# Transformations to Parallel Codes for Communication-Computation Overlap

Anthony Danalis, Ki-Yong Kim, Lori Pollock and Martin Swany
Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716
{danalis, kimky, pollock, swany}@cis.udel.edu

## Abstract

*This paper presents program transformations directed toward improving communication-computation overlap in parallel programs that use MPI's collective operations. Our transformations target a wide variety of applications focusing on scientific codes with computation loops that exhibit limited dependence among iterations. We include guidance for developers for transforming an application code in order to exploit the communication-computation overlap available in the underlying cluster, as well as a discussion of the performance improvements achieved by our transformations. We present results from a detailed study of the effect of the problem and message size, level of communication-computation overlap, and amount of communication aggregation on runtime performance in a cluster environment based on an RDMA-enabled network. The targets of our study are two scientific codes written by domain scientists, but the applicability of our work extends far beyond the scope of these two applications.*

## 1. Introduction

Clusters of workstations are in common use among engineers and domain scientists due to their high processing power to cost ratio. The major drawback of cluster-based parallel computing as compared to using shared memory multiprocessors is the network delay induced by the node interconnecting technology of clusters. Several interconnection technologies [5, 23, 31] have been developed with the goal of improving cluster message-passing performance by providing specialized low latency, high bandwidth networks for clusters.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for pro£t or commercial advantage and that copies bear this notice and the full citation on the £rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior speci£c permission and/or a fee.
*SC|05* November 12-18, 2005, Seattle, Washington, USA
Copyright 2005 ACM 1-59593-061-2/05/0011 ...$5.00.

Specialized networks perform better than their legacy counterparts, but unfortunately the low latency and high bandwidth that they provide is more often achieved in benchmarks than in real world parallel applications [8]. Although in theory, network traffic can be handled solely by a network co-processor while the CPU is performing useful computations, low level programming details and communication specific parameters are difficult to tune in a way that is both optimal and portable – resulting in complex scientific codes exhibiting suboptimal performance.

In this paper, we present *program transformations* directed toward achieving high communication-computation overlap in parallel programs that use MPI to achieve portable parallelism in a cluster environment. The overarching thesis of this effort is that the performance of many parallel applications can be improved by transforming message passing operations based on application data dependencies and platform particularities of the specific cluster environment.

A key aspect of the program transformation consists of replacing a collective operation with a set of point to point asynchronous operations that take advantage of data dependence by sending data as soon as it becomes available. This transformation restructures the computation loop into separate blocks, or *tiles*, and places new "send" operations between the tiles to achieve overlapping of communication and computation. While the benefits of such a communication approach have been observed before [29], it is still not widely used in engineering codes due to the complexity of selecting good values for the critical parameters. The choice of parameters such as tile size, and number and size of asynchronous message transfers, depends on a number of factors. These factors vary depending on the platform and applications, thus an automatic mechanism to compare transformation constants is highly desirable.

Another component of the program transformation involves replacing the existing synchronous communication calls with one-sided ones. This step consists of replacing MPI calls with lower-level calls that avoid the MPI runtime environment and its associated overhead. We have developed a set of **Ta**rget **C**ommunication **C**ode **S**egments (TACCS), that enable us to replace a com-

monly used subset of the MPI interfaces by abstracting memory registration and one-sided communication details. The version of TACCS used in this paper utilizes Myrinet's GM system and thus fully supports one-sided communication.

The goals of this paper are twofold. First, we want to enable parallel programming practitioners to apply our transformation strategies to their own applications. We include guidance for determining the suitability of our techniques as well as for actually transforming application code to exploit the communication-computation overlap available to a particular application on a specific cluster.

The second goal of this work is to inform our ongoing research into automatic program transformation for performance. We contend that these transformations should be automated in order to mitigate lack of portability and maintainability. The results reported in this paper validate the first part of our larger hypothesis: that performance improvements can be achieved through such transformations. The tools needed for automating the process are currently being developed, and prototypes that provide proof of concept are taking shape, but are out of the scope of this paper [17].

This paper will proceed with a detailed discussion of our transformation approach as well as empirical results from an evaluation of these transformations as applied to actual scientific codes. We have conducted experiments that examine the amount of communication aggregation and overlapping achieved by varying the tile size, and the number of concurrent outstanding transfers, in order to identify the cases in which performance is maximized. We present results from a detailed case study of the effect of the problem and message size, level of communication-computation overlap, and amount of communication aggregation on runtime performance in a cluster environment based on an RDMA-enabled network.

## 2. Overview of the Problem

Many parallel scientific applications perform aggregated communication at the end of computation loops, often using collective communication primitives [33]. This approach is simple and thus results in code that is easier to maintain. Collective operations are so common and intuitive that many cluster-oriented message passing systems optimize the `MPI_ALLTOALL()` call, resulting in significant performance improvements. One way to achieve further improvement is "pre-pushing" data i.e., aggressively send data as soon as it is generated within the computation loop through the use of asynchronous I/O operations. Such a transformation will improve application runtimes in that more of the communication will be concurrent with computation, widely known to improve performance. In applications where data dependency can be resolved statically, such a transformation could be applied by an automated optimization system, thus saving programming effort and preserving the original, maintainable code.

One of the keys to improving the overlap of communication and computation is the effective utilization of technologies to remove the burden of communication from the main processor. In many message passing applications, local and remote Direct Memory Access (DMA and RDMA) are not fully utilized, despite the fact that their use can eliminate sources of overhead such as repeated copying. Although specialized versions of MPI have been developed with the goal of exploiting the features of these cluster interconnects and operating system bypass, MPI's overall design and abstract nature make it difficult to eliminate delays that could have been avoided [6, 39]. For example, one-sided communication is not supported by MPI 1.0 and although supported by MPI 2.0, significant restrictions on memory access patterns are imposed [6]. Examples of the overheads caused by MPI's design are the "receive" and "unexpected" message queues maintained by an implementation to support matching messages based on their "tag" or "sender." These queues introduce copying for the data to arrive at the actual user buffer even if MPI is implemented over a technology such as Myrinet's GM [2] that supports true zero-copy transactions. Studies demonstrate that maintaining and traversing these queues could cause impediments to scalability [8].

In this paper, we present transformations that restructure the computation loop of parallel MPI codes into smaller tiles that perform part of the computation and the corresponding communication. Asynchronous I/O is used to pipeline the execution of consecutive tiles. Additionally, we demonstrate how the use of our TACCS reduces the communication overhead even further by using compiled communications to avoid the inherent delays associated with higher level libraries.

One enabling technology for this effort is memory mapped network interfaces. Cluster interconnects like Myrinet [5], Infiniband [23] and Quadrics [31] are in wide use now and support the necessary functionality quite well. Devices that support the Virtual Interface Architecture (VIA) [15] would also be appropriate as would any other network technology that supports RDMA [1]. In memory-mapped networks, the data can be transferred between main memory and the network interface through the use of DMA. This mechanism relieves the host CPU from the responsibility for moving data from memory and eliminates the need for unnecessary copying from user buffers to kernel buffers. It has been argued that efficient communication is achieved even if the kernel plays a role in the transfer, as long as the data can be moved directly from the user buffers to the network card [27].

Our experimental results support the observation that non-blocking and asynchronous I/O are not the same [10]. Non-blocking I/O can be supported by a library regardless of the network hardware. In a programmed communication environment, the call to the network operation returns (rather than blocking), yet the computation cannot proceed because the CPU and the memory are busy performing the transfer. Truly asynchronous I/O is achieved only if the network hardware can perform the transfer on its own, letting the host processor continue with its computation. Furthermore, in the case where

the main processor is performing the transfer, cache pollution may even turn communication-computation overlapping into a non-profitable transformation [32].

## 3. Related Work

Many compiler or language-based techniques translate higher-level parallel constructs into message passing primitives as appropriate. Examples include HPF [19], Fortran-D [20], Split-C [12], and more recently UPC [16]. While these approaches allow programs to be written in a single-program, multiple-data (SPMD) style, they focus on parallel optimization in the large rather than focusing on optimization of messaging on a single host. Due to the difficulties with obtaining satisfactory performance in distributed memory environments, and the lack of ubiquitous availability of parallel languages, many applications are parallelized with explicit message passing calls. These applications are the focus of this work.

There have been several research efforts designed to mitigate messaging overheads. These efforts include Active Messages [37], U-Net [36], Fast Messages [30], and the standard produced by Microsoft, Compaq and Intel known as the Virtual Interface Architecture (VIA) [15]. While the performance improvements offered by OS-bypass and user-level networking efforts have been demonstrated, others have observed that these approaches can be difficult to use [7] and that abstractions to make easier use of such techniques can nullify some performance gains [34].

When the communication patterns of a parallel application are known at compile time, the network resources can be managed statically and significant runtime overheads can be eliminated. This approach, known as *compiled communication* [41, 42] is an optimization technique that has received significant attention.

The most related project to our study that uses compiled communication is CC-MPI [25]. The main difference with our work, is that CC-MPI is designed for Ethernet-switched clusters, whereas our study focuses on the communication strategies, program transformations, and potential performance gains for RDMA-enabled networks [5, 31, 23].

Several other projects have suggested program transformations to minimize communication overheads in parallel applications. The PARADIGM compiler [4] as well as Goumas et al. in [18] suggest optimizations that can be performed by a parallelizing compiler in order to hide network latencies. Our work is similar to these projects since we structure the computation and communication in tiles that we try to execute in a pipelined fashion. The main difference though is that these projects consider serial loops as their starting point, where we consider scientific applications already parallelized using *MPI*.

Two additional studies that consider transformations similar to ours are presented in [26] and [22]. The main difference between these studies and our work is that they effectively try to perform prefetching by *peeling*, or *strip mining* the computation loops. In our study, we consider applications that generate data using local arrays, and we try to "prepush" them to the destination before they are needed. This difference can have important implications as one-sided `get` operations are not common, while one-sided `put` operations exist in all RDMA-capable networks.

Finally, global-address space (GAS) languages [16, 40, 28] perform communication overlapping optimizations, but none of them handle input code written in *MPI*.

## 4. Transformations and Applicability

Intuitively, applications that are appropriate for our transformations are regular and lend themselves to a decomposition in which iterations can be broken down into smaller units, or tiles, that can be completed and transmitted before the entire iteration is complete. We argue that the applications in this class represent an important subset of parallel applications.

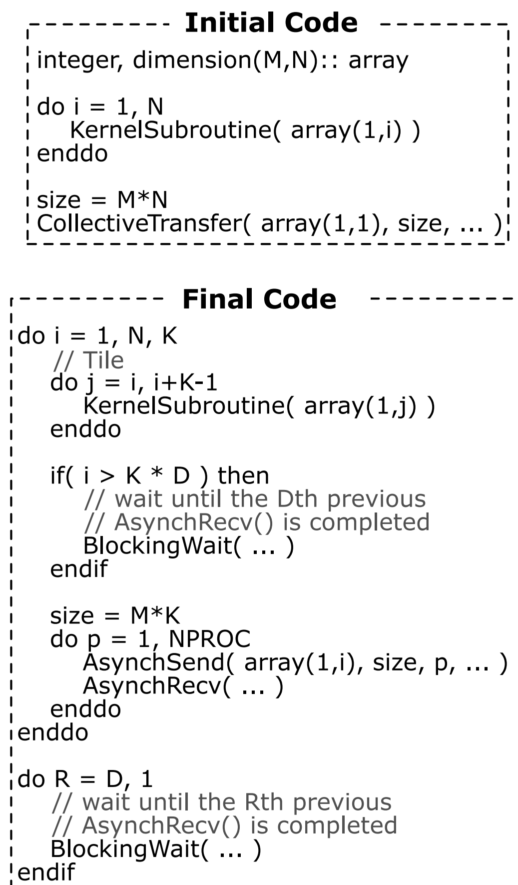


**Figure 1.** Code Transformation Overview

In Figure 1, we present an abstract form of code that represents the class of applications that can benefit from our transformations, before and after the transformation. In this form, it is clear that there are no dependencies across iterations of the computation loop in the initial code; however, there is significant communication

after the end of the loop. As can be seen in Figure 1, the transformation has the following effect on the code

- Restructures the computation into blocks, or tiles.
- Replaces all synchronous, collective communication operations by asynchronous operations.
- Issues the asynchronous calls at the end of each tile, such that the communication of a tile is overlapped with the execution of future tiles, enabling the tiles to proceed in a pipeline fashion.

The transformation process is covered in more detail in Section 4.4. Sorting [11], LU Factorization [13], Finite differences, and multi-dimensional FFT, constitute example algorithms that fit this abstract form, and can be transformed to exploit tile pipelining. In the rest of this paper, we demonstrate our transformations in a general way that can be applied to any application that fits the abstract form of Figure 1.

## 4.1 Strategies

The communication scheme shown as "Initial Code" in Figure 1 is the easiest for the programmer, and a collective communication call such as `MPI_ALLTOALL()` can perform sufficiently well in many cases. We examine several communication transformation models as alternatives that could provide performance gains. To understand how different communication strategies compare to one another, we investigate five different schemes.

1. `MPICH ALLTOALL` - The `MPI_ALLTOALL()` collective communication function is used to perform the communication after the entire computation of the inner loop, as shown in Figure 1. The implementation of *MPI* is *MPICH*, which is widely used, but not specifically designed for an RDMA-enabled network.
2. `MPICH-GM ALLTOALL` - The `MPI_ALLTOALL()` collective communication function is used in the same way as the first scheme, but the *MPI* implementation is *MPICH-GM*, which is provided by our network vendor and uses the advanced features of the hardware (e.g., RDMA).
3. `MPICH iSEND` - The computation is reorganized into independent tiles and communication is performed in every tile, as shown in Figure 2. The communication primitives are the non-blocking `MPI_iSEND()` and `MPI_iRECV()`. The *MPI* implementation used is *MPICH*.
4. `MPICH-GM iSEND` - The program is structured in the same way as scheme 3, but the *MPI* implementation is *MPICH-GM*.
5. `TACCS, one-sided I/O` - The program is structured similar to schemes 2 and 3, but the communication is handled by a set of **Ta**rget **C**ommunication **C**ode **S**egments (TACCS) built directly on top of *GM*. Abstract code representing this scheme is presented in Figure 3.

```
integer, dimension(M,N):: array

do iter = 1, MAX
  do i = 1, N, K

    do j = i, i+K-1
      /* computation kernel */
      subroutine( array(1,j) )
    enddo
    if( i > K * D ) then
      /* block for the arrival of the   */
      /* data generated D tiles ago */
      MPI_WAITALL( request( i - K * D ) )
    endif

    size = M*K
    /* network transfer */
    MPI_ISEND( array(1,i), size, ... )
    MPI_IRECV( destn(...), request(i), ... )
  enddo

  do r = D, 1
    MPI_WAITALL( request( i - r * K ) )
  enddo
enddo
```

**Figure 2.** Tiling and non-blocking I/O

```
integer, dimension(M,N):: array

do iter = 1, MAX
  do i = 1, N, K

    do j = i, i+K-1
      /* computation kernel */
      subroutine( array(1,j) )
    enddo
    if( i > K * D ) then
      /* block for the arrival of the   */
      /* data generated D tiles ago */
      poll_received_flags( i - K * D )
    endif

    size = M*K
    targMem = (i/K)%(2*D)
    set( flags( i ) )
    /* network transfer */
    put(array(1,i), size, targMem, flags, ... )
  enddo

  do r = D, 1
    poll_transfer_flags( i - r * K )
  enddo
enddo
```

**Figure 3.** Tiling and one-sided I/O

TACCS was developed atop *GM* in order to bypass the abstraction and possible hidden delays of *MPI*. It provides some minimal abstractions, because for example, a Fortran program cannot handle C pointers, but it is otherwise a set of wrappers for the GM API. Using this communication strategy, we issue one-sided RDMA send operations (i.e., `gm_directed_send()`). Each send transfers the requested data plus two more numbers at the two edges of the data buffer that are used as flags. In order for the flags to be used for transfer completion verification they are assigned a predefined value by the receiver during initialization and upon receiving each message. This value, that the receiver assigns to the flags, signifies pending transfer. At the same time prior

to transmission, each sender sets the edge elements of the send buffer to a predefined value different than the one the receiver uses. This value, that the sender assigns to the flags, signifies transfer completion. Since the message is transferred as a single buffer (at least at the application layer) when both edge elements (or flags) of the receive buffer are overwritten by the value the sender uses, transfer completion can by safely assumed.

Unfortunately this method cannot always be used. For example if a send buffer were part of a bigger buffer, it would be impossible to pad it without damaging the bigger buffer. In such cases, additional control messages can be used to verify transfer completion, as long as the network guarantees message order preservation, a feature that virtually every cluster interconnection network supports. The latter method is safer and easier to implement, but might impede scalability as a large number of control messages can create contention in the network.

### 4.2 Tiling Parameters

The specifics of the transformation vary across applications and cluster platforms. The behavior of the communication in cases three to five is controlled by two parameters, $K$ and $D$. As can be seen in Figures 1, 2 and 3, $K$ controls the size of each tile and $D$ specifies the delay (in tiles) after which the application will block waiting for a network operation to complete. In other words, $D$ determines the size of a non acked window.

Determining the best value for these parameters is not a trivial task as there are several trade-offs and details associated with them. In particular, increasing $K$ leads to a larger tile size which corresponds to fewer and larger messages. This is a desirable effect since the fixed overhead of a single transfer operation is amortized over larger transfers. At the same time, higher average bandwidth is achieved, as larger messages experience higher transfer rates [35]. On the other hand, as $K$ increases, the size of the last tile increases and that communication cannot be overlapped with useful computation.

As can be seen in Figure 2, in the extreme case where $K$ is equal to a complete iteration (i.e., `N`), the code is equivalent to the non-overlapping initial code shown in Figure 1 (assuming $D = 1$). Thus, one could expect that the optimum value for $K$ would be in a range in which the messages are relatively large, and the last non-overlapped tile is relatively small.

$D$ controls the pipelining of the tiles. Using values of $D$ higher than one, is equivalent to using a polling function (such as `MPI_Test()`) to avoid, or postpone, blocking for the arrival of the data. Higher values of $D$ increase the size of the pipeline of panels in flight and allow for communication-communication overlapping in addition to communication-computation overlapping, since more than one message is transferred at the same time. On the other hand, increasing $D$ leads to more outstanding messages. This imposes a demand for more duplicate buffers (as send or receiver buffer cannot be reused until the corresponding message has been waited for) and higher use of network resources and pinned memory. Since the need for resources increases with $D$, and the potential for communication-communication overlapping in RDMA-enabled networks is not high, one could expect that small values of $D$ (i.e., 1 to 3) would yield the best results.

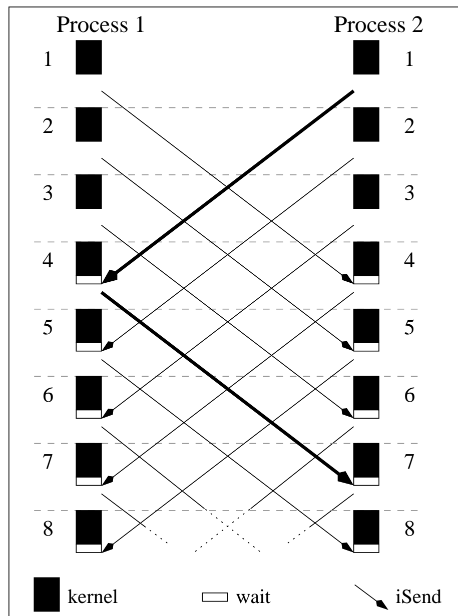


**Figure 4.** Synchronization when $D = 3$

### 4.3 Preserving Semantics

For correctness to be preserved when one-sided asynchronous communication is used, the application needs to keep $D + 1$ send buffers and $2 \times D$ receive buffers. For example, in Figure 4 (where $D$ is chosen to be 3), the message sent in iteration $I_1$ of *Process 2* is waited for in iteration $I_4$ of *Process 1*. *Process 2* though is not informed that this happened until it reaches its own iteration $I_7$. Therefore, iterations $I_1$ through $I_6$ (i.e., $2 \times D$) of *Process 2*, have to submit to different receive buffers. Regarding the send buffers, Figure 4 shows that each process waits for the arrival of the first block of data after iteration $I_4$. This blocking operation also waits for the completion of the `gm_directed_send()` that was initiated $D$ iterations earlier. Therefore, when the kernel of iteration $I_5$ needs to save its output into a send buffer, it can reuse the buffer of iteration $I_1$. Consequently, only four (i.e., $D + 1$) distinct send buffers are needed.

Note that the total amount of memory that is needed for send and receive buffers is equal to the data generated by $(3 \times D + 1) \times K$ iterations of the loop we are tiling. This should be much smaller than the memory needed by the original application that needs to execute (and therefore save) all the $N$ iterations before transferring any data over the network, as the optimal $K$ is expected to be much smaller that $N$ (see Section 4.2).

At the same time the need for allocating these buffers does not impose a scalability issue as the total size does not depend on the number of nodes used. Indeed, when the number of nodes is higher, more, but smaller messages need to be sent, for the results of the computation to be exchanged.

### 4.4 Transformation Process

In order for a programmer or an automated optimizing system to apply the proposed program transformations, the actions described below and demonstrated in Figure 5 need to be performed.

1. Identify the function calls that perform synchronous collective data transfer (e.g., `MPI_ALLTOALL`).
2. Select the calls `SC` that send data previously computed (or in any way altered) by a heavy computation loop, `L`.
3. From the set `SC` in step 2, select the calls `C` where parts of the data are ready before the end of the entire computation loop `L`.
4. Tile the loop `L` by restructuring the computation into a double nested loop structure `LN` semantically equivalent to the original loop `L`.
5. Modify each communication call `c` in `C` so that `c` sends only the data generated by one tile and place `c` inside the outer loop of `LN`, after the execution of the tile (inner loop).
6. For each collective send call `c`, replace `c` by a loop of asynchronous calls `c'`.
7. For each `c'`, insert a blocking call to wait for the data arrival, such that it is separated from the send in `c'` by one or more tile executions. The number of tiles separating each send-wait pair defines the number of outstanding concurrent transfers, or in other words the depth of the tile pipeline.
8. Insert code to wait for the arrival of the last blocks of data after the end of `LN`.

The first step of the transformation can be generalized such that any kind of data exchange is taken into account. Unfortunately, matching individual send and receive operations statically is undecidable in the general case. Nevertheless, if the transformation is performed by a human programmer, or a semi-automatic system that is guided by a human programmer, the acceptable codes can be extended far beyond those that limit themselves to collective communication operations.

For simplicity reasons, our example figures were made with the assumption that $N\%K == 0$. When $N\%K \, != 0$, i.e., the number of iterations of the original loop is not divisible by the selected value for the tile size, additional code is generated to perform the transfer of the "remaining" data and the corresponding blocking wait. This is necessary in order to ensure that the transformed code will be semantically and algorithmically equivalent to the original code.

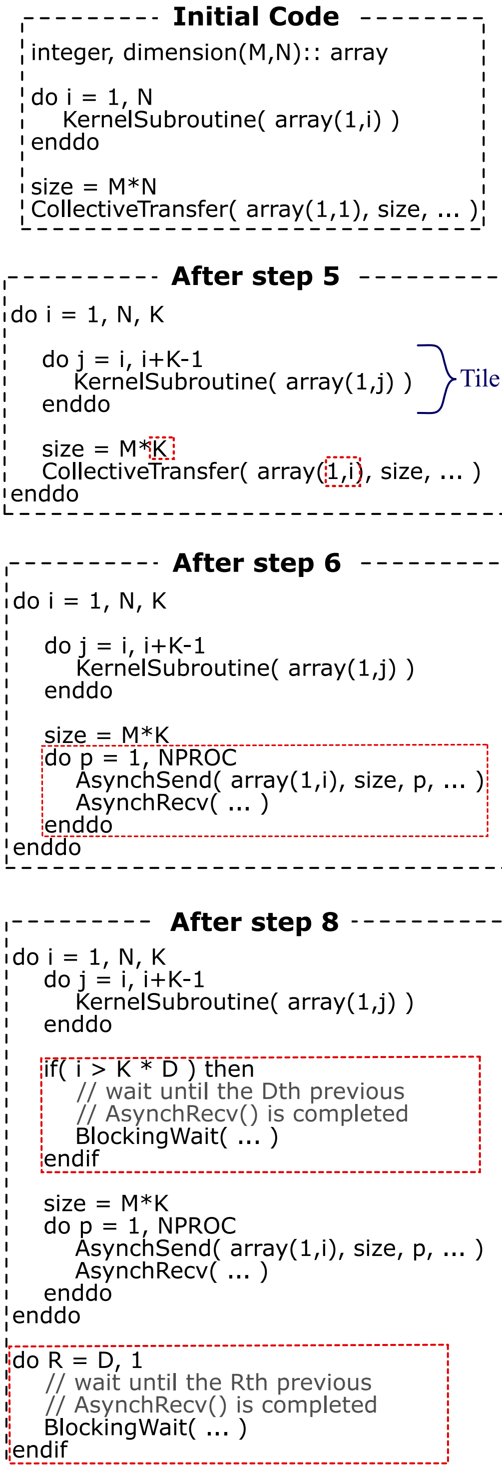


**Figure 5.** Transformation Steps

## 5. Experimental Study

We designed and conducted experiments to evaluate the described communication strategies, and to re-

veal relationships across several dimensions regarding the communication overhead of parallel programs under the different schemes. The particular questions targeted by this study are as follows.

1. Is there a significant difference in performance between *MPICH* and *MPICH-GM* when using a *Myrinet* cluster?

2. What performance gain can be attained when non-blocking I/O is overlapped with the computation compared to the case where communication is not overlapped with the computation?

3. How does the specialization of the *MPI* library to a particular interconnecting network affect the answer to the previous question?

4. What is the performance gain that can be attained if one-sided communication operations are used?

5. How sensitive is the overhead of each communication strategy to the number of processors?

6. How sensitive is the overhead of each communication strategy to the message size?

### 5.1 Experimental Methodology

Following the steps described in Section 4.4, we created versions of two scientific applications to exhibit the communication strategies presented in Section 4.1. We executed experiments varying several parameters, in order to compare their relative significance. In particular, for every implementation, we varied the number of processors ($NP$), the tile size ($K$), the depth of the tile execution "pipeline" ($D$), and the problem size.

The dependent variable that we measured is the execution time. While performing our experiments, the cluster was not busy running any major parallel application. Smaller programs were running on some of the nodes and using the network connections. For this reason, in order to minimize the noise in our results, for every value of the parameters we wanted to examine, we took more than thirty measurements. To compare the different communication strategies we use the minimum execution time achieved by each strategy, as the random error in execution time can only be positive and therefore the minimum value should be the most accurate.

The cluster that we used for the experiments consists of 20 Sun Microsystems Ultra Enterprise 450 machines, each with four 250MHz Ultra-II processors. Since the focus of our research is on the network overhead, we only used one processor per node, to force all communication to be performed over the network. Eight nodes of the cluster have 1GB of memory 4-way interleaved. The other 12 nodes have 512MB of memory 4-way interleaved. Each of the 20 nodes has a 4GB system disk. The interconnecting network is Myrinet. The Myrinet cards are model M2M-PCI32b (LANAI 0x0403, 32-bit) and each card has 512K SRAM. Despite the absolute performance of this hardware, we assert that the relationships between processor and I/O speeds are such that our approach remains applicable for faster processors and interconnection networks.

The programs we used for our study include an application given to us by a group of scientists [14] in the Physics Department and an application given to us by a group of engineers [21] in the Chemical Engineering Department, both groups at the University of Delaware. The first application, which we call *magnet*, is used for investigation of magneto-hydrodynamic turbulence through spectral methods, and the second application, which we call *visco*, simulates viscoelastic turbulent flow in a channel. Despite the differences in the structure, functionality, and programming styles of *magnet* and *visco*, our transformations were performed by the same process. We only manipulate the communication and the computation loop that "generates" the data to be sent. In both cases, the code that we transformed contained loops executing one or two dimensional *FFT* using the *FFTW* library. Also, in both cases, all the code that we altered was located within one function, or subroutine. Consequently, we only timed the subroutine we changed, in order to study the impact of our transformations regardless of the percentage of the time each application spent in the modified code.

Because we applied the same transformation process to both applications despite major differences between the two applications, we believe that our ideas are fairly general and can be used in a wide range of parallel applications. In addition, although not presented in this paper, we have been working with additional complex scientific applications, including an *N-body simulation* and an application that performs *Constant (N)umber (V)olume (T)emperature Molecular Dynamics Simulations of Liquid Water* considering all the $O(N^2)$ particle interactions.

The applications were compiled with *mpich-1.2.5*, *gm-mpich-1.2.6..13*, or *gm-1.2.3* according to the needs of each communication scheme.

## 6. Results and Analysis

Figures 6 and 7 present the normalized execution times of the studied communication strategies for *magnet* and *visco*, respectively. The normalization was performed by dividing all execution times by the pure computation execution time. To measure this base time, we created a version of each application where the communication part was commented out. Clearly, such a version does not achieve mathematically correct values, since no data is being exchanged among the processors and thus the computations are performed on arbitrary values. Nevertheless, since the computation time is not affected by the values of the data the computation is performed on, this version of the code is perfectly legitimate for performance comparisons. In addition, it constitutes a theoretical optimum code version (given that we do not alter the actual computation), since it exhibits zero communication delay.

Since our graphs are normalized to the execution time, the actual values of the `Y` axis represent the slowdown of the application, due to communication over-

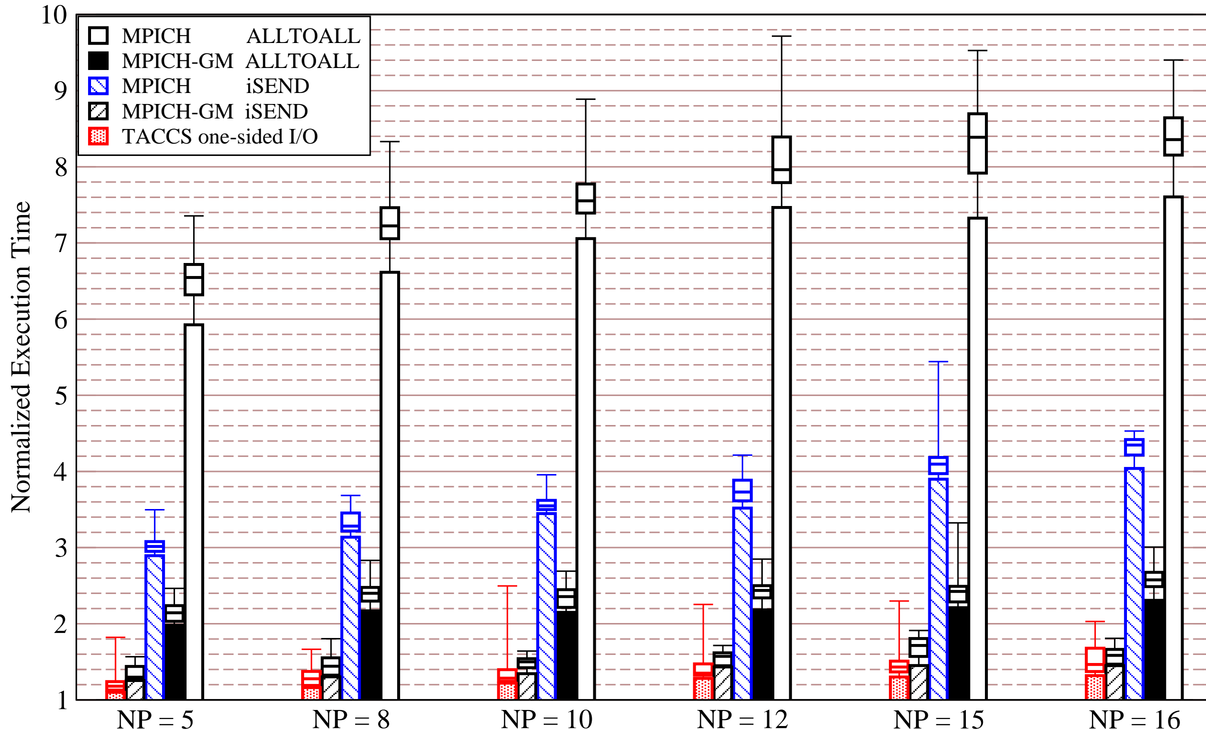


**Figure 6.** *Visco*, Problem size: 480x480x48 complex numbers ≈ 168MBytes

head. In all bar figures, the bars represent the overall best case for every strategy. This means that in order to compare the different communication strategies, we searched the whole parameter space to find the optimal set of parameters for each strategy, and we use these optimal parameters for each strategy in order to generate the data that represents it. The execution time of the timed subroutines was in the order of seconds, so we added boxes and whiskers in Figure 6 to demonstrate that our experiments are not significantly affected by noise. Since in most cases the boxes (25%, median, 75%) of the different strategies do not even overlap, our results can be safely considered statistically consistent.

The first assessment that can be made by looking at our results is that the performance of programs using *MPICH-GM* is significantly better than the corresponding ones using *MPICH*. This is an intuitive result, since *MPICH-GM* is tuned for the specific network infrastructure; however, the magnitude of the library's positive impact was not expected. Nevertheless, the difference between these two libraries is not related to our work, so we will not analyze it further.

A second observation, which is intuitive as well, is that moving from a communication model where all the I/O is performed after the entire computation to one where non-blocking I/O is used to overlap communication and computation, yields significant performance improvements. In addition, this is a clear trend in our graphs for both implementations of *MPI*. Although this is an expected result described in theory even in textbooks [29], it is very important that such a performance improvement can be witnessed in real application kernels and is achievable through the methodical transformation described in Section 4.4.

In answering the fourth question about the potential performance gain of one-sided communication, the one-sided operations seem to provide an advantage over all other strategies. This advantage is magnified when small message buffers need to be used. Figure 8 presents the normalized execution times for *visco* for a large range of K values, keeping $D = 1$. Figure 8 shows that the *MPI* strategies achieved their best case for large values of $K$, and they performed poorly for very small values of $K$ (more than twice the overhead of the custom library for values of $K$ up to 5). This means that in order for the *MPI* to perform the best, it needs to send few, large messages. On the other hand, the code using our custom library performed almost equally well for every value of $K$. The implications of this observation are important. In applications similar to *magnet* and *visco*, where tiling can be performed and the size of the tile (and therefore $K$) can be selected by the programmer (or the compiler), *MPI* can perform almost equally well with GM. However, in applications where frequent exchange of small messages is inevitable (because of data dependencies), the smaller overheads of GM can lead to substantial performance improvements over higher level libraries.

The relative performance of the different strategies was consistent across different numbers of processors. Some variations in the relative performance exist, but there is no conclusive observation. We plan to experiment with clusters that feature higher number of CPUs in order to investigate whether there exist trends that

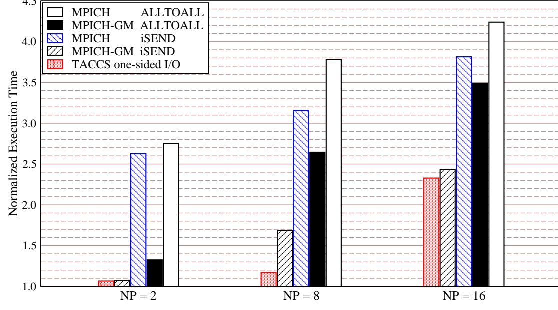


**Figure 7.** *Magnet*, Problem size: 1024x1024x4 complex numbers = 64MBytes

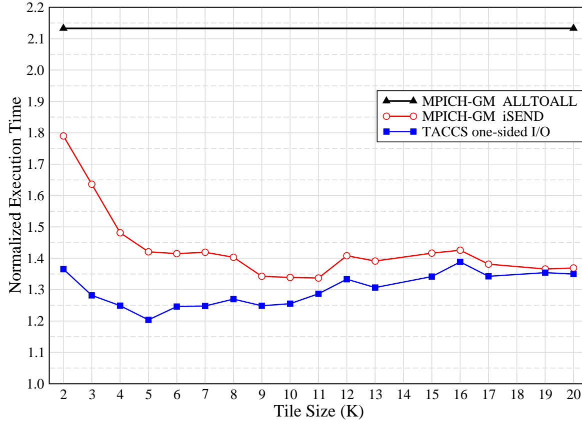


**Figure 8.** Effect of tile size ($K$) on performance of *visco*

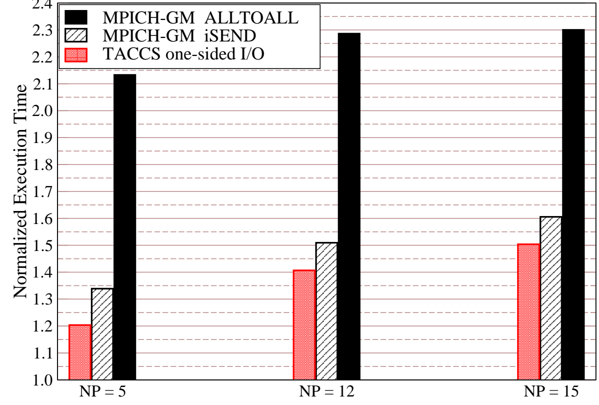


**Figure 9.** *Visco*, Problem size: 240x480x48 complex numbers ≈ 84MBytes

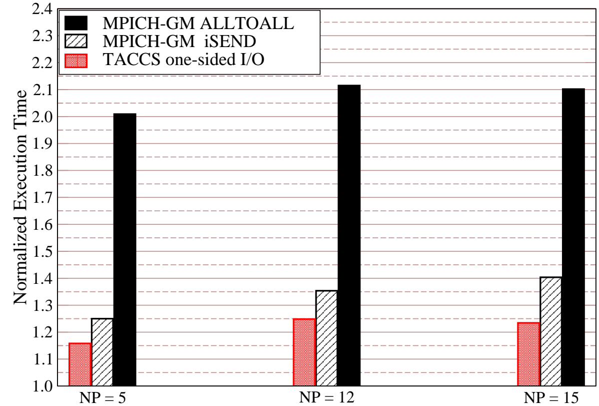


**Figure 10.** *Visco*, Problem size: 960x480x48 complex numbers ≈ 337MBytes

only become clear for large number of processors.

The relationships between the different strategies were consistent across the two applications as well as they were consistent across different problem sizes. Figures 9 and 10 exhibit the consistency across problem size, by showing very similar behavior in two configurations of *visco* where the problem size differed by a factor of 4.0. Actually, the graphs show that the *normalized* communication overhead drops as the problem size increases, which was expected since the communication load is $O(n)$, but the computation load (of FFT) is $O(n \cdot log(n))$.

In Figure 7, where a small problem size is represented, we can see that for large number of processors the overhead of all communication strategies grows significantly. An explanation for this behavior is as follows. The strategy that uses the *MPICH-GM* implementation of `MPI_ALLTOALL()` (solid black bar) takes $T_{cp}$ time for the computation, $T_{cm}$ time to transfer the actual data, and $\delta_1$ time for extra overheads related to registering memory, synchronization, cache pollution and the library's internal bookkeeping. The strategy that uses our one-sided asynchronous I/O provided by our thin library, (shaded bar) takes $max\{T_{cp}, T_{cm}\}$ time for the computation and the data transfer (because of the overlapping) and $\delta_2$ time for extra overheads including the slowdown of the computation and communication due to contention caused by the overlapping. The difference between the two normalized bars would be

$$\Delta = \frac{T_{cp} + T_{cm} + \delta_1}{T_{cp}} - \frac{max\{T_{cp}, T_{cm}\} + \delta_2}{T_{cp}}$$

If we assume that $\delta_1$ is relatively small, then the reason that the `MPI_ALLTOALL()` case (solid bar) exceeds 2.0 is because $T_{cm} > T_{cp}$. This would give:

$$\Delta = \frac{T_{cp} + T_{cm} + \delta_1}{T_{cp}} - \frac{T_{cm} + \delta_2}{T_{cp}} = 1 + \frac{\delta_1 - \delta_2}{T_{cp}}$$

Therefore, if $\delta_1$ and $\delta_2$ are small relative to $T_{cp}$ (and close to one another), the distance between the original strategy (solid bar) and the strategy with the best overlapping can not be more than 1.0 (in a normalized graph). In other words, by overlapping communication and computation, we can not hide more network latency than the computation time. A very interesting observation is that in all our graphs the difference between the performance of the original code and our best strategy is very close to one. This means that we have achieved

near maximum overlapping. Even more, the fact that in some cases the difference is higher than 1.0, suggests that the "extra" overhead associated with our custom library ($\delta_2$) is smaller than that of *MPICH-GM* ($\delta_1$).

As mentioned previously, the parameter $D$ controls the depth of the pipeline, i.e., the number of outstanding transfers. We observed that the best value of $D$ was always a small number between one and three, inclusive. Particularly, *MPI* implementations seemed to gain some performance improvements by using a value higher than one, but the TACCS scheme was not affected noticeably. This was expected as the low level TACCS code performs just an RDMA transfer without any secondary bookkeeping operations, so there cannot be overlapping between more than one such operations. On the other hand, MPI performs more operations than data transfer, such as send and receive queue management, so overlapping is feasible and provides further overhead reduction. In addition, our GM hardware sets a limit less then 30 on the number of pending requests through the available send tokens. This limit imposes a small upper limit on $D$ for large number of CPUs when the TACCS scheme is used. For large values of $D$, the performance degraded considerably under every strategy, since the resource usage becomes excessive.

All our experiments were performed on the same cluster. This could be considered a threat to validity as cluster-sensitive parameters can be expected to affect the performance of parallel applications. We argue that the trends and relationships identified in this paper extend beyond the limits of a single experimental environment. First, we only mention relative behaviors, which should not be altered if a parameter is altered. In addition, most existing clusters have two CPUs per node. In our case, there were four CPUs per node, providing additional computing power for the non-optimized schemes that make significant use of the host processor, since we only used one CPU per node for the application. It is our belief that in a cluster with dual SMP nodes, our observations will still hold, and possibly be magnified.

All the reported performance numbers refer to the execution time of the code that we optimized and not the overall application. These timings were taken for two reasons. First, we wanted to emphasize the impact of our transformation. Total application times would skew our numbers depending on the percentage of time each application spends inside the optimized code. Second, the codes we optimized were scientific applications that take several weeks to months to execute. Given our broad sampling of the parameter space (Number of Processors, Problem Size, $K$ and $D$), it was impractical to experiment with the whole application code.

# 7. Conclusions and Future Work

In this paper, we presented a set of transformations that enable parallel applications written using *MPI* to achieve communication-computation overlapping and thus, decrease their total execution time. We describe the transformations in an abstract but explicit manner. We provide steps to be followed to achieve the transformations in order to guide parallel programming practitioners, or developers of automatic optimization systems, through the optimization process.

To study the impact of our transformations, we experimented with two complex scientific applications. Our experiments indicate that the proposed transformations can yield significant performance improvements depending on the original design of an application. Particularly, we showed that when using an RDMA-enabled network infrastructure, the performance gap between an optimized *(MPICH-GM)* and a non-optimized *(MPICH)* version of *MPI* is significant. In addition, migrating from a collective communication strategy (`MPI_ALLTOALL()`) to a non-blocking strategy (`MPI_iSEND()`) that allows for overlapping, might decrease maintainability of the code, but can provide a considerable performance improvement. Finally, we showed that a low level, hardware-dependent API *(GM)*, that does not introduce abstraction and hidden delays, can improve performance even further. Unfortunately, replacing *MPI* with a hardware-dependent library would severely damage portability. Nevertheless, in cases where the message size is small, the performance improvements are so significant that it might be a viable option. Furthermore, if the transformations were performed by an automated source-to-source optimizing system, both high performance and portability could be achieved.

In the future we plan to examine the effects on performance of various cluster characteristics and to predict the best values for parameters such as tile size ($K$) and pipeline depth ($D$). We will approach this prediction by either using an analytical model in the spirit of [3, 24, 9] or a system similar to ATLAS [38] that can empirically infer the optimal values. In addition, we are working on developing a source-to-source optimizing system with ability to automatically identify appropriate opportunities and apply our transformations to parallel applications.

# 8. References


[1] Ammasso. `http://www.ammasso.com/`.

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

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[4] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *First International Workshop on Parallel Processing*, 1994.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[6] D. Bonachea and J. Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. In *2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing (SHPSEC-03)*, 2003.

[7] L. Bouge, J. Mehaut, R. Namyst, and L. Prylli. Using VIA to build distributed, multithreaded runtime systems: a case study. Research Report 1999-27, CNRS-INRIA-ENS LYON, 1999.

[8] R. Brightwell and K. D. Underwood. An analysis of the impact of MPI overlap and independent progress. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 298–305. ACM Press, 2004.

[9] K. W. Cameron and R. Ge. Predicting and Evaluating Distributed Communication Performance. In *Supercomputing*, Pittsburgh, PA, 2004.

[10] F. Chaussumier, F. Desprez, and L. Prylli. Asynchronous Communications in MPI - the BIP/Myrinet Approach. In *Euro PVM/MPI '99*, 1999.

[11] M. J. Clement and M. J. Quinn. Overlapping Computations, Communications and I/O in parallel Sorting. *Journal of Parallel and Distributed Computing*, 28(2):162–172, 1995.

[12] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel programming in Split-C. In *Supercomputing*, pages 262–273, 1993.

[13] F. Desprez, J. Dongarra, and B. Tourancheau. Performance study of LU factorization with low communication overhead on multiprocessors. *Parallel Processing Letters*, 5:157–169, 1995.

[14] P. Dmitruk, L.-P. Wang, W. H. Matthaeus, R. Zhang, and D. Seckel. Scalable parallel FFT for spectral simulations on a Beowulf cluster. *Parallel Computing*, 27:1921–1936, 2001.

[15] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. IEEE Micro, pages 66–76, March/April, 1998.

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

[17] L. Fishgold, A. Danalis, L. Pollock, and M. Swany. An Automated Approach to Improving Communication-Computation Overlap in Clusters. In *Parallel Computing 2005*, 2005.

[18] G. Goumas, A. Sotiropoulos, and N. Koziris. Minimizing completion time for loops tiling with computation and communication overlapping. In *15th International Parallel and Distributed Processing Symposium*, 2001.

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

[20] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Supercomputing*, pages 86–100, 1991.

[21] K. Housiadas and A. Beris. An efficient fully implicit spectral scheme for DNS of turbulent viscoelastic channel flow. *Non-Newtonian Fluid Mechanics*, 2004.

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

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

[24] F. Ino, N. Fujimoto, and K. Hagihara. Loggps: a parallel computational model for synchronization analysis. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 133–142, 2001.

[25] 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.

[26] G. Liu and T. Abdelrahman. Computation-Communication Overlap on Network-of-Workstation Multiprocessors. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998.

[27] K. Magoutis, M. I. Seltzer, and E. Gabber. The Case Against User-level Networking. In *Third Workshop on Novel Uses of System Area Networks (SAN-3) (Held in conjunction with HPCA-10)*, 2004.

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

[29] P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997.

[30] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995.

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

[32] M. Prieto, I. M. Llorente, and F. Tirado. Data locality exploitation in the decomposition of regular domain problems. *IEEE Trans. Parallel Distrib. Syst.*, 11(11):1141–1150, 2000.

[33] R. Rabenseifner. Automatic MPI Counter Profiling of All Users: First Results on a CRAY T3E 900-512. In *Message Passing Interface Developer's and User's Conference 1999 (MPIDC'99)*, pages 77–85, 1999.

[34] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local-area communication with fast sockets. In *Proceedings of Usenix Annual*

*Technical Conference*, pages 257–274, 1997.

[35] D. Turner and X. Chen. Protocol-Dependent Message-Passing Performance on Linux Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2002)*, Chicago, Illinois, 2002.

[36] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 40–53, Dec 1995.

[37] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architecture*, 1992.

[38] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Supercomputing 1998: High Performance Networking and Computing*, 1998.

[39] J. B. White III and S. Bova. Where's the Overlap? An Analysis of Popular MPI Implementations. In *MPI Developer's and User's Conference (MPIDC '99)*, 1999.

[40] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect, 1998.

[41] X. Yuan, R. Melhem, and R. Gupta. Compiled Communication for All-Optical TDM Networks. In *Supercomputing'96*, 1996.

[42] X. Yuan, R. Melhem, and R. Gupta. Algorithms for Supporting Compiled Communication. *IEEE Transactions on Parallel and Distributed Systems*, 14(2):107–118, 2003.