AN AUTOMATED APPROACH TO IMPROVING COMMUNICATION-COMPUTATION OVERLAP IN CLUSTERS

by

Lewis Fishgold

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Honors Bachelor of Science in Computer Science with Distinction

Spring 2005

AN AUTOMATED APPROACH TO IMPROVING COMMUNICATION-COMPUTATION OVERLAP IN CLUSTERS

by

Lewis Fishgold

Approved: _____
Lori Pollock, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____
Martin Swany, Ph.D.
Committee member from the Department of Computer & Information Sciences

Approved: _____
James Glancey, Ph.D.
Committee member from the Board of Senior Thesis Readers

Approved: _____
John Courtright, Ph.D.
Director, University Honors Program

ACKNOWLEDGEMENTS

Most of all, I thank my family for the love and support I have been lucky to receive my entire life. I also thank my fellow researcher, Anthony Danalis, for the opportunity to work with him and the help he provided, the rest of the HiperSpace lab, and the members of my thesis committee, Dr. Martin Swany and Dr. James Glancey. Finally, I thank Dr. Lori Pollock for all her work over the past two years in being a conscientious and encouraging advisor.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Parallel clusters have become common platforms for programmers to achieve desired runtime performance for applications with high processing demands. Unfortunately, scaling these applications to larger numbers of CPUs for even higher performance gains often fails because the communication overhead increases at a similar rate. The compiler optimization research group at the University of Delaware has developed a transformation that can significantly reduce the communication overhead of a class of parallel MPI programs by restructuring a program towards maximizing communication-computation overlap. This thesis describes research targeted to developing a source-to-source optimizer that automates this transformation by automatically identifying safe transformations and performing the necessary restructuring to pre-push data during computation to exploit the underlying capabilities of an RDMA-enabled network. The optimizer specifically targets the large community of domain scientists that use MPI to implement their parallel algorithms to be executed on RDMA-enabled network clusters.

Chapter 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 [6, 17, 24] have been developed with the goal of improving cluster message-passing performance by providing specialized low latency, high bandwidth networks for clusters. Such technology can theoretically reduce communication latency by overlapping communication with computation by handling network traffic solely on a network co-processor, freeing the CPU to perform useful computations.

Unfortunately, most existing scientific codes cannot take advantage of this capability because they do not utilize the opportunities latent in, but unexploited by their structure to overlap communication and computation. Most scientific applications are structured with iterations consisting of a computation loop followed by collective communication operations to exchange data for the next iteration. Although this structure is easy to code and exhibits modularity and high maintainability, it prevents communication-improving network technology from being fully utilized.

To overcome the restrictions imposed by such overlap-naïve code, the code can be transformed so as to aggressively send data as soon as it is generated within

the computation loop using non-blocking, asynchronous I/O operations. To accomplish this, the computation loop is partitioned into blocks, or tiles, that execute only part of the maximum number of iterations. At the end of each tile, a subregion of the whole array is generated, and depending on data dependencies between iterations, a subset of that data will not be altered in later iterations. In the transformed version of the code, an asynchronous send operation is initiated at the end of each tile to transfer the data that is finalized. This optimization could be described as "pre-pushing", as it is similar to prefetching, only in the reverse direction.

Danalis et al. [9] have evaluated the potentially achieved performance gains of this pre-push transformation, by identifying applications that could potentially benefit from the transformation, and performing the transformation manually. From the results of the study, it is clear that after applying the transformation, near maximum communication-computation overlap is achieved, significantly reducing communication overhead in comparison to the original code.

Although the pre-push transformation can be performed by an experienced programmer, there are several reasons to build an automated system.

1. Asynchronous communication can be difficult to program, particularly when many processors are involved and each one has several outstanding messages at all times.

2. The optimal value for the tile size has to be recomputed (or rediscovered through extensive profiling) every time the code changes, or the cluster CPUs or network changes.

3. The suggested transformations have a negative impact on the maintainability of the code and in the case where low level communication primitives (eg., Myrinet's GM) are used, portability is also affected.

4. Having an automated system perform the transformation opens the optimization to a wider audience of applications such as legacy codes, and those whose programmers are unaware of the optimization.

Using an automated system, programmers can develop and maintain the simple, original code, while the cluster executes the complex but efficient form.

Given the motivation for automating the transformation process, this thesis presents original research on a source-to-source transformation system which can automatically optimize an application by analyzing its source code, identifying opportunities for communication-computation overlap, and replacing the existing communication with optimized communication, while preserving correctness. The optimization is source-to-source so that the output optimized source code can be fed into the latest optimizing compiler for any given architecture, to complement the communication-computation optimization with traditional optimizations. To target the scientific applications that could most benefit from the optimization, the implementation targets applications written in Fortran 90 using MPI communication. Significant research has focused on optimizing communication latency but none can handle explicitly parallel codes written using MPI, as these pose extra challenges. The ability to optimize explicitly parallel applications written using MPI makes our approach different from all the related studies that try to improve communication latency in cluster environments.

The following chapter describes the pre-push transformation, in preparation for Chapter 3, where the techniques developed to automate the transformation are presented. Chapter 4 describes the prototype implementation of the transformation system, Chapter 5 provides an overview of related work, and finally Chapter 6 closes with conclusions and suggestions for future work.

Chapter 2

A TRANSFORMATION TO IMPROVE COMMUNICATION-COMPUTATION OVERLAP

The transformation described in this chapter reduces the communication latency of explicitly parallel, message-passing scientific codes executing on clusters by restructuring code to provide opportunities for communication-computation overlap. This "pre-push" transformation, developed by Danalis et al. [9] was empirically proven to significantly reduce the runtime of two real-world scientific applications.

2.1 Cluster Architecture

Clusters are composed of a set of workstations (*nodes*) which communicate with each other by sending messages to each other across a node interconnection network. Programs executed on clusters are written in a *message-passing* style, where explicit communication directives orchestrate the sending and receiving of data amongst peer nodes. Although clusters are relatively inexpensive, they have the downside of high communication latency which is an impediment to their scalability, and a detriment to overall performance. One of the keys to reducing communication latency is the effective utilization of technologies to remove the burden of communication from the main processor, or CPU.

One enabling technology for this effort is memory-mapped network interfaces, such as cluster interconnects like Myrinet [6], Infiniband [17] or Quadrics [24], which are in wide use now and support the necessary functionality. In memory-mapped

networks, the data can be transferred from the main memory to the network interface and back, 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 [20] that efficient communication is achieved even if the kernel plays a role in the transfer, as long as the data can be DMAed directly from the user buffers to the network card.

Although such technology exists, few scientific applications have been written with structure that utilizes the potential of RDMA-enabled networks. Most scientific applications execute blocking communication operations, which wait until the communication completes before returning to the main computation. By using blocking communication calls, the capability of the network co-processor to perform communication while the CPU proceeds with the computation is wasted. Later in this chapter, we present a transformation that replaces blocking with non-blocking communication so that this untapped potential can be utilized.

2.2 Message Passing with MPI

The *Message Passing Interface* (MPI) [21], is a standardized, platform-independent library of message-passing subroutines. It is intended to increase the portability of message passing codes by creating a layer of abstraction above the platform-specific communication primitives used to pass messages. The techniques presented in this thesis are applicable to all message-passing code; however, only message-passing codes written using MPI are discussed herein to promote consistency and concreteness and because they are the target of the prototype implementation described in chapter 4.

With MPI, each node executes the same exact program; however, each node may branch differently, based upon its *rank* which is its unique positive integer identification number. Each node determines its rank by calling `MPI_COMM_RANK`

(Table 2.1), and determines the total number of nodes, NP, executing the program by calling `MPI_COMM_SIZE` (Table 2.2).

`MPI_COMM_RANK(comm, rank, IERR)`	
`comm`	communicator (handle)
`rank`	number of processes in the group of comm
`IERR`	error code

Table 2.1: `MPI_COMM_RANK`

`MPI_COMM_SIZE(comm, size, IERR)`	
`comm`	communicator (handle)
`size`	number of processes in the group of comm
`IERR`	error code

Table 2.2: `MPI_COMM_SIZE`

The simplest way to communicate involves the sender calling `MPI_SEND` to send the contents of an array to another node, which calls `MPI_RECV` to receive. However, many MPI codes, specifically ones that are the target of this study, use *collective communication* to communicate with all other nodes using a a single procedure call. The type of collective communication call targeted in this thesis is `MPI_ALLTOALL` (Table 2.4). An array, $\mathcal{A}_s$, of size N being exchanged using `MPI_ALLTOALL` is divided into NP equal-sized partitions, where NP is the number of nodes. When `MPI_ALLTOALL` is called, each node, for all k, $0 \leq k < NP$, sends the k^{th} partition, beginning at memory offset $k \times \frac{N}{NP}$ to the k^{th} node. In this way, data is exchanged by all the nodes, and the resulting array $\mathcal{A}_r$ contains the k^{th} partition from the k^{th} node.

`MPI_ALLTOALL` is synchronous and blocking, meaning that a call to it does not return until matching calls to `MPI_ALLTOALL` have been executed by the other

nodes, and all communication has completed. Synchronous, blocking communication is contrasted with asynchronous, non-blocking communication, which allows communication to be overlapped with computation. `MPI_ISEND` and `MPI_IRECV`, which implement asynchronous, non-blocking communication allow computation to proceed while communication is handled independently by the network co-processor. `MPI_ISEND` sends the data to the receiving node, but does not wait for the receiving node to execute the corresponding receive call, or for the send buffer to be sent to the receiver. Thus, execution of the program can continue as soon as a send or receive is initiated. At some point, before the communicated data is to be used, we must confirm that initiated sends and receives have completed, by calling `MPI_WAITALL` (Table 2.3). In the program transformation described in this thesis, we replace collective communication calls to `MPI_ALLTOALL` with non-blocking calls to `MPI_ISEND` and `MPI_IRECV`.

Asynchronous, non-blocking communication can also be achieved using one-sided I/O, in which one side, the sender, is responsible for transferring data. Using the address of the receive array on the receiver's machine, the sender writes directly to the receive array. The receiver can check to see if the transfer has completed, but no corresponding receive operation needs to be initiated. This communication model allows for more flexibility and reduced communication latency. The details of one-sided communication are omitted because in this thesis, we focus on asynchronous, non-blocking I/O using MPI, although the techniques presented can be easily extended to incorporate one-sided communication.

2.3 Overlapping Communication and Computation

Explicitly parallel, message-passing codes often perform some computation inside a loop, aggregating the results in an array, and then sending the aggregated results to other nodes. Using this traditional "compute and then communicate" structure, the execution time is τ_{tr} where $\tau_{tr} = \tau_{comp} + \tau_{comm}$. If somehow the

MPI_WAITALL(count, requests, status, IERR)	
count	size of requests array
requests	array of requests (handle)
status	status of request
IERR	error code

Table 2.3: MPI_WAITALL

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf recvcnt, recvtype, comm, IERR)	
sendbuf	starting address of send buffer (choice)
sendcount	number of elements to send to each process (integer)
sendtype	data type of send buffer elements (handle)
recvbuf	address of receive buffer (choice)
recvcount	number of elements received from any process (integer)
recvtype	data type of receive buffer elements (handle)
comm	communicator (handle)
IERR	error code

Table 2.4: MPI_ALLTOALL

communication can be overlapped with the computation, that is, if the communication can be made concurrent with some computation, execution time can be reduced, and this time is given by τ_{over} where $\tau_{over} = max(\tau_{comm}, \tau_{comp})$. Ideally, when $\tau_{comp} > \tau_{comm}$, τ_{comm} is totally hidden by τ_{comp}, so $\tau_{over} = \tau_{comp}$, which is the theoretical minimum, and represents perfect overlap. However, in practice, $\tau_{comp} < \tau_{over} \leq \tau_{tr}$ since there is always some overhead incurred in initiating send operations, and there are also leftover elements that need to be sent synchronously after all computations are complete.

For communication to be overlapped with computation, there needs to be some computation that can proceed after the send has been initiated, while it is being completed by the network co-processor. To preserve correctness, the send

operation must begin after the sender has generated the data, and finish before the receiver needs to use the data. Therefore, the communication needs to be placed with care. If placed too early, data will be sent before it is finalized, and if too late, old values of the array will be used by the receiver. Simply changing the original communication calls to their non-blocking analog is usually incorrect, since received data is often used immediately upon its receipt.

2.4 The Pre-Push Transformation

In order to provide the opportunity for communication-computation overlap and to satisfy the requirement that data be sent "not too early, but not too late", we restructure code so that data is "pre-pushed," or sent as it is generated, before it is needed. The transformation restructures code so that for every *tile* (subset) of the array that is generated, a non-blocking send is initiated, which is then completed by the network co-processor, while the CPU continues computing the next tile of the array. Also, a non-blocking receive to match each send is initiated by the receiver for each tile, when using two-sided communication. Note that pre-pushing data only allows the level of overlap to approach its theoretical maximum if the time to compute a tile exceeds the time to communicate a tile, effectively hiding the communication latency.

In general, to restructure code to pre-push results of its computation, we must first determine the three following pieces of information by analyzing the program code.

1. The computation loop(s) that write(s) to the array being sent, since it/they will be restructured to pre-push results.

2. The receive operation(s) corresponding to each send operation, since both the send and the receive must be transformed in concert.

3. The point in the receiver's execution where the receive array is done being used before the original receive operation. The sender cannot write to the receive array until this point is reached, or else values about to be used may be overwritten, resulting in incorrect results.

The last two pieces of information, in general, are difficult and in some cases impossible to determine automatically. The ability to determine these values automatically for arbitrary codes would make for a more generally applicable optimization, but is beyond the scope of this thesis, and is the subject of future research. Therefore, we restrict the optimization to a very specific, although common, communication-computation pattern, in order to make automating the transformation feasible. This pattern is described in the next section.

2.5 Transforming Scientific Codes

Scientific codes can be analyzed relatively easily, having regular structures with simple array access patterns. Many scientific codes contain frequently executed sections consisting of a double-nested loop in which the inner loop performs some heavy computation to compute an array of values which is then exchanged using `MPI_ALLTOALL` at the end of each iteration of the outer loop. This communication-computation pattern is the domain on which our current transformation is focused. More specifically, the transformation is applicable to a communication-computation pattern, where the communication is accomplished using a call to `MPI_ALLTOALL` that is definitely executed by all nodes after a single computation loop that all nodes definitely execute to compute the values of an array to be sent. Figure 2.1 shows an abstract target code fitting the pattern described above.

To ensure that the `MPI_ALLTOALL` call is executed by all nodes, it must not be in the body of any conditionals, which are typically used to have different nodes execute different statements. It may seem that any call to `MPI_ALLTOALL` is executed

```
integer A_s(1:NX)
integer A_r(1:NX)
...
do iy=1, NX  !outer loop
   do ix=1, NX  !inner computation loop nest
      ...
      A_s(ix) = ...
   enddo
   ...
   !sends A_s and receives into A_r
   call blocking-collective-commn(A_s,A_r)
enddo
```

Figure 2.1: Abstract target pseudocode

simultaneously by all nodes, because when one node calls `MPI_ALLTOALL`, it does not return until all other nodes also call it. However, it is possible for nodes to execute different calls to `MPI_ALLTOALL`, that are just matched due to their coincidence in time and argument values.

In the general case, there may be more than one computation loop which finalizes values of the array to be transmitted. Each of several loops preceding the communication might finalize disjoint subsets of the array. Although the possibility of having multiple computation loops correspond to a single communication exists, in this thesis, we only consider cases where there is only one computation loop that all nodes execute prior to communication. This allows us to assume that the sender and receiver are executing the same computation loop. If the sender and receiver are executing the same computation loop, after tiling the computation loop, the sender and receiver are synchronized at the start of each tile, so that the receiver knows which tile to receive. This knowledge is needed if the original communication is being replaced by two-way communication (as opposed to one-way communication, where the receiver needs no information about what it is about to receive). Also,

by knowing what section of code the receiver is executing when the sender is pre-pushing data in its computation loop, knowledge of whether the receive array is in use between the time of the first pre-push and the communication can be determined easily. As long as there are no uses of the receive array in the computation loop, it is guaranteed that it is safe to send into the receive array. In the case when the receive array is used in the computation loop, the pre-pushed data must be received into a newly introduced temporary array, which has to be copied to the receive array before the time of the original communication. If this extra copy needs to be introduced, the extra cost of copying will reduce the benefit of overlapping communication and computation, although as long as local memory copying is cheaper than network transfers, there should still be some benefit.

Two real-world parallel scientific codes containing instances of the target pattern are described here to provide a more concrete idea of the set of codes to which our optimization can be applied. These two codes are used in the empirical study described in Section 2.6. The first application, which we call *magnet* [11], was developed by physicists to investigate magneto-hydrodynamic turbulence through spectral methods. The second application, which we call *visco* [15], was developed by chemical engineers to simulate viscoelastic turbulent flow in a channel. Despite the differences in the structure, functionality, and programming styles of *magnet* and *visco*, the transformation is performed similarly. In both cases, the original code contains loops executing (one or two dimensional) *FFT* using the *FFTW* library. Because *FFTW* is one of the most common libraries used in scientific codes, and calling subroutines of *FFTW* usually results in code characteristic of our input pattern, we are confident that there are many existing codes to which the pre-push optimization is applicable.

Figure 2.2 shows a segment of the original code for *visco*. As can be seen, a computation kernel (real-to-complex FFT) is being executed in a doubly nested loop.

```
!C--------------------------------------------------------------------
!C FOURIER TRANSFORM IN X DIRECTION; SCALE BY 1/NX AFTER Z TRANSFORM
!C AND PREPARE ARRAY CAT2 FOR ALLTOALL BETWEEN X AND Z
!C--------------------------------------------------------------------
DO IZ = 1, (NZ/NPROC)
   DO IY = 1, NYP
      CALL REAL_TO_COMPLEX( REAL_IN=AP(:,IY,IZ), COMPLEX_OUT=A3 )
      DO IX = 1, (NX/2)
         IPROC  = (IX-1)/((NX/2)/NPROC)
         IXEFF  = IX - IPROC*((NX/2)/NPROC)
         CAT2(IXEFF,IY,IZ,IPROC+1) = A3(IX)
      ENDDO
   ENDDO
ENDDO
!C--------------------------------------------------------------------
!C   ALLTOALL BETWEEN X AND Z TRANSFORM
!C--------------------------------------------------------------------
NT = NYP*(NZ/NPROC)*((NX/2)/NPROC)
IF ( NPROC > 1 ) THEN
   CALL MPI_BARRIER( MPI_COMM_WORLD, ERR )
   CALL MPI_ALLTOALL( CAT2(1,1,1,1), NT, MPI_DOUBLE_COMPLEX, &
       CAT(1,1,1,1), NT, MPI_DOUBLE_COMPLEX, MPI_COMM_WORLD, ERR )
ELSEIF( NPROC == 1 )THEN
   CAT = CAT2
ENDIF
!--------------------------------------------------------------------
!FOURIER TRANSFORM IN Z DIRECTION
!FIRST REORDER A2(IZ,...)
!FOR REALITY ZERO OUT HIGHEST MODE IN Z DIR (THIS MODE HAS NO
!COMPLEX CONJUGATE AND MAY THEREFORE CAUSE NUMERICAL INSTABILITIES)
!--------------------------------------------------------------------
DO IX = 1, ((NX/2)/NPROC)
   DO IY = 1, NYP
      DO IZ = 1, NZ
         IPROC = (IZ-1)/(NZ/NPROC)
         IZEFF = IZ - IPROC*(NZ/NPROC)
         AUX(IZ,IY,IX) = CAT(IX,IY,IZEFF,IPROC+1)
      ENDDO
      CALL COMPLEX_TO_COMPLEX( SIGN=-1, N=NZ, INOUT=AUX(:,IY,IX) )
      AUX(NZHP,IY,IX) = DCMPLX(0.0D0,0.0D0)
   ENDDO
ENDDO
!--------------------------------------------------------------------
!CHEBYSHEV TRANSFORM IN Y-DIRECTION
!AND SCALE FOR X AND Z TRANSFORM BY 1 / (NX*NZ)
!--------------------------------------------------------------------
DO IX = 1, ((NX/2)/NPROC)
   DO IZ = 1, NZ
      A1 = AUX(IZ,:,IX)*OONXZ
      CALL CFCTMLT( A1=A1, A2=AS(:,IZ,IX), IS=-1 )
   ENDDO
ENDDO
```

Figure 2.2: Original code for *visco*

The resulting data is being packed into a new array, exchanged with all the peer nodes, and transposed upon arrival into a new array; then computation proceeds (with FFT being executed on the other dimensions).

To demonstrate the result of the transformation, Figure 2.1 shows an abstract target code before being transformed, and Figure 2.3 shows the same code after transformation. Sorting [1], LU Factorization [10], Finite differences, and multi-dimensional FFT constitute examples of algorithms fitting this abstract form, and can be transformed to exploit tile pipelining.

```
integer A_s(1:NX)
integer A_r(1:NX)
...
do iy=1, NX  !outer loop
   do ix=1, NX  !inner computation loop nest
      ...
      A_s(ix) = ...
      wait for communication of previous tile to complete
      if(ix mod K == 0) then
         to=...
         size=K
         call non-blocking-send(A_s(...),size,to,...)
         ...
         call non-blocking-recv(A_r(...),size,from,...)
      endif
   enddo
   ...
enddo
```

Figure 2.3: Abstract target pseudocode after transformation

The tiling of the computation loop nest is controlled by the parameter K and the level of nesting of the tile loop, in which the non-blocking communication is placed. As can be seen in Figure 2.3, K controls the number of iterations of the tile loop per tile. Determining the optimal tile size is not a trivial task, and is best performed by an automated system, since the value may change as applications migrate across platforms.

2.6 Evaluating the Transformation

Danalis et al. [9] manually transformed the two scientific applications, *visco* and *magneto*, using each of the following communication strategies.

1. `MPICH-GM 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 2.1. The *MPI* implementation is *MPICH-GM*, which uses the advanced features of the hardware (e.g., RDMA). This communication strategy is the control of the experiment, representing the original, unoptimized code.

2. `MPICH-GM ISEND` - The computation is reorganized into independent tiles and communication is performed in every tile, as shown in Figure 2.3. The communication primitives are the non-blocking `MPI_ISEND()` and `MPI_IRECV()` and the *MPI* implementation is *MPICH-GM*. This strategy tests the effects of communication-computation overlap using non-blocking MPI communication calls.

3. `Custom Library, one-sided I/O` - The program is structured similar to scheme 2, but the communication is handled by a low level library built directly on top of *GM* and providing one-sided I/O. This strategy tests the effects of communication-computation overlap combined with elimination of some of the inefficiencies due to the abstraction of MPI.

Experiments were performed varying several parameters, in order to compare their relative significance. In particular, for every communication strategy, the number of processors (NP), the tile size (K), and the problem size were varied. The execution time was the dependent variable measured in each trial. The experiments were run on a cluster of 20 machines connected with a Myrinet network.

Figure 2.4 shows the execution time of *visco* using the different communication strategies as the number of nodes increases, normalized to the execution time where no communication takes place (i.e., when only one node executes the program). Figure 2.5 displays the data for a larger problem size. From the two graphs, it is clear that overlapping communication and computation reduces runtime significantly, and the effect is consistent as the number of nodes increases, and as the

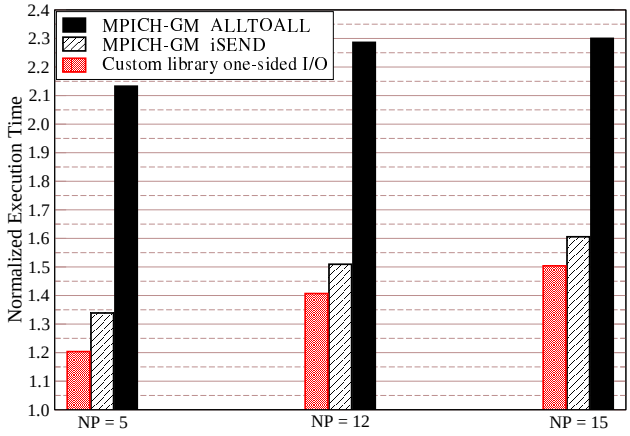


Figure 2.4: *Visco*, Problem size: 240x480x48 complex numbers ≈ 84MBytes

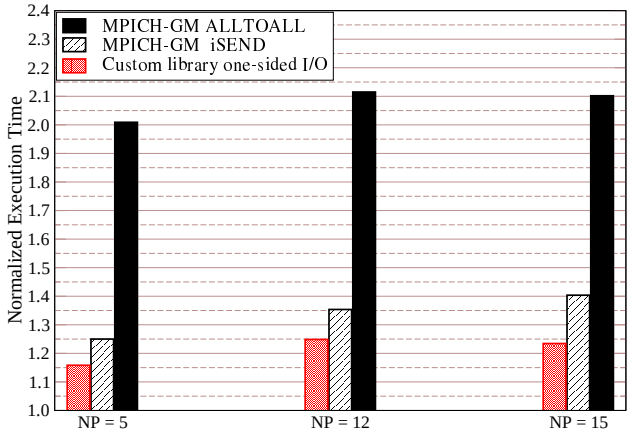


Figure 2.5: *Visco*, Problem size: 960x480x48 complex numbers ≈ 337MBytes

problem size increases. Additionally, Figure 2.6 demonstrates that execution time of *visco* varies widely as K is changed, but it seems to fit a pattern. A model is being developed to predict which values of K will result in the lowest run time. Since the results using *magnet* were similar to *visco*, they are not shown. With the pre-push optimization yielding such successful results, it is clear that developing a system capable of automating the transformation is worthwhile.

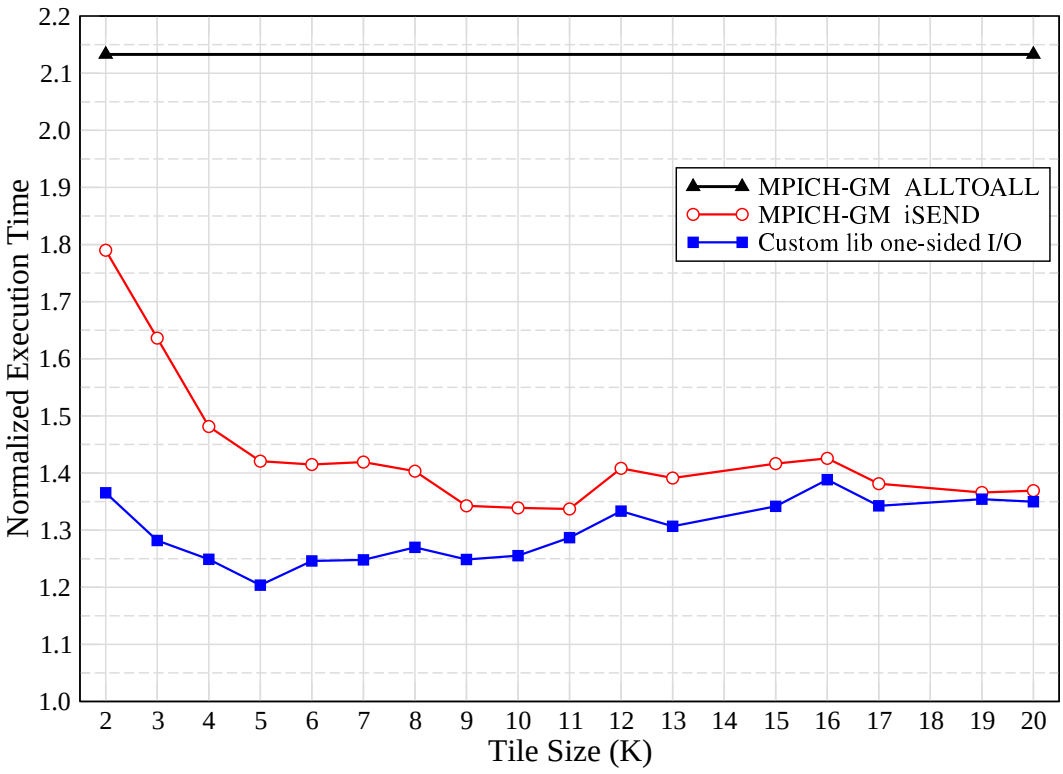


Figure 2.6: Effect of tile size (K) on performance of *visco*

Chapter 3

AUTOMATING THE TRANSFORMATION

In this chapter, we describe an automated approach to perform the pre-push transformation given in the last chapter.

3.1 Overview

The following steps provide an overview of the process by which the transformation is automatically applied. The details for each step are given in subsequent sections, which are referenced in this overview. Figure 3.1 shows a very simple example of code before transformation, and Figure 3.2 shows the same code after transformation. Note that the code in Figure 3.2 does not demonstrate all features of the transformation, but is rather kept simple. Later examples demonstrate all aspects of the transformation.

```
integer A_s(1:9)
integer A_r(1:9)
...
do ix = 1, NX  !ℓ
   ...
   A_s^f(ix) = ...
enddo
...
call MPI_ALLTOALL(A_s,NX,...,A_r,NX,...)  !C
```

Figure 3.1: Example: Code Before Transformation

1. Perform analysis to enable symbolic expression comparison throughout the program, which is used in some of the following steps. (Section 3.2)

2. Generate the set of all potential transformation tuples, *tuples*, which represent all the opportunities for transformation. Each tuple contains: a call, $\mathcal{C}$, to `MPI_ALLTOALL`, a send array $\mathcal{A}_s$, a receive array $\mathcal{A}_r$, a loop nest, ℓ, and the kind of computation (described later). ℓ is the loop that finalizes (writes to for the last time) $\mathcal{A}_s$ before $\mathcal{C}$. (Section 3.3)

3. For each transformation tuple, find the array use $\mathcal{A}_s^f(i_1, i_2, \ldots, i_n)$, where $i_{1,2,\ldots,n}$ are index expressions, that has no output dependences on it. During the runtime of ℓ, $\mathcal{A}_s^f$ refers to the value of the element most recently *finalized* (written to for the last time). (Section 3.4)

4. For the finalizing array reference $\mathcal{A}_s^f(i_1, i_2, \ldots, i_n)$, convert all index expressions to canonical form. (Section 3.5)

5. For *each* loop ℓ_k, $1 \leq k \leq m$, nested in and including ℓ, where ℓ_1 is the outermost loop of the loop nest ℓ, and ℓ_m is the innermost loop surrounding $\mathcal{A}_s^f$, perform *array access analysis* on $\mathcal{A}_s^f$. This analysis determines the size of the blocks of contiguously accessed array elements, or *blocks*, written during the runtime of ℓ_k, denoted $size(\ell_k, \mathcal{A}_s^f)$, and a set of the offsets of the blocks, denoted $offsets(\ell_k, \mathcal{A}_s^f)$. (Section 3.5)

6. Based on the array access analysis, choose a loop within the loop nest to split into tiles, ℓ_t, and the number of iterations of ℓ_t per tile, K, that gives the best performance. (Section 3.6)

7. If there is a loop, ℓ_{copy}, that copies an array temporarily holding intermediate results, $\mathcal{A}_t$, to $\mathcal{A}_s$, remove ℓ_{copy}, and treat $\mathcal{A}_t$ as $\mathcal{A}_s$. (Section 3.9)

8. Generate a communication loop nest ℓ_{comm}, which iterates through all $o \in offsets(\ell_t, \mathcal{A}_s^f)$ and contains non-blocking communication calls to transmit $size(\ell_t, \mathcal{A}_s^f)$ elements from $\mathcal{A}_s(o)$. (Section 3.8)

9. At the end of the body of ℓ_t, insert an `if` statement that executes every K iterations, containing ℓ_{comm}. This `if` statement initiates the exchange of each tile.

10. Insert a blocking call to wait for all outstanding receives from the previous tile to complete, before the `if` statement inserted in the previous step.

11. Insert code, to exchange any leftover elements not sent by the last tile, which result from K unevenly dividing the number of iterations of ℓ_t.

12. Insert code, after ℓ and before $\mathcal{C}$ to wait for the arrival of the last blocks of data after the end of ℓ.

13. Remove $\mathcal{C}$, the original communication.

3.2 Symbolic Expression Comparison

Several stages of the automation process rely on the ability to compare two expressions symbolically to determine their relative magnitude. A simple example of this would be if we needed to determine (at compile-time) that the expression x is less than the expression $x \times x$ during the runtime of the program. If we can guarantee (at compile-time) that $1 < x < \infty$, then we can infer that the expression, x, is less than the expression, $x \times x$. In cases where the expressions being compared only involve constants, it is possible to propagate the constants, evaluate the expressions and then compare them numerically. However, we cannot assume that every expression only uses constants. Fortunately, by using expression propagation, range propagation and

```
parameter NX=9
integer A_s(1:NX)
integer A_r(1:NX)
integer reqNum
integer requests(2*1)
integer K=3
...
do ix = 1, NX  !ℓ_t
   ...
   A_s^f(ix) = ...
   reqNum=0
   if(ix mod K == 0) then
      to=(ix*NX)/NP
      size=K
      if(ix > K) !after first tile
         call MPI_WAITALL(2*1,requests(1),stat,err)
      endif
      reqNum=reqNum+1
      call MPI_ISEND(A_s(ix − K + 1),size,...,to,...,requests(reqNum*2+1,...)
      reqNum=reqNum+1
      call MPI_IRECV(A_r(ix − K + 1),K,...,from,...,requests(reqNum*2+2,...)
   endif
enddo
call MPI_WAITALL(2*1,requests(1),stat,err)
```

Figure 3.2: Example: Code After Transformation

algebraic simplification, we can often compare expressions symbolically at compile-time. Blume and Eigenmann have developed algorithms using such techniques to perform symbolic expression comparison [5], and have implemented these algorithms in the Polaris compiler [4]. The details of symbolic expression comparison are beyond the scope of this thesis; however, we assume that expressions can be compared symbolically hereafter.

3.3 Identifying Opportunities for Transformation

Before any transformations can be performed, we must first identify all opportunities for semantics-preserving transformation. Once again, the transformation is applicable to transformation tuples, which consist of:

- $\mathcal{C}$, a call to `MPI_ALLTOALL` which is not in the body of any conditional statements
- $\mathcal{A}_s$, the array sent by $\mathcal{C}$, which is the 1^{st} argument to $\mathcal{C}$.
- $\mathcal{A}_r$, the array received by $\mathcal{C}$, which is the 4^{th} argument to $\mathcal{C}$.
- ℓ, the loop nest which finalizes all elements in $\mathcal{A}_s$, before $\mathcal{C}$. ℓ is the last loop nest that is a mutator of $\mathcal{A}_s$ that lexically precedes $\mathcal{C}$. Determining whether a loop nest is a mutator of $\mathcal{A}_s$ is discussed in Section 3.3.1. Just like $\mathcal{C}$, ℓ must not be in a conditional statement. Since ℓ is not in a conditional statement, and lexically precedes $\mathcal{C}$, it is guaranteed that ℓ is always executed before $\mathcal{C}$. With this guarantee, it is correct to pre-push data from inside ℓ, since we know that ℓ is finalizing the data in $\mathcal{A}_s$ before $\mathcal{C}$ is called.
- The way in which the computation of $\mathcal{A}_s$ is performed which is either: Intra-Loop Direct, Inter-Procedural Indirect, or Inter-Procedural Direct. These terms and how to determine the kind of computation is discussed in Section 3.3.2.

3.3.1 Mutator Loops

A loop nest, ℓ, which is a *mutator* of $\mathcal{A}_s$ can mutate, or write to, $\mathcal{A}_s$. The determination of whether ℓ is a mutator of $\mathcal{A}_s$ is decided on the basis of which of the following three cases holds, after inlining any procedures called in ℓ that are defined within the program being transformed.

Case 1 $\mathcal{A}_s$ is on the left hand side (*LHS*) of an assignment. Therefore, ℓ is a mutator of $\mathcal{A}_s$.

Case 2 There is no reference to $\mathcal{A}_s$. Hence, ℓ is not a mutator of $\mathcal{A}_s$.

Case 3 $\mathcal{A}_s$ is passed as an argument to a procedure, $\mathcal{P}$, which is called in ℓ. We must inspect $\mathcal{P}$ to see if it mutates $\mathcal{A}_s$. Just because $\mathcal{A}_s$ is passed to $\mathcal{P}$ does not guarantee that it is written by $\mathcal{P}$. In a language where parameters can be marked as constants and programmers always mark parameters as constants when appropriate, or be marked as being passed-by-value, $\mathcal{P}$ does not need to be inspected. However, we cannot rely on the programming language providing such constructs, or on programmers always using them. Therefore, we break case 3 into two subcases.

Case 3a If there are no other loops mutating $\mathcal{A}_s$ preceding $\mathcal{C}$, then it is clear that ℓ is a mutator, since there is nowhere else that $\mathcal{A}_s$ could have been written before being sent.

Case 3b Otherwise, there are other loops mutating $\mathcal{A}_s$ that precede $\mathcal{C}$. In this case, it is impossible to determine without user intervention whether ℓ mutates $\mathcal{A}_s$. Fortunately, it is unlikely for this case to occur in parallel scientific codes. In an actual implementation, an error message could be emitted and the user could be asked to verify whether ℓ mutates $\mathcal{A}_s$.

3.3.2 Kind of Computation

Once it is determined that ℓ is the final mutator loop nest of $\mathcal{A}_s$, we need to determine where and how the bulk of the computation performed in generating the contents of $\mathcal{A}_s$ takes place. The transformation is performed differently depending on the kinds of computation, which are as follows.

Intra-Loop Direct As seen in Figure 3.3, $\mathcal{A}_s$ appears on the LHS of an assignment statement where the right hand side (RHS) is not an array reference, but rather either a scalar reference, a binary arithmetic expression, or a function call. In this case, the computation must be located in ℓ, and the results of the computation are directly written into $\mathcal{A}_s$.

Inter-Procedural Indirect As demonstrated in Figure 3.4, $\mathcal{A}_s$ appears on the LHS of an assignment statement where the RHS is a different array, $\mathcal{A}_t$. $\mathcal{A}_t$ is passed as an argument to $\mathcal{P}$, a procedure called before the assignment. Because $\mathcal{P}$ is inside a loop, it computes part of the results that are needed for $\mathcal{A}_s$ in each iteration. The assignment statement is contained in a loop, ℓ_{copy},

```
do ix = 1, N  !loop nest ℓ
   A_s(ix) = ...
enddo
```

Figure 3.3: Example: Intra-Loop Direct

succeeding $\mathcal{P}$, which copies or transposes $\mathcal{A}_t$ to $\mathcal{A}_s$. The computation takes place in $\mathcal{P}$, but since it is stored in a temporary array $\mathcal{A}_t$ and then copied or transposed to $\mathcal{A}_s$, the computation of $\mathcal{A}_s$ is indirect. The goal of Section 3.9 is to remove ℓ_{copy}, and directly send the contents of $\mathcal{A}_t$, as this is more efficient.

```
do iy = 1, N  !loop nest ℓ
   call P(..., A_t)
   do ix = 1, N  !ℓ_copy
      tx = ix/X
      ty = ix/(X*X)
      A_s(tx,ty,iy) = A_t(ix)
   enddo
enddo
```

Figure 3.4: Example: Inter-Procedural Indirect

Inter-Procedural Direct In codes similar to that in Figure 3.5, $\mathcal{A}_s$ is an argument to a procedure, $\mathcal{P}$, which computes part of the contents of $\mathcal{A}_s$ each time it is called. The computation of $\mathcal{A}_s$ is direct and is located in $\mathcal{P}$. Since this kind of computation is uncommon in scientific codes, we omit its consideration in the rest of this thesis, to simplify the exposition.

```
do iy = 1, N  !loop nest ℓ
   call P(..., A_s(...,iy))
enddo
```

Figure 3.5: Example: Inter-Procedural Direct

3.4 Array Dependence Analysis

For a given transformation tuple, where ℓ is the computation loop nest, it is our goal to determine which parts of the send array, $\mathcal{A}_s$, can be safely sent at some time, t, in the run time of ℓ. If an array reference, $\mathcal{A}_s^2$, at time t_2, overwrites elements previously written at time t_1 by another array reference, $\mathcal{A}_s^1$, then the element referenced by $\mathcal{A}_s^1$ is unsafe to send between time t_1 and t_2, as it will be written to again at time t_2. By sending at t, $t_1 \leq t \leq t_2$, the send would be premature, leading to incorrect results.

Using array dependence analysis, it is possible to systematically determine whether the element referenced by a given array reference is safe to send. There are several different types of data dependence relations; the one relevant to this discussion is the output dependence relation.

Output Dependence [32] If any elements accessed by an array reference are assigned in one statement s_1 and reassigned in a subsequently executed statement s_2, s_2 is output dependent on s_1, which is written as $s_2\ \delta^o\ s_1$.

```
do iy = 2, N  !ℓ
   As(iy) = ...  !s1
   As(iy-1) = f(As(iy-1)) ...  !s2
enddo
```

Figure 3.6: Example: Output Dependence

To test whether the value that is referenced by some assignment to $\mathcal{A}_s$, s, is safe to send, we check that none of the other assignment statements to $\mathcal{A}_s$ are output dependent on s_1. Let S be the set of all assignments to $\mathcal{A}_s$ in ℓ. As long as we assume that all $s \in S$ reference the same set of elements by the end of the execution of ℓ, there will be exactly one assignment statement, s_f, such that for all other $s \in S$, $s \not\delta^o\ s_f$. Since we make that assumption, the array reference that

is safe to send is the one on the LHS of s_f, which is such that for all other $s \in S$, $s \not\delta^o s_f$. This finalizing array reference is denoted $\mathcal{A}_s^f$. Figure 3.6 depicts the output dependence $s_1 \ \delta^o \ s_2$. In this case, since $s_2 \not\delta^o s_1$, $\mathcal{A}_s^f = \mathcal{A}_s(iy-1)$.

The techniques for determining the data dependences in ℓ are beyond the scope of this thesis, but an excellent source of information on the topic is contained in Wolfe, 1991 [32].

3.5 Array Access Analysis

Using the techniques of the previous section, we can determine which array reference has no output dependences on it. This array reference, $\mathcal{A}_s^f$, represents the latest element to have been finalized in ℓ, which is thus safe to send. At this point, it is correct to transform the program so that each element referenced by $\mathcal{A}_s^f$ is sent one at a time, as each is generated. Although correct, it is desirable for reasons of efficiency to *aggregate* these single element sends into fewer, larger send operations. Since a send operation can only transfer one contiguous block of elements at a time, and $\mathcal{A}_s^f$ may have been written in several disjoint contiguous blocks during the execution of the last tile, we need to find the set of contiguous blocks of array elements written to $\mathcal{A}_s^f$.

To do so, we use *array access analysis*, which in general, produces descriptions of various levels of accuracy and detail of the region of an array accessed by an array reference. Previous research by other authors has developed techniques for performing array access analysis. A good overview of the techniques is contained in Paek, Hoeflinger and Padua [22]. Most existing methods are too complicated for our purposes, providing precision that may be useful in certain cases, but that make developing and implementing the rest of our techniques too burdensome. We use the simplest, most course-grained access representation, known as a partial triplet, since it is adequate for our purposes and is easy to implement. A partial triplet just contains the symbolic upper and lower bound of an index expression. We formulate

our access analysis in such a way that is amenable to tiling, extend it to aggregate accesses into contiguous blocks so that we can generate communication from it, and combine it with a variation of index traversal order, which was first defined in Balasundaram and Kennedy [2].

In order to find which of the m loops nested in ℓ, *if made the tile loop*, gives the best performance, we need to analyze *each* loop $\ell_{k,1\leq k\leq m}$ nested in and including ℓ, since the contiguousness of the array access region changes for each ℓ_k, and this changes the number of send operations required.

Before array access analysis can be performed, the array reference is put into a canonical form, in order to simplify the analysis of each index expression. Although it is possible to perform the analysis directly on the original array reference, doing so complicates the exposition. Additionally, if we were instead to perform the analysis directly, any shortcomings of the process for converting to canonical form would be carried over to the analysis stage, so no power is lost by first converting to canonical form. To be in canonical form, every index expression, $i_{k,1\leq k\leq n}$, of $\mathcal{A}_s^f$ must contain only constants, loop invariant variables, and loop induction variables. This means that variables whose values vary during the runtime of ℓ, but are not loop induction variables cannot be in i_k. If there are any such computed variables in the index expression, demand-driven expression propagation must be performed. The only time it is impossible to do expression propagation is when there is a cycle in the data dependence graph. This situation is unlikely to occur, but if it does, the programmer of the code must manually transform it to the canonical form. An example of converting to canonical form is given in Figure 3.7.

3.5.1 Index Expression Bounds

After putting the index expression $\mathcal{A}_s^f(i_1, i_2, \ldots, i_n)$ into canonical form, we construct the symbolic expressions representing the upper and lower bounds of the values of the index expressions during the execution of ℓ_k. We use these bounds

```
!  Original
do iy = 1, N  !loop nest ℓ
   do ix = 1, N
      a = ix
      b = iy + a
      A_s^f(a,b,1) = ...
   enddo
enddo

!  Canonical Form
do iy = 1, N  !loop nest ℓ
   do ix = 1, N
      A_s^f(ix,iy+ix,1) = ...
   enddo
enddo
```

Figure 3.7: Example: Before and After Converting to Canonical Form

to infer which elements of $\mathcal{A}_s$ have been written during the execution of ℓ_k, whose values are the ones that can be sent.

The upper and lower bounds are denoted by $u(\ell_k, i)$ and $l(\ell_k, i)$, respectively. But first, we must construct the upper and lower bounds of each variable that the index expressions use. There are three different types of variables that remain in index expressions after converting to canonical form; the way to construct their lower and upper bounds, is as follows.

- *Invariant*: i is defined outside the scope of ℓ_k, which guarantees that it is invariant, or never changes, during the runtime of ℓ_k. The lower and upper bounds of the index expression are both i since i's value is invariant.

- *Tile Induction*: i is the loop induction variable of ℓ_k, which is the loop that is potentially being split into tiles. The upper bound is represented by i because it is the current value of i, and we are assuming that induction variables are only incremented forward. Since K is the number of iterations of the tile loop

per tile, the upper bound of the previous tile is $i - K$. By adding one to the upper bound of the previous tile, we have the lower bound of the current tile which is $i - K + 1$.

- *Intra-Tile Induction*: i is the induction variable of a loop $\ell_{j,k<j\leq m}$, that is nested inside the potential tile loop. Therefore, all values between the lower and upper bound of the loop are taken by i during the execution of the potential tile. Hence, the upper and lower bounds of i during the potential tile are the upper and lower bounds of i declared in the loop, ℓ_j.

After constructing the upper and lower bounds of the variables that the index expressions depend on, it is straightforward to construct the upper and lower bounds of the index expressions. To construct the upper bound of an index expression, simply replace all references to variables with their upper bounds which have been previously constructed. The lower bound of an index expression is constructed analogously. The following equations summarize how to determine the lower and upper bounds of variables and index expressions.

$$upper(\ell_k, i) = \left\{ \begin{array}{l} i, \textbf{ invariant} \\ i, \textbf{ tile} \\ upper(\ell_j), \textbf{ intratile} \\ i \mid (\forall v \in i) v \leftarrow upper(v), \textbf{ expression} \end{array} \right\}$$

$$lower(\ell_k, i) = \left\{ \begin{array}{l} i, \textbf{ invariant} \\ i - k + 1, \textbf{ tile} \\ lower(\ell_j), \textbf{ intratile} \\ i \mid (\forall v \in i) v \leftarrow lower(v), \textbf{ expression} \end{array} \right\}$$

3.5.2 Index Count and Span

Once the symbolic bounds for each index expression are constructed, we produce a symbolic expression representing the number of times the index will be

```
integer A_s(1:NX,1:NY,1:NZ)
...
a=5
do iz = 5, NZ+5  !ℓ1
   do iy = 1, NY  !ℓ2 (potential tile loop)
      do ix = 1, NX  !ℓ3 (intra-tile loop)
         A_s^f(ix,iy,iz-a) = ...
      enddo
   enddo
enddo
```

i_k of $\mathcal{A}_s^f$	$lower(\ell_2, i_k)$	$upper(\ell_2, i_k)$
i_1	1	NX
i_2	$iy - K + 1$	iy
i_3	$iz - a$	$iz - a$

Figure 3.8: Example: Index Expression Bounds

incremented during the execution of a single tile. We call this expression the index expression count, and it is given by

$$count(\ell_k, i_j) = u(\ell_k, i_j) - l(\ell_k, i_j) + 1$$

We also decide whether or not the value of an index expression fully or partially spans the length of the entire dimension during the execution of a tile. For an index expression i_j, where the declared size of the corresponding array dimension is Δ_j, we have

$$span(\ell_k, i_j) = \left\{ \begin{array}{l} full,\ count(i_j) = \Delta_j \\ partial,\ count(i_j) \neq \Delta_j \end{array} \right\}$$

In Figure 3.10, i_1 and i_3 have a full span, whereas i_2 has a partial span. In order to compare Δ_j with i_j, for equality, we must be able to compare symbolic expressions, which is discussed in Section 3.2.

3.5.3 Index Traversal Order

The index traversal order can be viewed as an ordering of index expressions by the "speed" they are traversed relative to one another. Consider the array reference, $\mathcal{A}(ix, iy)$, in the loop

```
do iy = 1, NZ !ℓ
   do ix = 1, NY
      A(ix,iy)=...
   enddo
enddo
```

Here, ix goes through its entire range before iy is incremented. Therefore, we say that ix is faster than iy, or that $speed(\ell, ix) > speed(\ell, iy)$.

By determining the index traversal order, we can infer not only the set of elements accessed, but the sequence in which the elements are accessed. We will depend on the ability to order index expressions in Section 3.9, where we attempt to remove the copy loop when the computation is inter-procedural indirect. Index traversal order was first defined by Balasundaram and Kennedy [2]. However, index expressions containing division operations cannot be handled by their definition. A significantly modified version of the original definition is presented below and is then extended to handle division operations.

Dominant Induction Variable Let $\mathcal{A}(i_1, i_2, \ldots, i_n)$ be an array reference contained within m loops with induction variables $\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_m$ with nesting depth k, $1 \leq k \leq m$, where $\ell_1 = \ell$ is the outermost loop, and ℓ_m is the innermost loop. The dominant induction variable, D_j, of i_j is determined as follows:

- If i_j does not contain any induction variables $\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_m$, i_j has no dominant induction variable.
- If i_j consists of *exactly one* induction variable, $\mathcal{I}_{l,1 \leq l \leq m}$, $\mathcal{I}_l$ is the dominant induction variable.

- If i_j contains more than one induction variable, the dominant induction variable is the one defined in the most deeply nested loop (i.e., having the greatest j).

Traversal Order Let $D_1, D_2, \ldots, D_p$ be the list of distinct dominant induction variables of the index expressions $i_1, i_2, \ldots, i_n$. Construct the sets $V_1, V_2, \ldots, V_p$, where: $V_i = \{i_{j,1\leq j\leq n} \mid D_i \text{ is the dominant induction variable of } i_j\}$. Also construct the set $V_0 = \{i_{j,1\leq j\leq n} \mid i_j \text{ has no dominant ind. var.}\}$. The traversal order of $\mathcal{A}$ is $V_0 \succ V_1 \succ \ldots \succ V_p$, where $V_1 \succ V_2$ means that the index expressions in V_2 are incremented faster than those in V_1. When each set V_i contains a single unique index expression, as a shorthand notation, we order the index expressions directly. For instance, if $V_1 = \{i_1\}$ and $V_2 = \{i_2\}$, we write $i_1 \succ i_2$. Furthermore, from such an ordering, we write that the fastest index expression has $speed(\ell_k, i) = 1$, the second fastest index expression has $speed(\ell_k, i) = 2$, and so on up to the slowest index expression which has $speed(\ell_k, i) = n$

To see why the above definitions need to be extended to handle division operations, consider the two index expressions, $i_1 = \frac{D_k}{e_1}$ and $i_2 = \frac{D_k}{e_2}$. If $e_2 > e_1$, then even though they are in the same set V_k, i_2 is traversed more slowly than i_1. To extend the above definition to handle division operations, we split the set V_k into the sets $V_k^1, V_k^2, \ldots, V_k^q$, which are ordered as

$$V_1^1 \succ V_1^2 \succ V_1^{\cdots} \succ V_1^q \succ V_2^{\cdots} \succ \ldots \succ V_p^{\cdots}$$

To split V_k, we find all the expressions that divide D_k for all the index expressions in V_k. More concretely, we are finding the e in $i = \ldots + \frac{D_k}{e} + \ldots$. If D_k is not divided in i, we consider e to be 1. Now, label each expression, $e_{j,1\leq j\leq q}$, so that all $e_j > e_{j+1}$, effectively matching the order of the subscripts inversely to the magnitude of the expressions. We are able to order the expressions using symbolic expression comparison, discussed in Section 3.2. We construct each V_k^j such that

$$V_k^j = \{i \in V_k \mid i \text{ contains } \tfrac{D_k}{e_j}\}$$

Figure 3.9 contains an example demonstrating the ordering of index expressions.

```
do iz = 1, NZ  !ℓ1
   do iy = 2, NY-1  !ℓ2
      call P(..., A_t)
      do ix = 1, NX  !ℓ3
         A_s^f(ix mod NP,iz,iy - 1,ix/NP) = A_t(ix)
      enddo
   enddo
enddo
```

i_k of $\mathcal{A}_s^f$	dominant ind var	loop	set
i_1	ix	ℓ_3	V_3^2
i_2	iz	ℓ_1	V_1^1
i_3	iy	ℓ_2	V_2^1
i_4	ix	ℓ_3	V_3^1
$i_2 \succ i_3 \succ i_4 \succ i_1$			

Figure 3.9: Example: Index Traversal Order

3.5.4 Contiguous Block Size and Offsets

Having performed the analysis of the preceding sections, we can now enumerate the set of all elements accessed by $\mathcal{A}_s^f$ during the execution of a single tile, which is called the *array access region* given by,

$$region(\ell_k, \mathcal{A}_s^f) = \left\{ \mathcal{A}_s^f(I_1, I_2, \ldots, I_n) \;\middle|\; \begin{array}{c} l(\ell_k, i_1) \leq I_1 \leq u(\ell_k, i_1), \\ l(\ell_k, i_2) \leq I_2 \leq u(\ell_k, i_2), \\ \ldots, \\ l(\ell_k, i_n) \leq I_n \leq u(\ell_k, i_n) \end{array} \right\}$$

Although we have the region of $\mathcal{A}_s^f$ written during K iterations of ℓ_k, we are still unable to generate aggregated send operations, which is the reason for performing the array access analysis in the first place. Again, since send operations can only transfer contiguous blocks of memory, and the accessed elements may not be contiguous, we must develop a representation of the accessed elements that is expressed

in terms of the size and offsets of contiguous blocks of elements, which can each be sent by a single send operation.

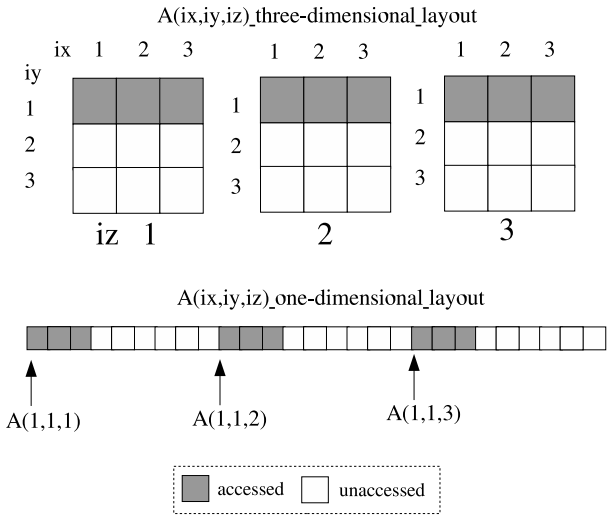
Determining the contiguous region of memory space accessed by a one-dimensional array reference $\mathcal{A}(i_1)$, is easy. For example, given $l(\ell_k, i_1) = 5$ and $u(\ell_k, i_1) = 15$, and assuming that i_1 monotonically increases with $step = 1$, the contiguous block consists of all elements $\mathcal{A}(I_1)$ such that $l(\ell, i_1) = 5 \leq I_1 \leq u(\ell, i_1) = 15$. We can express this access pattern as the size and offset of a single contiguous block. In this case, the size of the contiguous block is $count(\ell, i_1) = 10$ and the offset is $l(\ell, i_1) = 5$. We assume that all index expressions are monotonic functions; Paek, Hoeflinger and Padua concluded that most are monotonic [22].

For a multi-dimensional array reference $\mathcal{A}(i_1, i_2, ..., i_n)$, there will be a number of contiguous blocks of the same size, having different offsets. Breaks in contiguousness are caused by index expressions that only partially span their full range during the execution of a tile. In Figure 3.10, for $\mathcal{A}_s^f$, $span(\ell, i_1) = full$, $span(\ell, i_2) = partial$, and $span(\ell, i_3) = full$. The array pictured is a three-dimensional cube, with i_1 traversing the columns, i_2 traversing the rows, and i_3 traversing the planes. Since i_2 never increments during the runtime of ℓ, only the first row of each plane is accessed. Consequently, the access region is broken into three contiguous blocks of size three, with $offsets(\ell, \mathcal{A}_s^f) = \{\langle 1,1,1\rangle, \langle 1,1,2\rangle, \langle 1,1,3\rangle\}$.

In general, for an array reference $\mathcal{A}(i_1, i_2, ..., i_n)$ where $\forall i_{j,1\leq j<m\leq n}\ span(i_j) = full$ and $span(i_m) = partial$, the size of the contiguous blocks accessed by $\mathcal{A}$ is given by

$$size(\ell, \mathcal{A}) = \prod_{k=1}^{m} count(\ell, i_k)$$

The offsets of the contiguous blocks in $\mathcal{A}$ are given by the set of n-tuples, $offsets$,




```
integer As(1:3,1:3,1:3)
...
do iz = 1, 3  !ℓ
   do ix = 1, 3
      Asf(ix,1,iz) = ...
   enddo
enddo
```

Figure 3.10: Example: Contiguousness

where

$$offsets(\ell, \mathcal{A}) = \left\{ \langle 1, 1, \ldots, 1, I_m, I_{m+1}, \ldots, I_n \rangle \;\middle|\; \begin{array}{c} l(\ell, i_m) \leq I_m \leq u(\ell, i_m), \\ l(\ell, i_{m+1}) \leq I_{m+1} \leq u(\ell, i_{m+1}), \\ \ldots, \\ l(\ell, i_n) \leq I_n \leq u(\ell, i_n) \end{array} \right\}$$

3.6 Choosing Tile Size

After performing the array access analysis for each loop nested in ℓ, we must determine how many iterations, K, of which loop, ℓ_t, will comprise each tile. As mentioned in Section 2.6, determining K and ℓ_t is not trivial as there are several

tradeoffs involved. For each possible tile loop ℓ_k, the number of contiguous blocks differs due to variations in the span of index expressions, which means that the number and size of messages that are needed also differs. Additionally, increasing K leads to a larger tile size which corresponds to fewer and larger messages. Having as few transfer operations as possible is desirable 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. On the other hand, as K increases, the size of the last tile, whose communication typically cannot be overlapped with computation, increases.

To determine which values of K and ℓ_t will result in greatest efficiency, we use information gathered from the previous analysis stages along with timing information collected at runtime. The goal is to find the values of K and ℓ_t that minimize the following formula which represents the time spent in excess of that achieved with perfect communication-computation overlap.

$$(numTiles \times max(0, \tau_{comm} - \tau_{comp})) + \tau_{comm}$$

where:

- $numTiles$ is the total number of tiles in ℓ. Let $range(\ell_k)$ denote the upper bound minus the lower bound of loop ℓ_k. Then, $numTiles = \prod_{j=1}^{k-1} range(\ell_j) \times \frac{range(\ell_k)}{K}$.

- τ_{comp} is the time to compute a single tile, which can be found through profiling.

- τ_{comm} is the time to communicate the data generated by a single tile, which can be approximated as $numSends \times timeToSend(size(\ell_k, \mathcal{A}_s^f))$.

  - $numSends$ is the number of sends per tile which is equal to $\mid offsets \mid$, which can be constructed directly as $\prod_{k=m+1}^{n} count(I_k)$ when $\forall i_{j,1 \leq j < m}\ span(\ell, i_j) = full$, and $span(\ell, i_m) = partial$.

– $timeToSend(s)$ is the time to send a message with size of s, which can be approximated by profiling the system with a subset of the possible values of s and then interpolating to find specific values of s. Each message is of $size = size(\ell_k, \mathcal{A}_s^f)$.

Some of the input to the above formula contains symbolic expressions, for which the exact values may not be known at compile-time. Therefore, the tile size prediction routine must operate during the runtime of the code being transformed. The details of this profiling and prediction system are beyond the scope of this thesis, and are the subject of ongoing research.

3.7 Simulating MPI_ALLTOALL

Before generating the non-blocking communication, we first need to discuss how to preserve the semantics of the original call to `MPI_ALLTOALL` by simulating its node-wise partitioning of the send array. `MPI` is reviewed in Section 2.2, but here we elaborate on the semantics of `MPI_ALLTOALL`.

An n-dimensional array $\mathcal{A}_n(i_1, i_2, \ldots, i_n)$, where Δ_j is the size of the j^{th} dimension, is actually stored in memory as a one-dimensional array, $\mathcal{A}_1(i_1)$, for which the size of i_1 is $\gamma = \prod_{k=1}^{n} \Delta_k$. When $\mathcal{A}_1$ is exchanged using `MPI_ALLTOALL` and NP is the number of nodes, $\mathcal{A}_n$ is divided into NP equal-sized partitions of size $\frac{\gamma}{NP}$. In this case, the p^{th} partition, where $0 \leq p < NP$, begins at $\mathcal{A}_1(p\frac{\gamma}{NP} + 1)$. Converting this one-dimensional representation of the array back to its original multi-dimensional representation, we have the p^{th} partition beginning at $\mathcal{A}_n(1, 1, \ldots, p \times \frac{\Delta_n}{NP} + 1)$. When `MPI_ALLTOALL` is called, each node sends the p^{th} partition to the p^{th} node. In this way, data is exchanged by all the nodes, and the resulting array for each node contains the p^{th} partition from the p^{th} node. Figure 3.11 shows the partitioning of the three-dimensional and equivalent one-dimensional representation of the array $\mathcal{A}_3$ by `MPI_ALLTOALL`, according to the above specifications.

Given the semantics of MPI_ALLTOALL, we can find the mapping from a contiguous block offset to the rank of the node the element would be sent to by MPI_ALLTOALL. Recall that a contiguous block offset is an n-tuple, $\langle I_1, I_2, \ldots, I_n \rangle$, representing the base offset of a contiguous block starting at the element referenced by $\mathcal{A}(I_1, I_2, \ldots, I_n)$. The mapping is

$$destination(\mathcal{A}, \langle I_1, I_2, \ldots, I_n \rangle) = (I_n - 1)/\frac{\Delta_n}{NP}$$

3.8 Communication

Now that ℓ_t and K have been determined, we can use the size and offsets of the blocks that are written by $\mathcal{A}_s^f$ and the technique for finding the destination of a block, to generate the non-blocking communication to replace the original blocking communication. We will use MPI_ISEND to send, and MPI_IRECV to receive the blocks finalized during the execution of the last tile.

For a block offset $o \in offsets$, we can generate a send operation and its corresponding receive as follows.

```
size=size(ℓ_t, A_s^f)
to=destination(A_s^f, < I_1, I_2, ..., I_n >)
call MPI_ISEND(A_s^f(I_1, I_2, ..., I_n),size,to,...)
if(to == myRank) then
   do from = 0, NP-1
      call MPI_IRECV(A_r(I_1, I_2, ..., 1 + from*Δ_n/NP + I_n%Δ_n/NP),size,from,...)
   enddo
endif
```

The if statement and loop surrounding the call to MPI_IRECV iterates through all the node ranks if the destination of the current block is equal to its own rank. The if statement is necessary because a node should only receive a contiguous block if it is mapped to its rank, and the loop is needed to receive from all the nodes, since they each have a block to send.

Now that we can exchange a single contiguous block which is a particular $o \in offsets$, we describe how to enumerate and exchange *all* offsets using a loop nest, ℓ_{comm}, containing the replacement communication as given above. Recall that the set of offsets is defined as

$$offsets(\ell, \mathcal{A}) = \left\{ \langle 1, 1, \ldots, 1, I_m, I_{m+1}, \ldots, I_n \rangle \;\middle|\; \begin{array}{c} l(\ell, i_m) \le I_m \le u(\ell, i_m), \\ l(\ell, i_{m+1}) \le I_{m+1} \le u(\ell, i_{m+1}), \\ \ldots, \\ l(\ell, i_n) \le I_n \le u(\ell, i_n) \end{array} \right\}$$

which translates into the following loop nest, ℓ_{comm}, that generates all $o \in offsets$

```
do I_m = l(ℓ_t, i_m),  u(ℓ_t, i_m)    !ℓ_comm
   do I_{m+1} = l(ℓ_t, i_{m+1}),  u(ℓ_t, i_{m+1})
      ...
         do I_n = l(ℓ_t, i_n),  u(ℓ_t, i_n)
            !⟨1, 1, ..., 1, I_m, I_{m+1}, ..., I_n⟩    !o ∈ offsets
            ...
            MPI_ISEND(A_s^f(I_1, I_2, ..., I_n),...)
            ...
         enddo
      ...
   enddo
enddo
```

ℓ_{comm} needs to be placed in an `if` statement at the end of ℓ_t, which executes every K iterations, controlling the tiling of the loop, as follows.

```
do ix = 1, NX   !ℓ_t
   ...
   if(ix mod K == 0) then
      ℓ_comm
```

```
   endif
enddo
```

Also, provisions need to be made to send the data generated during any iterations leftover by the range of ℓ_t being unevenly divided by K. We therefore extend the code above as follows to handle leftover iterations.

```
k=K
...
do ix = 1, NX  !ℓ_t
   ...
   !every K iterations, or the last iteration
   if (range(ℓ_t) mod k == 0 ∨ ix == NX) then
      k = K
      !if k unevenly divides ix, which means this is the last iteration
      if (ix mod k ≠ 0) then
         !set k so that it is equal to the number of leftover iterations
         k = ix mod k
      endif
      ℓ_comm
   endif
enddo
...
```

Finally, by putting all of the above together, and inserting calls to `MPI_WAITALL` to wait for the communication of the previous tile to complete before the start of each tile, we have the code in Figure 3.12, which is inserted at the end of ℓ_t. In this code, reqNum represents how many communication operations have been initiated so far in the current tile. $numSends$ is the number of sends per tile which is equal to $\mid offsets \mid$, which can be constructed directly as $\prod_{k=m+1}^{n} count(I_k)$ when $\forall i_{j,1\leq j<m}\ span(\ell, i_j) = full$, and $span(\ell, i_m) = partial$.

3.9 Removing Redundant Array Copies

As discussed in Section 3.3.2 and shown in Figure 3.4, sometimes the computation is inter-procedural indirect, where the actual computation of the data contained in $\mathcal{A}_s$ is located in a procedure, $\mathcal{P}$, called in ℓ. Each call to $\mathcal{P}$ computes a portion of the final results and writes them to $\mathcal{A}_t$ (for temporary) which is passed by reference. After $\mathcal{P}$, the contents of $\mathcal{A}_t$ are copied to $\mathcal{A}_s$ in a copy loop, ℓ_{copy}. The original purpose of such intermediate array copies is to aggregate the partial results computed by each call to $\mathcal{P}$ so that they can be sent all together at the end of ℓ. However, since we are "unaggregating" the sends in order to overlap communication with computation, the copy is no longer needed, and we can directly send the contents of $\mathcal{A}_t$. Doing so can reduce runtime by eliminating the time taken to copy $\mathcal{A}_t$ to $\mathcal{A}_s$.

We only consider copy loops of the following form, which are the most common and easy to analyze. ℓ_{copy} must contain a single assignment statement with RHS $\mathcal{A}_s$ and LHS $\mathcal{A}_t$. $\mathcal{A}_t$ must be of the form $\mathcal{A}_t(t_1, t_2, \ldots, t_n)$ where $t_n \succ t_{n-1} \succ \ldots \succ t_1$, which means that $\mathcal{A}_t$ must be traversed linearly with a stride of 1. In this situation, the data is transferred from $\mathcal{A}_t$ to $\mathcal{A}_r$ by copying from $\mathcal{A}_t$ to $\mathcal{A}_s$ and then sending from $\mathcal{A}_s$ to $\mathcal{A}_r$. This can be represented as $\mathcal{A}_r \xleftarrow{send} \mathcal{A}_s \xleftarrow{copy} \mathcal{A}_t$. By transitivity, we can eliminate the copy and still complete the same operation by the equivalent $\mathcal{A}_r \xleftarrow{send} \mathcal{A}_t$. We can remove the redundant copy by sending one element of $\mathcal{A}_t$ at a time. Although sending one element at a time is wasteful, it is still instructive to see how to do so before advancing to the sending of whole blocks of $\mathcal{A}_t$ at a time. Figure 3.13 contains a code segment, before and after removing the copy.

Once again, we would like to send the largest blocks possible in order to minimize the number of send operations. To do so, we proceed similar to the way we send all blocks in the previous section, but here we have the additional requirement of having to send the blocks in the same order that they are copied to preserve the

mapping from $\mathcal{A}_t$ to $\mathcal{A}_s$. Also, we have to slightly modify our definition of the set $offsets(\ell_k, i_j)$ and $count(\ell_k, i_j)$.

For an array reference $\mathcal{A}(i_1, i_2, \ldots, i_n)$ where $\forall i_{j,1\leq j<m\leq n}\ (span(i_j) = full \wedge speed(\ell_{copy}, i_j) = j)$ and $(span(I_m) = partial \vee speed(\ell_{copy}, i_m) \neq m)$, the size of the contiguous blocks accessed by $\mathcal{A}$ is given by

$$size(\ell, \mathcal{A}) = \prod_{k=1}^{m} count(\ell, i_k)$$

This definition now involves the speed of the index expressions because we are now not only concerned with breaks in the contiguousness of what is written in $\mathcal{A}_s$, but also breaks in the contiguousness of the mapping from $\mathcal{A}_t$ to $\mathcal{A}_s$, which are caused by the speeds of the index expressions on the LHS and RHS being in a different order.

As before, the offsets of the contiguous blocks in $\mathcal{A}$ are given by the set of n-tuples, $offsets$, where

$$offsets(\ell, \mathcal{A}) = \left\{ \langle 1, 1, \ldots, 1, I_m, I_{m+1}, \ldots, I_n \rangle \;\middle|\; \begin{array}{c} l(\ell, i_m) \leq I_m \leq u(\ell, i_m), \\ l(\ell, i_{m+1}) \leq I_{m+1} \leq u(\ell, i_{m+1}), \\ \ldots, \\ l(\ell, i_n) \leq I_n \leq u(\ell, i_n) \end{array} \right\}$$

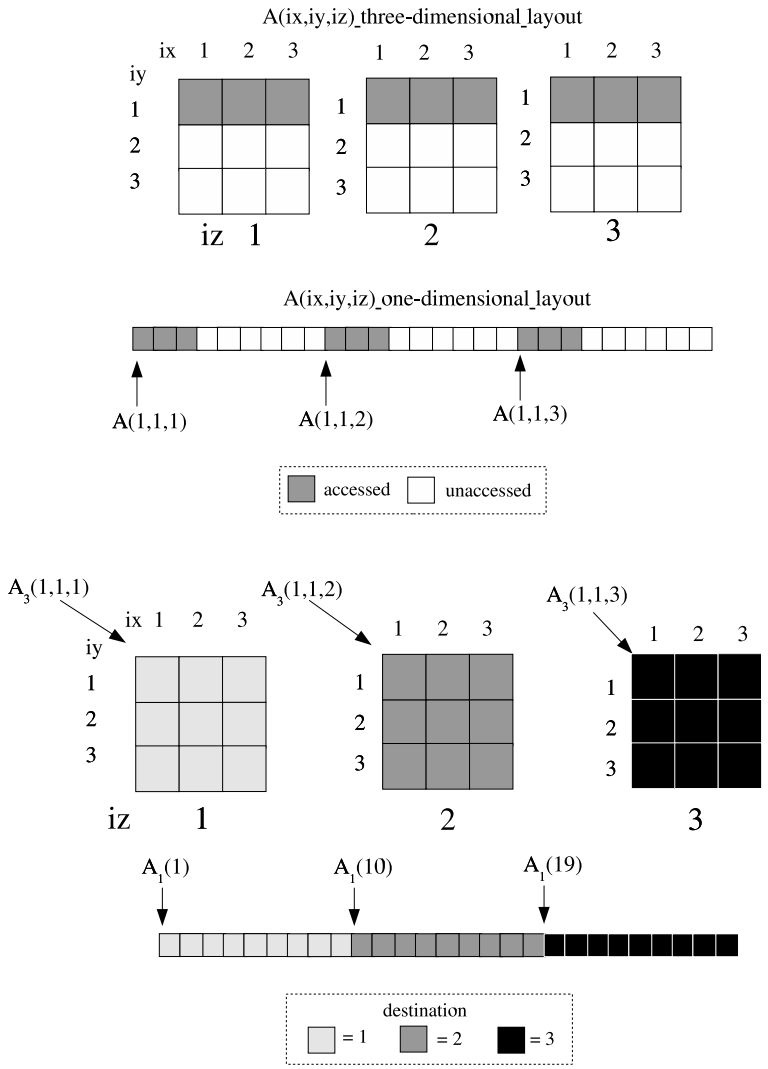
which translates into the following loop nest, ℓ_{comm}, which generates all $o \in offsets$:

```
tempOffset = 1
do I_m = l(ℓ_t, i_m),  u(ℓ_t, i_m)    !ℓ^m_comm
   do I_{m+1} = l(ℓ_t, i_{m+1}),  u(ℓ_t, i_{m+1})    !ℓ^{m+1}_comm
      ...
         do I_n = l(ℓ_t, i_n),  u(ℓ_t, i_n)    !ℓ^n_comm
            !⟨1, 1, ..., 1, I_m, I_{m+1}, ..., I_n⟩    !o ∈ offsets
            size=size(ℓ_t, A^f_s)
```

```
            to=destination(A_s^f, < I_1, I_2, ..., I_n >)
            call MPI_ISEND(A_t^f(offset),size,to,...)
            if(to == myRank) then
               do from = 0, NP-1
                  call MPI_IRECV(A_r(I_1, I_2, ..., 1 + from*Δ_n/NP + I_n%Δ_n/NP)),size,from,...)
               enddo
            endif
            tempOffset=tempOffset+size(ℓ_t, A_s^f)
         enddo
      ...
   enddo
enddo
```

Note that inside the loop nest ℓ_{comm} is the code to send the blocks from $\mathcal{A}_t$ to $\mathcal{A}_r$. In this case, the mechanics of the communication code is slightly different from that presented in the last section. Specifically, the offset of $\mathcal{A}_t$, which is being sent is controlled by a variable that is not a loop induction variable, but rather is a variable that is incremented in the body of the loop nest.

Because the blocks must be sent in the order that they are written to preserve the mapping from $\mathcal{A}_t$ to $\mathcal{A}_r$, we sort the loops, ℓ^k_{comm}, where $m \leq k \leq n$, comprising the loop nest ℓ_{comm} so that the innermost loop iterates on the "fastest" index expression and the outermost loop iterates on the "slowest" index expression of $\mathcal{A}^f_s$. Formally, we replace the superscript of each ℓ^k_{comm} with $speed(i_k)$, so that ℓ^k_{comm} becomes $\ell^{speed(i_k)}_{comm}$. Then, we sort the loops in descending order from outermost to innermost by the superscript of the loop. For an example of sorting loops, see Figure 3.14.

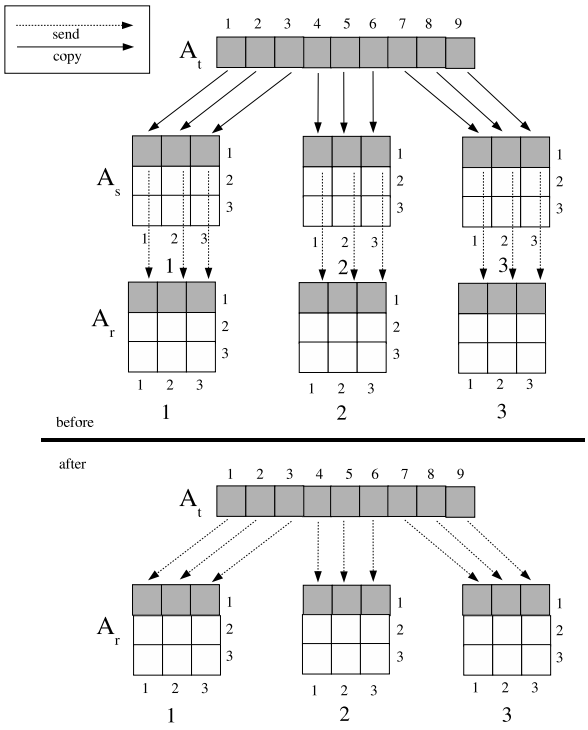



```
integer A3(1:3,1:3,1:3)
...
call MPI_ALLTOALL(A3,...)
```

Figure 3.11: Example: Simulating MPI_ALLTOALL

```
integer requests(1:2 × numSends)
...
do ix = 1, NX  !ℓ_t
   ...
   !every K iterations, or the last iteration
   if (range(ℓ_t) mod k == 0 ∨ ix == NX) then
      k = K
      !if k unevenly divides ix, which means this is the last iteration
      if ((ix mod k) ≠ 0) then
         !set k so that it is equal to the number of leftover iterations
         k = ix mod k
      endif
      if(ix > K) !after first tile
         call MPI_WAITALL(numSends,requests(1),stat,err)
      endif
      do I_m = l(ℓ_t, i_m),  u(ℓ_t, i_m)   !ℓ_comm
         do I_{m+1} = l(ℓ_t, i_{m+1}),  u(ℓ_t, i_{m+1})
            ...
               do I_n = l(ℓ_t, i_n),  u(ℓ_t, i_n)
                  !⟨1, 1, ..., 1, I_m, I_{m+1}, ..., I_n⟩   !o ∈ offsets
                  size=size(ℓ_t, A_s^f)
                  to=destination(A_s^f, < I_1, I_2, ..., I_n >)
                  call MPI_ISEND(A_s^f(I_1, I_2, ..., I_n),size,...,to,...,requests(reqNum))
                  reqNum = reqNum + 1
                  if(to == myRank) then
                     do from = 0, NP-1
                        call MPI_IRECV(A_r(I_1, I_2, ..., 1 + from Δ_n/NP + I_n % Δ_n/NP,size,...,from,...,requests(reqNum))
                        reqNum = reqNum + 1
                     enddo
                  endif
               enddo
            ...
         enddo
      enddo
   endif
enddo
call MPI_WAITALL(numSends,requests(1),stat,err)
```

Figure 3.12: Communication Code




```
!  BEFORE
do iy = 1, N  !ℓ
   call P(..., A_t)
   do ix = 1, 9  !ℓ_copy
      A_s(ix mod 3,1,ix/3) = A_t(ix)
   enddo
enddo

!  AFTER
do iy = 1, N  !ℓ
   call P(..., A_t)
   do ix = 1, 9  !ℓ_copy
      !  removed A_s(ix mod 3,1,ix/3) = A_t(ix)
      size=1
      to=destination(A_s,⟨ix mod 3,1,ix/3⟩)
      call MPI_ISEND(A_t(ix),size,to,...)
      if(to == myRank) then
         do from = 0, NP-1
            call MPI_IRECV(A_r(ix mod 3,1,ix/3),size,from,...)
         enddo
      endif
   enddo
enddo
```

Figure 3.13: Example: Removing Redundant Array Copies

```
!After replacing the superscript of ℓ^k_comm with speed(ℓ_copy, i_k)
do x = ... !ℓ^1_comm
   do y = ... !ℓ^3_comm
      do z = ... !ℓ^2_comm
         !⟨1, 1, ..., x, y, z⟩   !o ∈ offsets
         ...
      enddo
   enddo
enddo

!After sorting loops by superscript
do y = ... !ℓ^3_comm
   do z = ... !ℓ^2_comm
      do x = ... !ℓ^1_comm
         !⟨1, 1, ..., x, y, z⟩   !o ∈ offsets
         ...
      enddo
   enddo
enddo
```

Figure 3.14: Example: Sorting Loops when $\mathcal{A}_s^f = \mathcal{A}_s(\ldots, x, y, z)$ and $speed(x) = 1$, $speed(y) = 3$, and $speed(z) = 2$

Chapter 4

IMPLEMENTATION AND EVALUATION

4.1 Implementation

The approach presented in the last chapter for automating the pre-push transformation was implemented as a Fortran 90 source-to-source code transformer using the Nestor program transformation framework [28]. The transformer is named **The Compuniformer**, which is the the result of overlapping "communication" and "computation," reflecting the action of the transformer.

A source-to-source code transformer takes unoptimized source code as input, transforms it, and outputs optimized source code which then can be compiled to object code using the machine's own platform-specific compiler. By using a source-to-source transformer, we decouple our transformation from the specifics of any particular compiler designed for a particular architecture. Therefore, our optimization can be complemented with the traditional optimizations provided by the compiler.

The decision to take Fortran 90 code as input was made because we believe that the scientific and engineering community is shifting towards widespread usage of Fortran 90, as opposed to Fortran 77. There are many existing tools for analyzing and transforming Fortran 77, but very few for Fortran 90. Nestor is a transformation framework that can adequately handle Fortran 90 code.

The Nestor framework was developed by Georges Andre Silber to be a lightweight, easy-to-use, framework for implementing transformations to Fortran 90 code. Nestor

uses the Adaptor [8] open-source Fortran 90 parser to parse input, and provides an unparser, which translates the internal, hierarchical representation back to Fortran 90 code which then can be compiled to object code. The internal representation is a highly object-oriented AST-like representation of the input program. Nestor is implemented as a well-documented C++ library [29]. Transformations are implemented by accessing and modifying the objects representing the input program.

Nestor also has some built-in analysis tools, the most notable being its data dependence analysis which uses Petit [25] and the Omega Test [26]. To perform data dependence analysis, Nestor unparses the program as Petit code which is a simple Fortran-like language. Then the Petit compiler generates data dependence problems which the Omega Test solves, sending the results back to Nestor. The data dependence analysis capabilities are used as described in Section 3.4.

Although Nestor provides a basic transformation framework, it falls short in comparison to existing Fortran 77 tools in the breadth of the built-in analysis that it provides. The inlining, expression propagation and symbolic expression comparison that are needed for automating our transformation are not provided by Nestor. A Fortran 77 automatic parallelizing compiler framework called Polaris [4] developed by David Padua's group performs the required analysis, but cannot take Fortran 90 code. Polaris also provides more sophisticated array access analysis which could be useful in future versions of the optimizer. At this time, the portions of the implementation that require Polaris-like analysis are considered "semi-automatic," in that they require some user input.

4.2 Evaluation

We have performed a preliminary evaluation of our prototype implementation aimed at testing the correctness of the transformation. In doing so, we not only have the opportunity to verify the correctness of the implementation, but also the techniques that underly it. To evaluate the Compuniformer, we wrote a test program

which is simple, yet tests many of the features of the transformation process. The test code exhibits the Interprocedural Indirect computation pattern, which leaves us with a redundant array copy to remove. Additionally, the array access pattern of the send array creates some breaks in contiguousness, which complicates the communication loop generation.

Performing this evaluation has revealed some of Nestor's shortcomings which need to be overcome. One problem with Nestor is that it does not support Fortran 90 keyword `uses`, which means that codes with subroutines contained in external modules cannot be used at this time. We have transformed the test code manually before running it through the Compuniformer to circumvent this problem. Possible solutions to this problem include fixing Nestor directly, or writing a script that transforms codes into an acceptable form. A less serious problem due to Nestor is that its output sometimes contains lines that are over 120 characters, which exceeds the limit imposed by the Fortran 90 compiler. We have fixed this problem by writing a script that breaks long lines in the output of the transformer.

Aside from the necessary interventions described above to overcome the limitations of Nestor, the Compuniformer generates correct code, as evidenced by the transformed code producing the same output as the original code. The transformed code contains many large, hard-to-read expressions that result from building expressions based on the array access analysis. Unoptimized, these large expressions could slow down execution significantly. However, using constant propagation and subexpression elimination, the Fortran 90 compiler will reduce the size of the expressions to what they would be if written by a human programmer.

At this time, the Compuniformer does not implement all of the techniques developed in this thesis. Most of the work has been completed to enable transforming Intraprocedural Direct computation loops, and the selection of the tile loop and the value of K with Interprocedural Indirect computation loops, but these features

remain unfinished at the moment. Additionally, the Compuniformer does not check that all array index expressions are incremented with a step of one in the forward direction, which is an assumption of the transformation process. In the future, the Compuniformer will verify what it currently assumes about input.

Chapter 5

RELATED WORK

There have been several research efforts designed to mitigate messaging overheads. These efforts include Active Messages [30], U-Net [31], Fast Messages [23], and the standard produced by Microsoft, Compaq and Intel known as the Virtual Interface Architecture (VIA) [12]. 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 [27].

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* [34, 35] is an optimization technique that has received significant attention.

The project most related to our study that uses compiled communication is CC-MPI [18]. The main difference from our work is that CC-MPI is designed for Ethernet-switched clusters, whereas our study focuses on a program transformation for RDMA-enabled networks [6, 17, 24].

Several other projects have suggested program transformations to minimize communication overheads in parallel applications. The PARADIGM compiler [3] as well as Goumas et al. in [14] 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 into 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 [19] and [16]. 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, before they will be needed. This difference can have important implications, since one-sided `get` operations are not common, where one-sided `put` operations exist in all RDMA-capable networks.

Finally, global-address space (GAS) languages [13, 33] perform communication overlapping optimizations, but none of them handle input code written in *MPI*.

Chapter 6

CONCLUSIONS AND FUTURE WORK

This thesis described the novel techniques developed to automate the transformation of explicitly parallel code to pre-push data generated during computation. The result of this transformation is to reduce communication latency by providing opportunities for communication-computation overlap. Our transformation system benefits the large community of domain scientists that use MPI and Fortran 90 to implement their parallel algorithms and RDMA-enabled network clusters. Previous work by other researchers developed techniques for reducing communication latency, but none were applicable to explicitly parallel codes written using MPI, which pose extra challenges that were overcome in the development of this system. Some of the major contributions of this thesis include:

- Defining the pre-push transformation in a general way which paves the way for future work, and selecting a subset of this problem space that represents a large subset of real-world codes, yet is feasible to process at this pioneering stage of research.

- Identifying the different cases and situations that arise in a variety of parallel MPI codes, and addressing these in the design of the overall transformation process.

- Ensuring that the transformation is semantics-preserving; for instance, preserving the semantics of the original call to `MPI_ALLTOALL` by mapping contiguously written blocks of elements to the appropriate destination rank. Also,

using data dependence and array access analysis to generate correct, efficient communication.

- Removing redundant array copies by utilizing the transitivity of the copy/communication.
- Designing and implementing a prototype source-to-source optimizer which provides a proof-of-concept as well as a basis for further development into a production-quality system.

Significant work has been completed in this thesis; however, many directions remain in which this research can be extended. Some suggestions for future work include:

- Creating heuristics to deal with some of the cases arising in real-world codes that are not addressed by the general formulation of the techniques in this thesis.
- Continuing work with the prototype optimizer to implement more of the theory developed in this thesis, making the semi-automatic steps fully automatic.
- Evaluating the performance of the system on a variety of real-world codes, which should inform future work on extending the system's generality.
- Extending the system to target other types of collective communication such as scatter and broadcast.

In conclusion, the broader impact of this work is the performance improvement of parallel MPI codes on networked clusters, enabling more scalable application of the parallel codes to larger numbers of processors.

BIBLIOGRAPHY

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

[2] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 41–53, New York, NY, USA, 1989. ACM Press.

[3] 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.

[4] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The Next Generation in Parallelizing Compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[5] William Blume and Rudolf Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, April 1995.

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

[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] Thomas Brandes. Adaptor, high-performance fortran compilation system. `http://www.scai.fraunhofer.de/291.0.html?&L=1`.

[9] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to Parallel Codes for Communication-Computation Overlap. *Technical Report 2005-12, University of Delaware, Department of Computer and Information Sciences*, 2005.

[10] Frederic Desprez, Jack Dongarra, and Bernard Tourancheau. Performance study of LU factorization with low communication overhead on multiprocessors. *Parallel Processing Letters*, 5:157–169, 1995.

[11] 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.

[12] 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.

[13] T. El-Ghazawi, W. Carlson, and J. Draper. UPC specification. http://upc.gwu.edu/documentation.html, 2003.

[14] 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.

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

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

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

[18] Amit Karwande, Xin Yuan, and David 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.

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

[20] Kostas Magoutis, Margo I. Seltzer, and Eran 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.

[21] MPI Forum. MPI: A message-passing interface standard, v1.1. Technical report, University of Tennessee, Knoxville, June 12, 1995.

[22] Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *ACM Trans. Program. Lang. Syst.*, 24(1):65–109, 2002.

[23] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995.

[24] 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.

[25] William Pugh. Release 1.10 of petit. `http://www.cs.umd.edu/projects/omega/`.

[26] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM Press, 1991.

[27] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-performance local-area communication with fast sockets. In *Proceedings of Usenix Annual Technical Conference*, pages 257–274, 1997.

[28] Georges-André Silber and Alain Darte. The Nestor library: A tool for implementing Fortran source to source transformations. In *High Performance Computing and Networking (HPCN'99)*, volume 1593 of *Lecture Notes in Computer Science*, pages 653–662. Springer Verlag, April 1999.

[29] Georges-André Silber, Alain Darte, and Guillaume Huard. Nestor programmers guide. `http://www.cri.ensmp.fr/people/silber/nestor/doc/index.html`.

[30] 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.

[31] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner 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.

[32] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[33] 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. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

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

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