes, as well as the computation that includes these calls, in order to improve communication-computation overlap. Namely, the contributions of this paper are the following:

- We categorize existing compiler transformations based on their role towards improving communication-computation overlap in MPI programs.
- We describe the behavior of MPI functions through a set of data flow equations and rules that can be used as the basis for an optimizing compiler to automatically transform MPI programs.
- We present an optimization algorithm that specifies when and how to apply the described program transformations within an optimizing compiler.
- We present experimental results from manually applying the optimization algorithm to kernels from complex scientific codes, and demonstrate the benefits and risks associated with the different transformations, and their role within the chain of transformations.

2. TRANSFORMATIONS FOR COMMUNICATION-COMPUTATION OVERLAP

We define the term *Overlap Window* to be a region of an MPI code between a call to a communication-initiation function (i.e., `MPI_Isend()`, or `MPI_Irecv()`) and a call to the matching communication-termination function (i.e., `MPI_Wait()`). The overlap window defines the maximum computation that can be overlapped with a given MPI data transfer. If an optimizing compiler can increase the temporal length of the overlap window, the potential for communication-computation overlap will increase. Several different compiler transformations can be applied to MPI codes to increase, or enable the increase of, the temporal length of the overlap window of different data transfers. These transformations can be categorized into six classes based on the effect they have on the target code. These classes are:

Transformation of Blocking MPI calls to Non-Blocking: Since non-blocking calls do not allow for communication-computation overlap, an early transformation to enable further optimization is the translation of blocking operations to an equivalent pair of a non-blocking operation and a corresponding wait.

Communication Library Specific Transformations: The communication-computation overlap achieved by a data transfer can be improved, if a compiler substitutes calls from a specialized communication library (such as Gravel [10], GASNet [4], or ARMCI [28]), for the MPI calls that perform the data transfer.

MPI Collective Call Decomposition: In a system where the network does not support native collective operations, such as *all-to-all* and *gather*, in hardware, the communication library must internally implement each collective data exchange pattern through a sequence of point-to-point communication operations. If this sequence of point-to-point operations is inlined into the program and thus exposed to the application layer, the compiler can restructure the code, optimizing the individual transfers by overlapping them with computation. In systems were collective operations are not equivalent to a sequence of point-to-point operations, non-blocking collectives [22] can be used to leverage communication-computation overlap.

Code Motion for Overlap Window Expansion: The most straightforward transformation for creating or increasing communication-computation overlap is code motion. This class includes transformations that hoist non-blocking, data transfer initiation calls (such as `mpi_isend()` and `mpi_irecv()`) earlier in the execution path of the application and sink transfer termination calls (such as `mpi_wait()`) later in the execution path.

Variable Cloning: Data dependencies between different communication operations can limit or prevent code motion. However, in some cases, these data dependencies can by removed via transformations that create one or more clones of the variables involved in these dependencies (similar to register renaming).

Loop Nest Optimizations (LNO) to create Independent Code Blocks: Data dependencies between the communication calls and the computation intensive parts of an application might be such that no code motion can be performed to expand the overlap window. However, LNO can create additional opportunities for expanding the overlap window, by creating communication independent computation. That can be achieved by separating a computation loop with dependencies on the communication into multiple loops such that at least one of the resulting loops includes only computation with no dependencies on the communication.

Additional overlapping can be achieved via *Communication and Computation Tiling & Pipelining* (CCTP). If a data exchange operation is partitioned into several smaller transfers (i.e., communication tiling) and pipelined with the computation, after the computation is also partitioned into corresponding smaller units of computation (i.e., computation tiling), the data exchange will be overlapped with computation.

3. EXAMPLE AND CHALLENGES

Consider the loop of Figure 1(a). A compiler can easily determine that it is safe to perform loop fission. Consider now that the original loop also includes a call to function `F()` between the two stores, as in Figure 1(b). The compiler would have to examine the body of function `F()` to assess if it is still legal to perform loop fission. However, if the body of `F()` is not available to the compiler, the compiler must conservatively estimate the effects of function `F()`. Compilers typically avoid transforming code that includes calls to unknown library functions, unless special analysis of the library has taken place, as in telescoping languages [25].

In contrast, a human programmer can safely transform a program with calls to functions of a well-defined library such as MPI. Regardless of the MPI implementation, the behavior of each MPI function is well-defined with respect to how the arguments are used or manipulated by the function body and the side effects to the calling code. Consider the loop mentioned previously, replacing the call to function `F()` by a call to function `MPI_Isend()` followed by a call to `MPI_Wait()`, as shown in Figure 1(c). A human developer can easily deduce that loop fission can safely separate the assignment to `B[]` into a loop of its own, as in Figure 1(d), and then the `MPI_Wait()` into a loop of its own, as in Figure 1(e). Finally, a human knows that code motion can safely move the loop with the wait operation to the point after the loop that assigns into array `B[]`, as in Figure 1(f).

Performing this series of transformations can have a positive effect on performance by enabling communication-computation overlap. Furthermore, all the transformations are widely available compiler transformations, as is the data flow analysis required to guarantee their safety in sequential programs. Nevertheless, no traditional compiler would perform these transformations, because it requires knowing about `MPI_Isend()` and `MPI_Wait()` side effects.

4. SAFETY ANALYSIS FOR MPI PROGRAM TRANSFORMATION

Enabling a compiler to optimize MPI applications is achievable if we can enable the compiler to recognize the semantics of MPI functions that a human developer uses to assess which transformations are safe. We argue that if a compiler respects all the rules outlined in this section, the transformations described earlier can

```
do j=1,10
  A[j] = j
  B[j] = 2*j
enddo
```

(a) Simple loop

```
do j=1,10
  A[j] = j
  F(j)
  B[j] = 2*j
enddo
```

(b) Loop with function call

```
do j=1,10
  A[j] = j
  MPI_Isend(A[j],1,req[j],...)
  MPI_Wait(req[j],...)
  B[j] = 2*j
enddo
```

(c) Loop with MPI function calls

```
do j=1,10
  A[j] = j
  MPI_Isend(A[j],1,req[j],...)
  MPI_Wait(req[j],...)
enddo
do j=1,10
  B[j] = 2*j
enddo
```

(d) After loop fission

```
do j=1,10
  A[j] = j
  MPI_Isend(A[j],1,req[j],...)
enddo
do j=1,10
  MPI_Wait(req[j],...)
enddo
do j=1,10
  B[j] = 2*j
enddo
```

(e) After 2nd loop fission

```
do j=1,10
  A[j] = j
  MPI_Isend(A[j],1,req[j],...)
enddo
do j=1,10
  B[j] = 2*j
enddo
do j=1,10
  MPI_Wait(req[j],...)
enddo
```

(f) Loops after code motion

Figure 1: Potential transformations with knowledge of MPI call side effects

be automatically applied by a compiler on an MPI application and the transformed code will be semantically equivalent to the original MPI code. The following analysis focuses on MPI calls that exchange primitive types, or derived data types that are contiguous in memory.

MPI function equivalence rules: A call to a blocking MPI function (e.g. `MPI_Send()` or `MPI_Recv()`) is equivalent to a call to the corresponding non-blocking function (e.g. `MPI_Isend()` or `MPI_Irecv()`), immediately followed by `MPI_Wait()`. The arguments passed to the non-blocking function must be the same as those passed to the original blocking function with the additional argument, *request*. The new argument, *request*, must also be passed to the matching `MPI_Wait()` function. In the case of `MPI_Recv()`, the last argument, *status*, is not passed to the `MPI_Irecv()` function, but rather to the matching `MPI_Wait()` function.

Application-layer data flow effects: We use two commonly defined sets [27] to describe the side effects of each function `F` in terms of data flow. Namely:

- *DEF*: the set of all variables that must be modified by `F`.
- *USE*: the set of all variables that may be referenced by `F` without being defined by `F`, or may be referenced by `F` before being defined by `F`.

In addition to the sets shown in Table 1, the FORTRAN bindings for MPI specify one additional argument, *err*, for every MPI function (subroutine in FORTRAN parlance), except `MPI_Wtime()` and `MPI_Wtick()`. This additional argument is in the *DEF* set only of all MPI functions. MPI specifies named constants such as `MPI_ANY_TAG`, `MPI_ANY_SOURCE`, `MPI_STATUS_IGNORE`, etc., that can be passed to MPI functions as parameters. These named constants are not defined (modified) by parallel applications or MPI library functions. Therefore named constants are not involved in data dependencies and are thus excluded from the corresponding *USE* or *DEF* sets of Table 1.

Library-layer data flow effects: The data flow rules outlined in Table 1 place some limitations on compiler transformations applied to an MPI program, but are not sufficient to guarantee correctness upon transformation of MPI programs. This is true because the data flow side effects described in Table 1 do not capture side effects of the MPI functions that are internal to the library and are not visible to the application layer. Consider the following example. Each (blocking or non blocking) receive operation writes N elements into the receive buffer.[1] Thus, two consecutive receive operations that receive into different receive buffers do not seem to have any data dependence with one another (provided that the *status* and/or *request* arguments differ). Therefore, a (human or compiler) optimizer should be allowed to swap the execution order of the two receive operations according to the side effect sets. However, this is not a safe transformation and would lead to incorrect code in the general case.

Since each incoming message is uniquely identified using the sender, tag and communicator information (`src`, `tag` and `comm`), an MPI receive operation conceptually includes the array-to-array copy operation:
`buf[0:N-1] = inMesg[src][tag][comm][0:N-1]`
that copies the N elements of the incoming message into the N elements of the receive buffer. Note that `inMesg` is an artificial array introduced by this analysis, to represent the incoming messages and does **not** exist in the original program. In this abstraction `inMesg` is four dimensional to allow for messages coming from different sources, or having different tags, or communicators to be treated as independent from each other, and the FORTRAN notation `0:N-1` signifies the first N consecutive elements of an array. An MPI message can be received only once. Therefore, for this array copy analogy to be correct, the message has to be deleted from `inMesg` after it has been received. This is satisfied, if we introduce the following artificial redefinition of the incoming message array:
`inMesg[src][tag][comm][0:N-1] = artificialVar`, immediately after the aforementioned copy. Note, that these artificial assignments do not exist in the code and are not subject to compiler transformations such as code motion, copy propagation, etc. These assignments serve only as an analogy so that the safety analysis phase can limit the transformations that are considered safe to only those that lead to code semantically equivalent to the original code. A summary of the assignment perspective for all the MPI calls that we consider in this paper is provided in Table 2.

[1]The size, N, of the message is not specified by the *count* argument of the receive function, but rather the *count* argument of the matching send function on the sender's side.

Function prototype	Interprocedural data flow sets
MPI_Send(buf, count, datatype, dst, tag, comm)	DEF: ∅ USE: {all arguments}
MPI_Recv(buf, count, datatype, src, tag, comm, status)	DEF: {buf, status} USE: {count, datatype, src, tag, comm}
MPI_Isend(buf, count, datatype, dst, tag, comm, request)	DEF: {request} USE: {all arguments except request}
MPI_Irecv(buf, count, datatype, src, tag, comm, request)	DEF: {buf, request} USE: {count, datatype, src, tag, comm}

MPI_Wait(request, status)	DEF: {request, status}
	USE: {request}

Table 1: Sets of *uses* and *definitions* for key MPI functions

Function prototype	Intra-library data flow effects
MPI_Send(buf, count, datatype, dst, tag, comm)	outMesg[dst][tag][comm][0:count-1] = buf[0:count-1]
MPI_Recv(buf, count, datatype, src, tag, comm, status)	buf[0:N-1] = inMesg[src][tag][comm][0:N-1] inMesg[src][tag][comm][0:N-1] = artificialVar
MPI_Isend(buf, count, datatype, dst, tag, comm, request)	outMesg[dst][tag][comm][0:count-1] = buf[0:count-1] buf[0:count-1] += artificialVar whichbuf[request] = buf
MPI_Irecv(buf, count, datatype, src, tag, comm, request)	buf[0:N-1] = inMesg[src][tag][comm][0:N-1] + artificialVar whichbuf[request] = buf inMesg[src][tag][comm][0:N-1] = artificialVar
MPI_Wait(request, status)	whichbuf[request][0:N-1] -= artificialVar

Table 2: Array-to-array copying analogy for MPI functions

A non blocking MPI data transfer is only guaranteed to be finished after the corresponding `MPI_Wait()` has returned. Therefore, in terms of side effects, `MPI_Wait()` behaves as if it may use and define the message buffer. However, the message buffer is not passed to `MPI_Wait()` as an argument. Rather, `MPI_Wait()` can infer the message buffer through the *request* argument and library internal side effects caused by the function that set the *request* variable (i.e., `MPI_Isend()` or `MPI_Irecv()`). This behavior is captured in our analogy by the artificial array `whichbuf[]`.

As we can see in Table 2, in addition to `MPI_Irecv()`, calls to `MPI_Isend()` and `MPI_Wait()` are also equivalent to a use and a definition of the elements of the message buffer, *buf*. This is because earlier versions of the MPI standard specified that the buffer of a message that is in transit, may not even be accessed during the duration of the transfer. Starting with version 2.1 of the MPI standard the restriction on the send buffer access was removed. This enables us to relax the data dependencies of `MPI_Isend` by removing the send buffer definition from the data flow effects of `MPI_Isend()` (`buf[0:count-1]+=artificialVar`). In addition, we must specialize the data flow effect of `MPI_Wait()` such that when it waits for a send operation to complete it only uses the message buffer and does not define it. Adding these artificial uses and definitions in the data flow effects of these functions, will guarantee that the safety analysis will prevent any uses or definitions of the message buffer from being inserted between a call to `MPI_Isend()` (or `MPI_Irecv()`) and the matching call to `MPI_Wait()`.

The use of wildcards affects the safety analysis since wildcards affect the message matching. In particular, no message source can be safely considered different from `MPI_ANY_SOURCE`, and no message tag can be safely considered different from `MPI_ANY_TAG`.

Control flow related rules for code motion: For safety of code motion, there are some additional limitations, since MPI calls can have side effects beyond those captured by the data flow analysis. In particular, a call to an MPI function cannot be introduced into execution paths that do not include the original location of the call, cannot be removed from execution paths that include the original location, and cannot be introduced into any locations that would cause more than one call to the MPI function during the execution of a path that included only one call originally.

More formally, in terms of dominator [27] and post-dominator [27] control flow information, a call to an MPI function can be moved from location L_A to location L_B only [2] if: a) L_B dominates L_A and L_A post-dominates L_B, or b) L_A dominates L_B and L_B post-dominates L_A.

MPI function segmentation rules: As shown in Table 2, a call to `MPI_Send()` (or `MPI_Recv()`) with a *count* argument of N corresponds to an array-to-array copy of N elements (based on the *count* argument of `MPI_Send()` and not `MPI_Recv()`). Thus the call achieves the same data transfer as M calls to `MPI_Send()` (`MPI_Recv()`) with a *count* of $\frac{N}{M}$ (for $M \leq N$ and $N\%M = 0$). However, handshake and synchronization requirements between the sender and the receiver demand the number of calls to `MPI_Send()` to equal the number of calls to the matching `MPI_Recv()`. Hereafter, we will refer to the process of replacing a send-receive pair with M send-receive pairs as *segmentation*. Segmentation can also be applied to non-blocking send and receive operations, as long as the corresponding `MPI_Wait()` operation is also called M times such that each resulting non-blocking transfer is waited for. Furthermore, to promote further optimizations, the arguments of the segmented calls that belong to the *DEF* set of the segmented call (i.e. *request* and *status*), should be vectorized by a factor of M to avoid output dependencies between the M resulting calls.

[2]These limitations can be relaxed by copying the MPI call across N unique execution paths that are all dominated by L_A and "collectively" post-dominate L_A, or N unique execution paths that are all post-dominated by L_A and "collectively" dominate L_A.

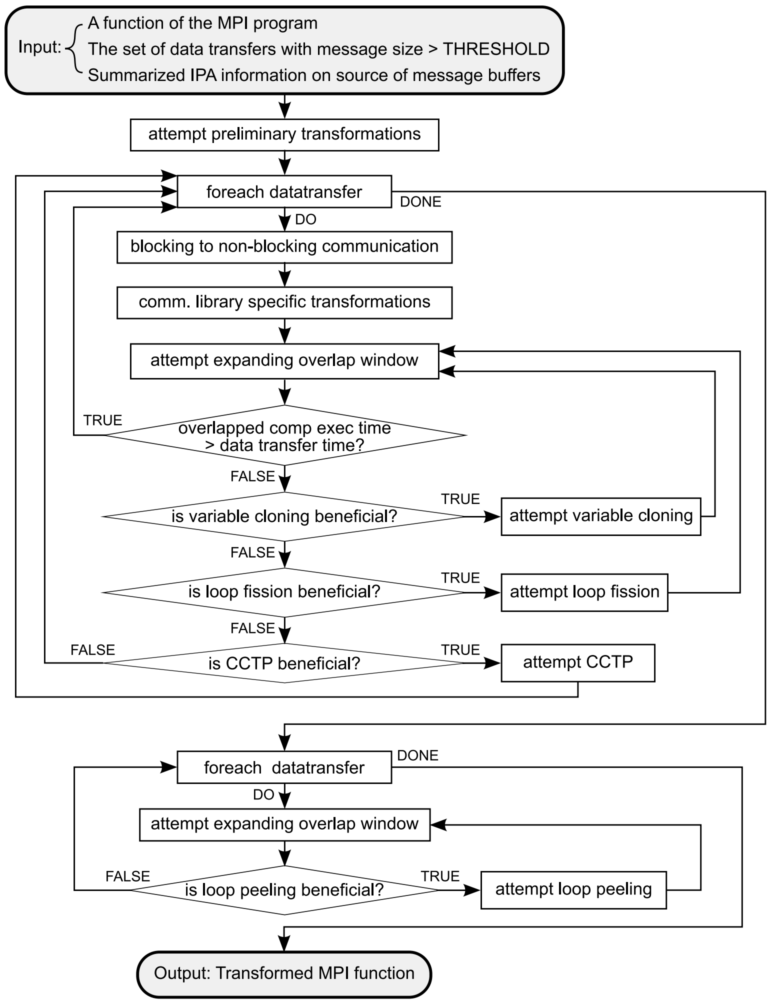
Input:
A function of the MPI program
The set of data transfers with message size > THRESHOLD
Summarized IPA information on source of message buffers
attempt preliminary transformations
foreach datatransfer
DONE
DO
blocking to non-blocking communication
comm. library specific transformations
attempt expanding overlap window
TRUE
overlapped comp exec time > data transfer time?
FALSE
is variable cloning beneficial?
TRUE
attempt variable cloning
FALSE
is loop fission beneficial?
TRUE
attempt loop fission
FALSE
FALSE
is CCTP beneficial?
TRUE
attempt CCTP
foreach datatransfer
DONE
DO
attempt expanding overlap window
FALSE
is loop peeling beneficial?
TRUE
attempt loop peeling
Output: Transformed MPI function

Figure 2: Overall Optimization Algorithm

5. OPTIMIZATION ALGORITHM

Overview

Figure 2 shows an optimization algorithm for ordering the transformations outlined in Section 2 within an MPI-aware optimizer. The optimizer will operate on one function of the source program at a time. We use the term *data transfer* to collectively designate the calls to the send, receive and potentially wait operations that need to execute for a single message to be transferred between two peers.

The main goal of the algorithm is to order the compiler transformations such that the overlap window is increased as much as possible. After some initial transformations that enable more overlap, the algorithm enters a loop that attempts to maximize the overlap window. This increase in overlap window is achieved by repeatedly applying transformations to the code that enable additional code motion by relaxing data dependencies and then attempting to expand the overlap window as much as is safe given the new data dependencies.

The algorithm is a fixpoint algorithm and will always terminate for the following reason. There are no antithetical transformations in the loops. That is, no transformation reverses the results of some other transformation, so the overlap window always expands monotonically. Since the algorithm operates on functions of finite size, the overlap window has an upper size limit; namely, the size of the function that is being processed. Therefore, the algorithm will perform a finite number of transformations, and when the overlap window can not be further extended, the algorithm will terminate.

Input

In this paper, we assume that the problem of matching send, receive and wait operations has been addressed before executing this algorithm, either by some external automatic tool, or via information provided by a human programmer. Therefore, the set of *data transfers* (i.e., matched send-recv pairs and the corresponding wait operations) is provided as input. A solution to the matching problem is complementary and orthogonal to the work presented in this paper. That is, if a tool able to match send and receive operations is developed, it can be used prior to the optimizations described in this paper regardless of the approach used by the tool to provide the matching. In the absence of a tool, a human developer can provide the matching through manual annotations of MPI send/receive calls. For this reason, as well as the complexity of providing an automatic matching of send and receive operations, we did not attempt to solve the matching problem as part of this work, but rather assumed that the matching is provided either by a human developer, or an external tool through annotations.

```
MPI_Irecv(rB[1],N,...,rReq)
do i=1,N
  sB[i] = ...
enddo
MPI_Isend(sB[1],N,...,sReq)
MPI_Wait(sReq)
MPI_Wait(rReq)
```

(a) Before CCTP

```
do T=1,N,K
  MPI_Irecv(rB[T],K,...,rReq[T])
enddo
do T=1,N,K
  do i=T,T+K-1
    sB[i] = ...
  enddo
enddo
do T=1,N,K
  MPI_Isend(sB[T],K,...,sReq[T])
enddo
do T=1,N,K
  MPI_Wait(sReq[T])
  MPI_Wait(rReq[T])
enddo
```

(b) Comp Tiling & Comm Segmentation

```
do T=1,N,K
  MPI_Irecv(rB[T],K,...,rReq[T])
  do i=T,T+K-1
    sB[i] = ...
  enddo
  MPI_Isend(sB[T],K,...,sReq[T])
  if( T > 1 )
      MPI_Wait(sReq[T-K])
      MPI_Wait(rReq[T-K])
  endif
enddo
MPI_Wait(sReq[N-K])
MPI_Wait(rReq[N-K])
```

(c) Loop fusion, alignment & peeling

Figure 3: Example effect of CCTP on MPI code

Additionally, the algorithm expects summarized interprocedural information regarding the arrays that are local in each function and the formal and actual arguments passed to each function in the program. This information is only used for the communication library related transformations and will not be further described in this paper. More information on the collection and use of interprocedural information regarding the message buffers can be found in [13].

Preliminary Transformations

Some parallel programs, that exchange data between multiple neighbor tasks, invoke the communication functions inside loops that execute just these communication functions. Despite the benefits of this coding style in reducing the source code size, it makes it more difficult for a compiler to differentiate between different data transfers and therefore optimize the code. Thus, we fully unroll all communication-only loops that have a small, statically known iteration space. However, unrolling such loops can lead to multiple copies of if-then-else branches that depend on the induction variable of the original loop. To eliminate the dead branches, we apply transformations such as constant folding and constant propagation, followed by dead code elimination. Finally, there exist parallel programs that store a value into the receive buffer, and then call the receive operation without first using the elements of the buffer that were just defined. Since the receive operation defines the receive buffer, the earlier assignment constitutes a redundant store. Redundant store elimination can eliminate the extra stores.

Communication Library Specific Transformations

Previous work, as well as the experiments in this paper, show that when possible, it is profitable to replace the MPI calls that perform a data transfer with the corresponding calls to libraries specialized for communication-computation overlap [10]. Furthermore, easy to use one-sided communication found in such libraries can simplify transformations such as tiling and pipelining (CCTP). Details on such a transformation between MPI and a specialized library are outside the scope of this paper, but can be found in [13].

Overlap Window Expansion

Each data transfer has two overlap windows, the code between the call to `MPI_Isend()` and the corresponding `MPI_Wait()`, and the code between the call to `MPI_Irecv()` and the corresponding `MPI_Wait()`. To overlap communication with computation as efficiently as possible, both overlap windows should be expanded, temporally, as much as possible. To expand an overlap window, the optimizer must first identify code that lies outside the window, and has such data flow that it can be brought safely into the window respecting all the rules discussed in Section 4. Such code will perform computation independent to the particular data transfer. Then, that "communication-independent" code can be moved into the window to increase the overlap window size.[3]

Variable Cloning

Expanding the overlap window is limited by data dependencies. However, in some cases an optimizer can remove dependencies. Some data dependencies occur when memory regions (with one or more elements) are reused to store unrelated values. As an example, consider a case where two calls to `MPI_Wait()` are passed the same *status* variable that is not used in between the calls. Data flow analysis will conclude that there is an output dependence between the two calls. However, such a dependence can be easily removed by scalar renaming [27]. In addition to such trivial cases involving scalars, array expansion [16] or array renaming [27] can be used to address cases where the receive buffer is used, or defined, in the code preceding a receive operation. We refer to all these transformations collectively as "variable cloning" since they all start with one variable, scalar or array, and result in one or more additional "clones" of that variable.

Loop Fission

When the overlap window has been expanded as much as possible through code motion and variable cloning, the optimizer can attempt to use loop nest optimizations to relax data dependencies between communication calls and computation loops that reside outside the overlap window. In particular, a computation loop can be split into two computation loops such that one has data dependencies with the communication calls and the other is independent of the communication. Traditional loop fission (also known as loop distribution [27]) can achieve this result in a safe way.

CCTP

Communication and Computation Tiling & Pipelining is a combination of several compiler transformations; namely, loop tiling, communication call segmentation, loop fusion, loop alignment and loop peeling [27]. CCTP is similar to the transformation discussed in [11], only it is applied to point-to-point data transfers rather than collective operations. It is also similar to *message strip mining* [23, 33], but CCTP is applicable to MPI programs with explicit message

[3]This approach is more general than hoisting the communication initiation calls and sinking the wait calls. A data dependence between MPI function F and a statement, S_1, can prevent F from being hoisted (or sunk). However, other statements, $\{S_j | 2 \leq j < n\}$, independent of both S_1 and F may exist beyond S_1. In this case, F cannot be hoisted (or sunk) further, but the statements $\{S_j\}$ can be moved into the overlap window of F.

passing, rather than HPF [18], or UPC [15] programs. An example of CCTP is shown in Figures 3(a) to 3(c).

The correctness of CCTP depends on the legality of fusion, since the other transformations are always safe. Indeed, loop tiling of a single computation loop, or of the outer most loop of a loop nest, is always safe (and is sometimes referred to as strip mining). Segmenting the communication calls that constitute a data transfer is also safe, so long as details, such as expansion of the variables that are defined by the calls, are handled correctly. Loop alignment, to delay the calls to `MPI_Wait()` by one iteration is trivial and always safe, as is peeling of the last iteration (of the aligned loop) that brings the last calls to `MPI_Wait()` outside the loop.

For CCTP to be profitable, the send buffer must be defined by the computation loop in such a way that after CCTP is applied, different computation tiles define equally sized regions of the send buffer. As a counter-example, a computation loop (nest) that defines the whole send buffer in the first iteration is not a good candidate for CCTP.

Loop Peeling

A computation loop (nest) that defines the whole send buffer in the first few iterations is the ideal candidate for loop peeling. Computation loops that behave as such are commonly found in codes that communicate with their neighbors in a stencil or wavefront pattern, operate on multi-dimensional arrays, and exchange only boundary data. If the send buffer is defined in the last few iterations, peeling can not help improve communication-computation overlap. However, loop reversal (if legal) can address this issue.

Transformation Ordering

The ordering of the transformations was chosen such that early transformations enable later transformations. Furthermore, variable cloning and loop fission can both relax data dependencies between computation and communication calls enabling additional code motion for expanding the overlap window, if needed. We chose to attempt peeling only after CCTP is finished, because of the interaction of the two transformations. Namely, it is easy to peel a tiled loop, but it is significantly more complicated to apply CCTP after peeling.

6. EXPERIMENTAL EVALUATION

Setup

This section summarizes our study of the performance effects of manually applying the optimization algorithm presented in Section 5 to evaluate the impacts of different transformations, using an MPI-aware approach with this optimization ordering. We extracted kernels from the NAS benchmarks [2], and the scientific application HYCOM [6]. Each kernel is an actual segment of the original program, with the communication calls inlined at the same level with the computation. We invoke each kernel from a custom driver, rather than the original application driver, so that we can execute each kernel multiple times (over 100) to amortize random noise effects. By doing so the observed noise in our measurements turned out to be insignificant and therefore it is not depicted in the graphs. All experiments were performed on an infiniband cluster with 24 Dual Core AMD Opteron 2.4GHz nodes running Linux 2.6.18. The infiniband cards have a rate of 20 Gbps (4X DDR). We used the MPI library mvapich-1.0 built on top of the infiniband layer provided by the OpenFabrics Alliance's OFED-1.3.

In the graphs seen in Figures 4 to 8, the Y axis depicts kernel execution time speedup $S = \frac{T_{original}}{T_{optimized}}$. Each bar shows the speedup of a kernel after a particular transformation has been applied to the kernel, in addition to all the previously applied transformations. Two sets of bars are depicted in each graph, shaded and white. The white bars show the version of the code that uses only MPI (when the "communication library specific transformations" phase was bypassed), and the shaded bars show the version of the code that utilizes the specialized library Gravel [10] (when this optimization phase was included). We chose to show both the white and the shaded bars, (MPI/Gravel) to demonstrate that the library specific transformations are beneficial but not a prerequisite for the following transformations.

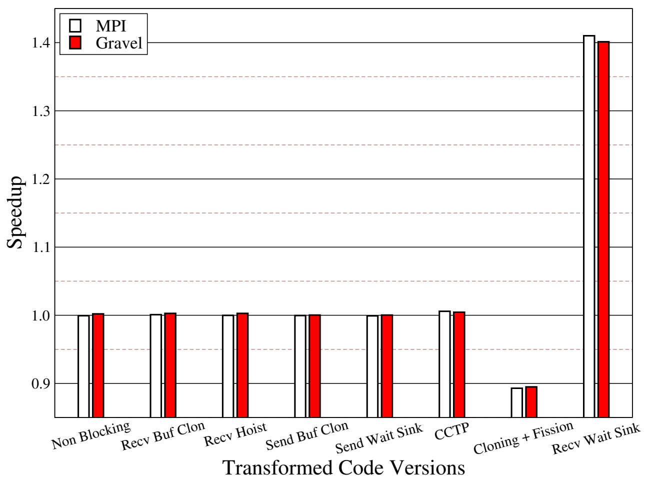

Figure 4: Kernel from NAS LU btls() (NP=16)

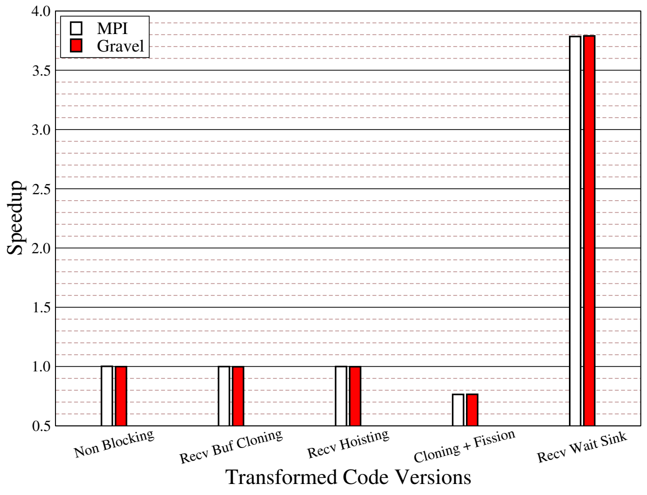

Figure 5: Kernel from NAS LU btls() precondition loop (NP=16)

Results and Analysis

The graphs in Figures 4 to 8 demonstrate that applying the proposed optimization algorithm on complex scientific codes is beneficial, since for all kernels the transformed codes exhibit speedup. The experiments also show that aggressive optimizations, such as buffer cloning, that enable further transformations, such as loop fission, can ultimately result to improved performance, even if buffer cloning itself has a negative effect on performance. Finally, the experiments show that communication library specific transformations, can lead to further performance improvements.

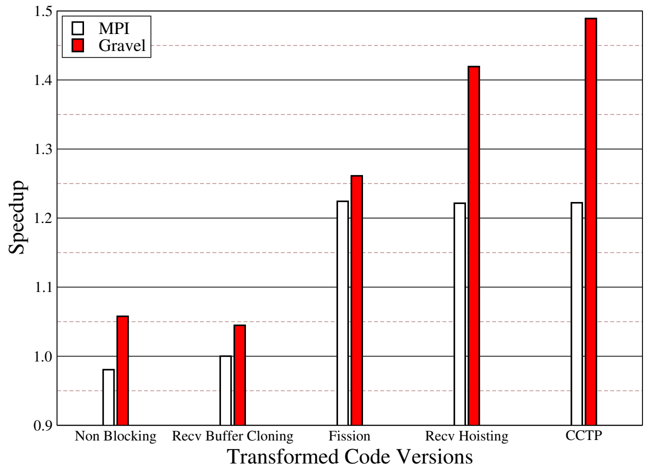

Figure 6: Kernel from HYCOM xcaget() (NP=20)

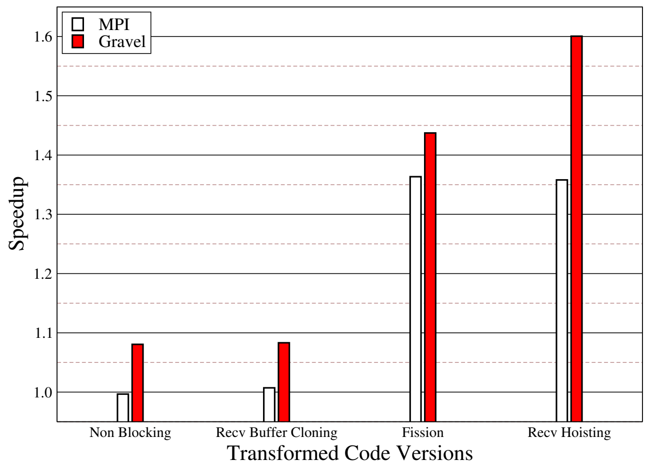

Figure 7: Kernel from HYCOM xcsum() (NP=20)

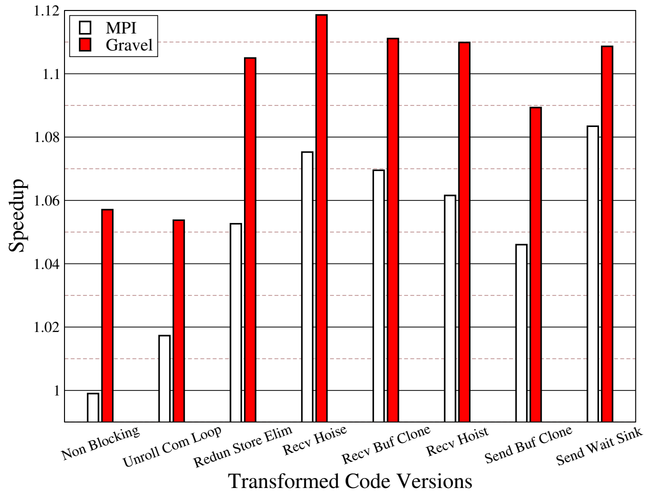

Figure 8: Kernel from NAS MG psinv() (NP=16)

An interesting result is that different kernels are affected by different optimizations and to different extents. This maps wells to the greedy design of the optimization algorithm. That is, the algorithm keeps attempting different transformations in a greedy loop that only terminates if no additional transformations can be applied.

LU blts is a wavefront kernel. That is, every MPI task waits (in a blocking receive) for the tasks on the "north" and "west". The transformations that lead to performance improvement split part of the computation (the preconditioning loop) into a communication dependent and a communication independent loop and overlap the independent computation with the communication (after converting the blocking receive into non-blocking). By doing so, part of the useful computation is performed during the time that the original application would wait for its neighbors to complete. Therefore, in addition to the communication overhead, the overhead of synchronization and filling the wavefront pipeline is reduced. In other words the optimized code executes a much more efficient pipeline and that is why the benefits can be large. To emphasize this behavior, we created the kernel shown in graph 5, where only the preconditioning loop of the blts operation is executed as computation. As shown in graph 5, having a small computation kernel emphasizes the importance of the synchronization and communication costs.

On the other hand, the kernels from xcaget() and xcsum() receive data into a temporary 1-D array and store it into a larger 2-D array and do that several times in a loop. The main benefit comes from cloning the receive buffer and distributing (fissioning) the loop in two, so that the first loop posts all the (non-blocking) receives and the second loop performs all the array-to-array copying. Hoisting the receive loop above some additional independent computation that exists in the kernels provides further benefits.

Finally, the kernel from the MG benchmark was optimized by hoisting the send and receive operations and sinking the corresponding wait operations. Interestingly, part of the optimization process, namely the second hoisting of the receive operations and the cloning of the receive buffers that enabled the hoisting, hurt performance and did not enable further optimizations that could compensate for it (the send buffer cloning and send wait sinking that follows, is orthogonal to the optimization of the receive operation). This behavior constitutes a weakness of the greedy nature of the optimization algorithm. Modeling the code can enable the optimization process to predict if a particular step would be beneficial, or not, and better guide the algorithm. In the absence of a model, the optimized code can be profiled after each optimization step so that detrimental steps are not included.

The different ways these kernels are optimized demonstrate that the optimization process described in this paper is not limited to a strict class of applications. The optimization process is designed to aggressively generate communication-computation overlap in MPI codes. Any application that experiences significant communication overhead and can be statically analyzed and transformed to achieve communication computation overlapping, can benefit from the optimization process described in this paper. Applications with significant communication independent computation, applications that generate messages in big computation loops that can be partitioned (tilled) into smaller loops independent from one another, loops that generate messages in only the first iterations of the loop, all can benefit from this optimization scheme.

Overall, our experiments demonstrate that compiler optimizations can be applied to MPI codes in a systematic way, and lead to significant reduction in execution time even when the original scientific kernels are already highly optimized.

7. RELATED WORK

Several studies have demonstrated the performance benefits of communication-computation overlap through manual transformation of codes [3, 11, 23, 31]. Achieving overlap by automatically manipulating communication has also been studied and implemented in optimizing compilers for data parallel languages such as Fortran D and High Performance Fortran (HPF) [5, 17, 20], as well as PGAS languages such as UPC [7, 8].

Regarding MPI applications, Hoefler et al. [21] have implemented a generic library function template that utilizes non-blocking collective operations. While this is not a compiler optimization, it reduces the manual development effort required to achieve overlap. CC-MPI [24] is an effort to extend MPI in order to provide more information about the communication to the application compiler, enabling it to optimize the communication. However, CC-MPI is limited to Ethernet clusters, and focuses on static, collective communications. Compiler optimization of MPI applications has been shown to have the potential to improve performance through communication-computation overlap [30]. Furthermore, the effects of focused compiler optimizations, such as code motion applied on `MPI_Wait()` operations [14], tiling and pipelining applied to `MPI_Alltoall()` [12], and library specific optimizations [13] have been demonstrated through proof-of-concept implementations.

The work presented in this paper is different from existing work, in that it is applicable to MPI programs and is neither limited to a specific type of communication pattern or function, nor a specific computing environment. Rather, this paper identifies and classifies several compiler transformations that can increase communication-computation overlap MPI programs, describes a systematic safety analysis that a compiler can use to assess the safety of these transformations, presents an algorithm that can group and order the transformations, and finally demonstrates, through experiments on complex scientific codes, the performance benefits that can result from applying these transformations.

Data flow analysis of MPI programs has been the focus of previous studies [26, 9]. However, those studies aim to extend data flow analysis to capture the SPMD semantics of MPI programs. In contrast, this paper aims to systematically describe the effects of MPI function calls in traditional data flow. Extending flow analysis to model the semantics of SPMD programs is crucial for *nonseparable* data-flow analyses such as reaching constants. However, none of the compiler transformations identified in this paper require nonseparable data-flow analysis. Rather, to perform the identified transformations a compiler must know how the different MPI function calls interact with each other and with the memory of the calling application.

8. CONCLUSIONS & FUTURE WORK

In this paper we have identified and categorized existing compiler transformations that can be applied to MPI programs in order to improve communication-computation overlap. We have also described the behavior and side effects of key point-to-point data exchange MPI functions through a set of data flow equations and rules that can be used by a compiler to automatically assess the safety of different transformations. We have described an optimization algorithm for applying the described program transformations within an optimizing compiler, and finally we have presented experimental results, from manually applying the optimization algorithm to scientific kernels, that demonstrate that the execution time of real, scientific kernels can be decreased by over 30%, on average, by using our algorithm.

We are currently working on extending the safety analysis and the algorithm to include collective communication operations, as well as non-blocking collective operations [22]. We are also studying the performance impact on whole applications. Finally, we have been implementing this algorithm within Open64 [1] and with the use of OpenAnalysis [32], so that the optimizations can be performed automatically. The current state of the implementation is promising regarding what can be implemented utilizing existing compiler infrastructure, but is outside the scope of this paper.

9. REFERENCES

[1] Open64. `http://open64.sourceforge.net`.

[2] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[3] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *20th International Parallel & Distributed Processing Symposium (IPDPS)*, 2006.

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

[5] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. A compilation approach for Fortran 90D/HPF compilers on distributed memory MIMD computers. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, pages 200–215, 1993.

[6] E. P. Chassignet, L. T. Smith, G. R. Halliwell, and R. Bleck. North Atlantic simulation with the HYbrid Coordinate Ocean Model (HYCOM): Impact of the vertical coordinate choice, reference density, and thermobaricity. *Journal of Physical Oceanography*, 32:2504–2526, 2003.

[7] W.-Y. Chen, D. Bonachea, C. Iancu, and K. Yelick. Automatic Nonblocking Communication for Partitioned Global Address Space Programs. In *ICS '07: Proceedings of the 21st annual International Conference on Supercomputing*, pages 158–167, 2007.

[8] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained upc applications. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 267–278, 2005.

[9] 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.

[10] A. Danalis, A. Brown, L. Pollock, M. Swany, and J. Cavazos. Gravel: a communication library to fast path MPI. In *EuroPVM/MPI*, Sep 2008.

[11] A. Danalis, K. Kim, L. Pollock, and M. Swany. Transformations to Parallel Codes for Communication-Computation Overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[12] A. Danalis, L. Pollock, and M. Swany. Automatic MPI application transformation with ASPhALT. In *Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL 2007), in conjunction with IPDPS 2007*, 2007.

[13] A. Danalis, L. Pollock, M. Swany, and J. Cavazos. Implementing an Open64-based Tool for Improving the

Performance of MPI Programs. In *The Open64 Workshop, in conjunction with IEEE/ACM International Symposium on Code Generation and Optimization (CGO) 2008*, Apr 2008.

[14] D. Das, M. Gupta, R. Ravindran, W. Shivani, P. Sivakeshava, and R. Uppal. Compiler-Controlled Extraction of Computation-Communication Overlap in MPI Applications. In *HIPS-POHLL joint Workshop on High-Level Parallel Programming Models and Supportive Environments and Performance Optimization for High-Level Languages and Libraries held in conjunction with the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2008)*, April 2008.

[15] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC specification v. 1.1. `http://upc.gwu.edu/documentation`, 2003.

[16] P. Feautrier. Array expansion. In *ICS '88: Proceedings of the 2nd International Conference on Supercomputing*, pages 429–441, 1988.

[17] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 71, New York, NY, USA, 1995. ACM.

[18] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. CRPC-TR92225, Rice University, Houston, TX, 1993.

[19] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Tech Report UCB/CSD-01-1163, U.C. Berkeley, November 2001.

[20] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Multiprocessor performance measurement and evaluation*, pages 57–71, 1995.

[21] T. Hoefler, P. Gottschling, and A. Lumsdaine. Leveraging non-blocking Collective Communication in high-performance Applications. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 113–115, 2008.

[22] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, 2007.

[23] C. Iancu, P. Husbands, and W. Chen. Message Strip Mining Heuristics for High Speed Networks. In *VECPAR*, 2004.

[24] A. Karwande, X. Yuan, and D. K. Lowenthal. CC-MPI: A Compiled Communication Capable MPI Prototype for Ethernet Switched Clusters. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[25] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, 2001.

[26] 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.

[27] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[28] 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.

[29] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. ACM Fortran Forum 17, 2, 1-31, 1998.

[30] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 125, New York, NY, USA, 2006. ACM Press.

[31] J. C. Sancho and D. J. Kerbyson. Improving the Performance of Multiple Conjugate Gradient Solvers by Exploiting Overlap. In *Euro-Par '08: Proceedings of the 14th international Euro-Par Conference on Parallel Processing*, pages 688–697, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] M. M. Strout, J. Mellor-Crummey, and P. D. Hovland. Representation-Independent Program Analysis. In *the Sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, September 2005.

[33] A. Wakatani and M. Wolfe. A New Approach to Array Redistribution: Strip Mining Redistribution. In *PARLE'94*, Athens, Greece, Jul 1994.