# A Compiler-based Communication Analysis Approach for Multiprocessor Systems*

Shuyi Shao[1], Alex K. Jones[2], Rami Melhem[1]

[1]University of Pittsburgh
Department of CS
Pittsburgh, PA 15260 USA
{syshao, melhem}@cs.pitt.edu

[2]University of Pittsburgh
Department of ECE
Pittsburgh, PA 15260 USA
akjones@ece.pitt.edu

## Abstract

*In this paper we describe a compiler framework which can identify communication patterns for MPI-based parallel applications. This has the potential of providing significant performance benefits when connections can be established in the network prior to the actual communication operation. Our compiler uses a flexible and powerful communication pattern representation scheme that can capture the property of communication patterns and allows manipulations of these patterns. In this way, communication phases can be detected and logically separated within the application. Additionally, we extend the classification of static and dynamic communication patterns and operations to include persistent communications. Persistent communications appear dynamically, however, they remain unchanged for large segments of the application execution. Our compiler is capable of detecting both static and persistent communication patterns within an application. We show that for the NAS Parallel Benchmarks, 100% of the point-to-point communications can be classified as either static or persistent and, with the exception of IS, 100% of the collective were either static or persistent. By comparison to application trace data, the predicted LBMHD, CG and MG communication patterns have been verified.*

## 1. Introduction

Understanding the communication behavior of parallel applications is becoming increasingly important in high performance computing research. For instance, circuit switching techniques have been proposed and used as network paradigms in massively parallel processing machines [7]. However, the overhead incurred in these techniques due to establishing connections limits the performance enhancement potential. The knowledge of the communication patterns can be used to reduce this overhead. An extreme example is Flat Neighborhood Networks [5], which can derive a much better design from the specification of particular communication patterns. Even for packet switching and wormhole routing techniques, the knowledge of communication patterns also provides significant guidance to the communication system designers [9].

The motivation of this work stems from the proposal to include an optical circuit switching (OCS) network in the design of next generation high performance computing systems [2]. In that proposal long-lived bulk data transfers are routed through all optical switches, which are characterized by high data rates with high overhead for circuit establishment. An OCS is less expensive than its electronic counterpart as it uses fewer optical transceivers. This interconnection technique is more effective if connections are pre-established and the relatively long switch times are amortized over the lifetime of the connections. Our compiler provides the analysis that makes this possible. Shalf et. al. proposes another interconnection network that use both passive(circuit switching) and active(packet switching) switches to deliver performance as a fully-interconnected network [14]. In their approach, the switches must be reconfigured to emulate a suitable interconnection topology to achieve best performance for the running application. The effectiveness of this topology optimization process heavily depends on how quickly the communication pattern of the application can be obtained. Our compiler can infer the communication topology for such an approach at compile-time to more effectively utilize this approach.

We target tackling MPI-based [13] parallel applications, which are supported by most supercomputer systems. Based on the temporal and spatial properties of

*: This work is in the context of the PERCS project at IBM, which is supported in part by the Defence Advanced Research Projects Agency(DARPA) under Contract No. NBCH3039004.

1-4244-0054-6/06/$20.00 

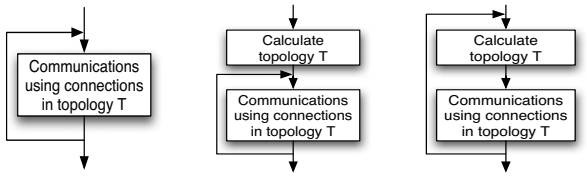
communications and the capability of the compiler to identify the temporal properties and the detailed topology – the spatial properties – of the communications, we classify communications during a phase of an application's execution into three categories: *static*, *persistent* and *dynamic*. In this context, the topology of communication is the specification of the source and destination of the messages exchanged.

**Static** – Communication is static if it can be completely determined through compile-time analysis. That is the compiler can identify both the temporal locality and the exact topology of the communication.

**Persistent** – Communication is persistent if, though the compiler cannot determine the exact topology of communication, it can determine that the topology does not change during the phase. That is, its temporal locality can be identified by the compiler, but its spatial properties remains unknown until run-time.

**Dynamic** – Communication is dynamic if it cannot be determined until run-time when the communication operations actually occur and they change within a phase.

Given that communication requirements of applications may change during different phases of execution, an important part of the analysis of communication patterns is to identify and segregate different communication phases. We observe that a main source of communication temporal locality originates from loop structures of MPI programs. Hence, it is natural to consider a loop, which contains communications as the building blocks of phases.




(a) Static: $T$ is resolvable at compile-time.

(b) Persistent: $T$ is calculated before a phase.

(c) Dynamic: $T$ changes inside a phase.

**Figure 1. Static, persistent and dynamic communications when the phases is a loop.**

In Figure 1, we illustrate the definition of static, persistent and dynamic communications when a phase is defined as a loop. Specifically, the communication operations are static if the topology (in terms of sources and destinations) can be completely resolved at compile-time, as in Figure 1(a). In Figure 1(b), the topology can not be determined until run-time. However, once defined, the topology is repeatedly used within the loop. In this case we call the communication operations *persistent*. In Figure 1(c), the communications are *dynamic* because during each iteration of the loop, the topology is re-calculated prior to use.

The above classification implies different possibilities for reducing communication overhead. For example, considering circuit switching networks with preloading capability such as an OCS network, the earliest opportunity for determining network configurations for a static communication operation is at compile-time. Thus configurations may be statically inserted into the code by the compiler at the phase boundaries. For persistent communication, the earliest opportunity to know the necessary topology of the network is at run-time. However, it is possible for the compiler to insert a symbolic expressions specifying the topology at compile time that may be resolved at runtime. By placing these symbolic expressions at the earliest point where the expression will be resolved the network reconfiguration may still be able to take place prior to use within a phase.

In this paper, we present a compiler-based communication analysis approach for identifying static and persistent communication patterns. Many previous efforts to analyze parallel applications' communications characteristics are based solely on trace analysis. However, the traces can provide the communication information for only a single execution instance of an application on a particular platform. Our goal is to reveal the underlying communication patterns available at compile-time.

We use a powerful scheme to represent communication patterns which includes both collective and point-to-point communications in terms of communication vectors and matrices. The vectors and matrices contain exact values if the communication pattern contains only static communications. Otherwise, they may contain symbolic expressions for later resolution. This scheme allows the manipulation of communication patterns through a set of convenient operations. It is also flexible and can be easily tailored to other types of communication analysis.

This paper is organized as follows. Section 2 presents some motivation and related work. We describe the stages of the compiler in Section 3. The communication pattern representation methodology is detailed in Section 4. Section 5 relates experimental results from our compiler on a set of parallel benchmarks. Conclusions are related in Section 6.

## 2. Related Work

The recent trends in network switching techniques have shown significant improvements in network bandwidth and a reduction in communication latency. These trends are often driven by improvements in implementation technology. For example, circuit switching hardware continues to improve due to improvements in the technology. Newer technologies such as optical networking continues to be an alternative to electronic circuit switching that provides several advantages such as capabilities to handle long wire lengths and achieve high bandwidths. However, the reconfiguration time of optical switching may be relatively long (ms vs $\mu$s). Hence, to implement circuit switching with relatively long switching latency mandates a technique to amortize connection establishment overhead.

Our previous work [6] shows that much of the circuit switching overhead can be amortized by pre-establishing connections and re-using connections as much as possible. In fact, the network configuration pre-loading scheme has been shown to perform better than traditional wormhole and non-predictive circuit switching techniques in many instances. However, this solution requires that the communication pattern must be known early enough, ideally at compile-time.

There have been several attempts [1, 8, 10, 15, 16] to understand the communication characteristics of parallel applications. These efforts focus on analyzing the communication characteristics of parallel applications such as the ratio of different kinds of communications. They do not provide accurate descriptions of communication patterns with respect to connections. Some researchers do try to identify the logical communication topologies, such as the 2-D meshes or hypercube.

Many interesting research projects require information about communication patterns. For example, Cappello and Germain proposed an approach to associate compiled communications and a circuit switched interconnection network [3]. Yuan et. al. explored using compiled communication as an alternative to dynamic network control [18]. The compiled communication technique requires that a large portion of static communications be identified at compile-time. Dietz and Mattox studied the Flat Neighborhood Network (FNN) which uses the communication patterns to determine the design of the network [5]. Liang et. al. described a compiler, which supports compile-time scheduled communication for their adaptive System-On-a-Chip (aSOC) communication architecture [12]. As previously described, our previous work introduces a switch design which can use our compilation technique to pre-program a TDM network switch [6]. All of the above efforts need the precise knowledge of communication patterns, which is the goal of our compiler. Each of these techniques can take advantage of compile-time knowledge of the communication pattern to reduce overhead in the network.

## 3. Compilation Framework

Analyzing communication patterns from the source code of a MPI application is challenging. In this section, we describe the compilation framework in detail.

### 3.1. Analysis Techniques

Although there still exist many technical challenges, compiler techniques today are capable of inferring information which is essential for analyzing the communication behavior of a whole MPI parallel program. In order to accomplish this, we must perform fairly sophisticated compiler transformations such as inter-procedural analysis on the source code. For example, the value of constant function parameters must be propagated during the analysis of callee functions to help determine whether communications are static. Also, many MPI functions utilize array parameters in MPI functions. This requires the compiler to employ array analysis techniques.

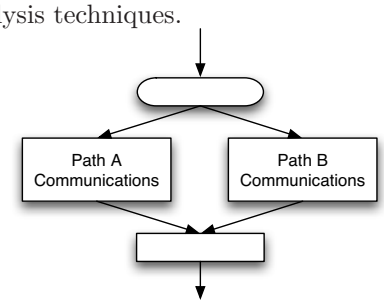



**Figure 2. Decision point.**

MPI programs are written in SPMD style. Each processor independently executes the same program on its private data. Nevertheless, often different processors take different execution paths. The graph in Figure 2 represents a decision point (DP) which introduces multiple, here 2, optional branches. Each processor will independently evaluate the conditions at the beginning of the DP and make its decision to take either Path A or Path B. Although we call it a decision "point", we will use this terminology to refer to all the instructions and communications between the condition instruction point and the merge point.

We attempt to resolve information from the leading conditions of a decision point about which set of processors are allowed to take a particular execution branch of the decision point. The information is crucial for identifying communication patterns. For example, the eligible source processors of an MPI send function can be determined if the information can be inferred at compile-time (e.g. the conditional depends on the processor id). The destination processors can be determined by analyzing the destination parameter in the MPI function. Therefore, the topology of this send operation may be completely resolved. Also, if the leading condition of a decision point is an equality or inequality test comparing to a constant, we propagate the value while analyzing the selected branch.

Because our focus is on inferring information about communication, we can safely prune the control flow graph (CFG) branches that do not contain communication operations and do not perform operations which modify information involved in analyzing communications. Additionally, though compiler analysis techniques are fundamental to accomplishing these inferrances, most of the techniques that we utilize are classical and well studied in the literature. Thus, we will not describe the details of their implementations. Rather, we focus on how these techniques are applied for analyzing communication patterns and treat compiler analysis techniques just as tools for providing information.

### 3.2. Compilation Framework Paradigm

The compilation framework can identify communication patterns and enhance applications with network

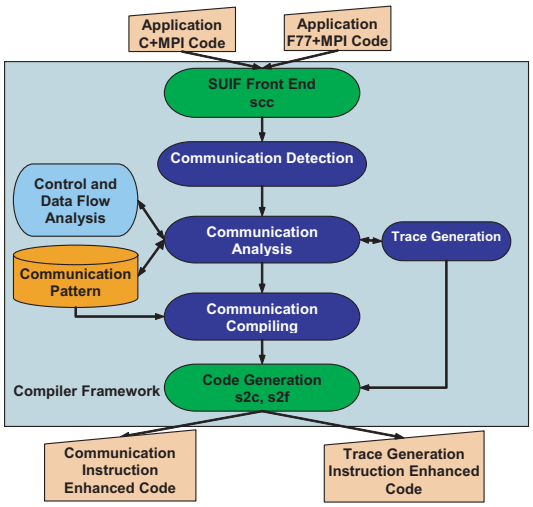



**Figure 3. The compilation framework.**

configuration instructions and trace generation instructions. A compiler prototype has been developed based on the SUIF compiler [17]. The SUIF compiler is an open source, source to source compiler for both the C and Fortran 77 languages.

Figure 3 shows the paradigm of our compilation framework. The front end of the SUIF compiler compiles parallel applications to the SUIF intermediate format. A tool, porky, from the SUIF compiler system is used to perform basic transformations, such as copy propagation and constant propagation in order to discover more details about the characteristics of different communication(e.g. static, dynamic, or persistent).

We do not intend to demonstrate a compiler which can schedule communications for specific interconnection networks. Instead, we focus on identifying communication patterns at compile-time. In other words we focus on the communication detection and communication analysis components of the framework.

### 3.2.1. Communication Detection Component

The communication detection component is responsible for detecting all the MPI communication calls from the SUIF intermediate code. Because MPI is a standard, we can explicitly enumerate all possible MPI operations. Based on this list, we can easily identify, within the compiler, all the MPI operations that occur in the code. This component collects information about these communication operations' parameters. For instance, the communication detection component identifies that `me` is the variable holding the rank value and `nprocs` is the variable holding the total number of nodes from the following source code.

```
call mpi_comm_rank(mpi_comm_world, me, ierr)
call mpi_comm_size(mpi_comm_world, nprocs, ierr)
```

### 3.2.2. The Communication Analysis Component

The MPI operations and parameter information identified during communication detection is used during communication analysis to identify and represent the communication pattern. First, the a complete control flow graph (CFG) is constructed and communication information is propagated through the graph. Then the communication analysis component builds a communication sequence, which can be represented either as a communication-intensive simplified version of the CFG (also called a communication graph) or a literal expression which contains similar information. The communication sequence is built from the CFG and has links pointing back to the original CFG. This process is described in more detail in Section 4.1. Then the the communication pattern is identified from the communication sequence and represented using a compact internal format. The communication behavior of the application is broken down into *communication phases* using techniques described in Section 4.3. By mapping the communication patterns into a series of phases it is possible to create smaller, more efficient *communication working sets* or groups of communications that occur in relatively close proximity. By improving the coupling of the phase to the underlying structure of the application it is possible to more efficiently utilize our network resources.

### 3.2.3. Communication Compiling Component

The communication compiling component compiles the communication pattern identified during analysis and inserts network configuration instructions into the application to assist configuring the network. The communication compiling component is designed for connection-oriented networks with the goal of reducing the communication overhead. For example, our compiler can insert instructions to preload network configurations for the switches proposed in [6]. Two types of network configuration instructions are considered in our current experimental compiler.

**Network configuration setup instructions** are used to pre-establish network connections. It can reduce the setup overhead of connections and overlap network control with computation.

**Network configuration flush instructions** are used to flush the current network configuration or a subset of circuits from the current configuration. Such instructions can be used to remove the expired circuits. It also provides potential to speedup the parallel applications as it reduces contention for the circuits.

For static communication patterns, the content of the communication pattern derived during analysis is inserted into the code using network configuration setup instructions designed to pre-configure the network at the beginning of each communication phase. For persistent communication patterns, the communication analysis component provides communication pattern information as a function of variables that will not be known until run-time. Code for calculating these symbolic expressions at the appropriate locations after all the required parameters are known is generated and inserted into the code. The scope and value availabilities of variables in the symbolic expressions,

in addition to the location of the communication operations, put constraints on the locations where these instructions may be inserted. The communication compiling component can also insert network configuration flush instructions at the end of a communication phase upon the determination that the next phase will use different set of connections.

#### 3.2.4. Trace Generation Component

Many efforts to study the communication patterns of parallel applications depend on study of application traces. Unfortunately, this technique has the drawback of seeing the execution based on a specific set of data and parameters. While our goal is the determination of the communication patterns of applications through compiler analysis, our compiler also has the capability to add trace generation instructions (e.g. print statements) within the application. These traces are primarily used to verify that the communication patterns detected by the compiler are accurate.

## 4. Pattern Representation

Several research groups have observed that the communication operations in many applications exhibit regular patterns [2, 10–12]. Additionally, it has been shown that these regular communication patterns can often be discovered through analysis of the source code [3, 4, 8]. Our compilation framework described in Section 3 and the subsequent communication analysis described in the following sections is motivated by these two discoveries.

To effectively perform compile-time communication analysis it is necessary to represent the communication patterns identified from the code in a form that is both concise and accurate. In the following sections we describe a communication sequence and a communication matrix/vector pair that are used to represent the communication patterns in the application. Additionally, we describe techniques for detecting communication phases within the application and how the communication pattern representations can be manipulated to closely correspond to the underlying communication network.

### 4.1. Communication Sequence

When studying communication patterns, we are primarily interested in the communication operations and control flow related code structures. Since the content of the CFG of an entire application includes much information unrelated to the communication behavior, our experimental compiler uses a *communication sequence* which prunes the branches that have no impact on analyzing communications. The communication sequence retains only the communications and communication-related control flow. It is constructed from the original CFG and contains links back to original CFG.

The communication sequence delineates the innate communication characteristics of the application. More formally, we use a concise literal representation to describe the communication in a program. The grammar in Figure 4 produces simple literal strings for describing communication sequences.

```
Comm_Sequence → ϵ | Comm_element Comm_Sequence
Comm_element → Col | P2P | Comm_Loop | Comm_DP
Col → AA | AV | BC | BR | RD | AR | ...
Comm_Loop → [ Comm_Sequence ]
Comm_DP → { Comm_Sequence || Comm_Sequence }
```

**Figure 4. The grammar for communication sequence literal representation.**

The symbols on the left side are non-terminals. The collective communications, or `Col`s are as follows: `AA` represents MPI_Alltoall, `AV` represents MPI_Alltoallv, `BC` represents MPI_Bcast, `RD` represents MPI_Reduce, `AR` represents MPI_Allreduce, `BR` represents MPI_Barrier, etc. `P2P` represents a point-to-point communication. `Comm_Loop` represents a loop that contains communication operations expressed by square brackets. `Comm_DP` indicates a decision point expressed by curly brackets. There are two types of `Comm_DP`, *if-then* or *if-then-else*.

**Example 1:** The communication sequence literal representation of the IS (integer sorting) program from NAS parallel benchmark suite [1] is shown below:

```
AR AA AV [ AR AA AV ] RD { P2P || } RD
```

There are three collective communications to start the application followed by a loop with three collective communications in the body. There is a reduce operation after the loop completes immediately followed by a conditional represented by a decision point. For this example using the representation from Figure 2, path A contains a `P2P` communication and path B contains no communication. The last operation is a reduce.

The communication sequence and its literal representation is constructed by the communication analysis component of our compilation framework. While the literal representation does not provide details of the point-to-point or collective communication operations, it does provide a fairly concise and surprisingly detailed map of the application's communication characteristics. It can be used to quickly familiarize developers with communications occurring in the application.

### 4.2. Phases and Pattern Representation

Often, a parallel program is written to solve a particular scientific problem. These applications are often organized in phases and although different processors may take different paths, they tend to work in the same computational phase at approximately the same time but primarily on their local data. Given that these parallel applications have computational phases, we can expect their communication operations to behave similarly. For example, the communication topologies of adaptive applications evolve during their execution time. Even for parallel applications that have static

communication patterns, their active communication working set may change as the phases change. The result is one or more *communication phases*. Communication phases are not identical to computational phases, but are strongly associated with them. For example, some computational phases contain no communication and thus can be ignored when identifying communication patterns. Several computational phases may just yield a single communication phase. The number of phases is an artifact of the analysis used to partition the *communication sequence* into *phases*.

**Example 2:** We can partition the communication sequence in example 1 into phases in two ways shown in Table 1. The first one partitions the sequence at the loop boundaries and three phases are identified. Actually the communications in the first two phases are identical. Hence we can merge them and obtain the two-phase partition.

**Table 1. Communication phases in IS.**

| *Phases sub − sequence* | *Phases sub − sequence* |
|---|---|
| *phase* 0 `AR AA AV` | *phase* 0 `AR AA AV [AR AA AV]` |
| *phase* 1 `[AR AA AV]` | *phase* 1 `RD { P2P || } RD` |
| *phase* 2 `RD { P2P || } RD` | |

The fact that the communication pattern of an application contains phases is important and must be described while representing the pattern. However, the traditional technique to represent the communication pattern of an application is to describe its rough logical topology, (e.g. 2-D mesh, hypercube, etc.), or to provide a communication matrix. Such representations are too coarse and can not describe the communication topologies accurately. They also fail to disclose temporal information. Our communication pattern representation scheme is designed specifically to avoid this limitation and to effectively represent the temporal and spatial properties of the pattern.

We define all the communication operations of an application as a *communication pattern*. When the execution is partitioned into phases, the communications within each phase need to be specified. There are two types of communications in MPI applications: collective communications and point-to-point communications. In order to represent the collective communications in a pattern, we define a *c-enumeration* to describe the set of collective communication functions invoked in a parallel application.

**Definition** A ***c-enumeration*** is a list of all the collective communications that appear in a parallel application. Each collective communication is represented by a pair, the function name and optionally the corresponding communicator. The communicator is omitted if it is the default MPI communicator. For each related MPI communicator, the same collective MPI functions have exactly one instance in the *c-enumeration*. Each communication pattern retains a unique *c-enumeration*.

**Example 3:** `CEE = {AA, AR , (AR, commu1), RD}` indicates that there are three different types of collective communications in the application. The first two and the last operations are performed in the default MPI communicator while the third operation, MPI_Allreduce, is performed in a user-defined communicator `commu1`. The communication detection component of the compilation framework is responsible for building *c-enumerations*.

A communication pattern consists of a sequence of phases. A basic *communication phase* is described by a $c-vector$ and a $p-matrix$ that represent all the collective and point-to-point communications, respectively. A phase may also be a loop of a sequence of phases that repeat in any execution instance of an application.

**Definition** A $c-vector$ corresponds to a *c-enumeration*. Each element of the vector represents the weight of the corresponding collective communication in the *c-enumeration*.

**Definition** A $p-matrix$ is a communication matrix that describes a set of point-to-point communications. Each entry of a $p-matrix$ represents the weight of the corresponding point-to-point communication.

**Definition** $\delta$ represents any unknown values, variable, vector, or matrix.

A $p-matrix$ is **deterministic** if the total number of processors $N$ is known at compile-time and each entry of the $p-matrix$ is a constant. A deterministic $p-matrix$ is used to represent static communications. Persistent communications can always be described by formula lists. They can also be described by a $N \times N$ matrix with symbolic entries if $N$ is known at compile-time, referred as a **symbolic** $p-matrix$.

**Example 4:** Figure 5 shows the deterministic $p-matrix$ in phase 2 of the communication pattern of IS in Table 2.

$$\begin{pmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{pmatrix}$$

**Figure 5.** $p-matrix$ **PM_IS(with 8 processors).**

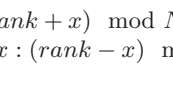
**Table 2. The communication pattern of IS.**

| *c−enumeration* | {AR,AA,AV,RD} | |
|---|---|---|
| Phases | *c−vector* | *p−matrix* |
| *phase* 0 | <1, 1, 1, 0> | NULL |
| *phase* 1 | <1, 1, 1, 0> | NULL |
| *phase* 2 | <0, 0, 0, 1> | PM_IS |

As shown in Figure 6, PM_A and PM_B are a formula list and a deterministic $p-matrix$, respectively. PM_A describes a communication pattern in which each processor *rank* sends to $rank + x$ and $rank - x$ if $rank - x > 0$ where $x$ is determined at run-time and $N$ is the total number of processors. In the case $x = 1$

$$\begin{pmatrix} (rank+x) \mod N \\ rank{>}x : (rank-x) \mod N \end{pmatrix}$$

(a) PM_A



(b) PM_B

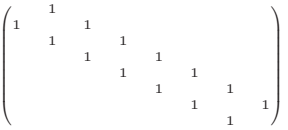
**Figure 6. A $p{-}matrix$ PM_A described by a formula list and the corresponding deterministic $p{-}matrix$ P-matrix_B.**

and $N = 4$, we infer a deterministic $p{-}matrix$ PM_B from PM_A.

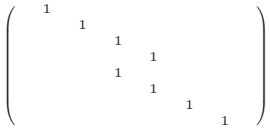
In the above definitions of $p{-}matrices$ and $c{-}vectors$ we do not enforce a specific meaning for the communication weight. However, some options include (1) a single bit value to indicate if point-to-point communications from the source processor to the destination processor exists (2) the message volume or (3) message count. In the case that the compiler cannot construct even a symbolic expression for a point-to-point communication the $\delta$ symbol is used in the symbolic expression for that matrix entry.

```
if (myrank < nprocs/2) {
  MPI_Send(buf,1, MPI_INT_TYPE,
    myrank+1,1000,MPI_COMM_WORLD);
} else {
  MPI_Send(buf,1,MPI_INT_TYPE,
    myrank-1,1000,MPI_COMM_WORLD);
}
```

(a) Decision point code.



(b) Conservative matrix.



(c) Matrix considering decision point.

**Figure 7. A phase and $p{-}matrices$.**

Decision Points are an important code structure when constructing $p{-}matrices$ because they can restrict the set of eligible processors to perform communication operations. Figure 7 shows an example of a decision point impacting the communication matrix. The code for the example is shown in Figure 7(a) where `myrank` holds the processor rank and `nprocs` holds the total number of processors. For this example, we show the case where `nprocs = 8`. If the decision point is ignored the resulting conservative $p-matrix$ is shown in Figure 7(b), which contains several non-existing connections. This technique assumes that each processor will take both the `then` and `else` branches and requires two connections for each processor. However, by static analysis it is possible to determine the exact paths taken by each processor. Figure 7(c) shows the optimized $p{-}matrix$ where each processor now only communicates with a single destination.

This communication pattern representation scheme can be used to represent both compile-time identified communication patterns and the communication patterns identified from execution traces. The main advantage of our pattern representation scheme is that it captures the time evolving, or *phased*, property of communication patterns. The concepts of phases suggests the need to manipulate the communication patterns from different phases and to schedule the communications in the pattern at different granularities according to parameters of the target system.

### 4.3. Phase Partitioning

One of the crucial tasks for identifying communication patterns is to effectively partition the program into communication phases. Application designers and developers usually have precise knowledge about different phases. However, requiring the developer to specify such information may reduce their productivity. Moreover, we do not want the user to worry about the underlying network parameters. For instance, if large number of connections can be established simultaneously, larger communication phases may be possible. Hence, we want the compiler to do the phase partitioning automatically.

We observed that loops and decision points are important code artifacts that have to be considered while partitioning communication phases. In our compiler, we use the following rules to partition phases:

- All the communications in an outermost decision point, which is also not enclosed in any loop, fall into the same communication phase.
- All the communications in an innermost loop unconstrained by any decision point form an initial communication phase.
- All the communications in the maximum interval between two immediately adjacent pairs of the above phases forms a communication phase.

Communication phase partitioning is performed by the communication analysis component in our compilation framework following the above rules. For each partitioned communication phase, the communication analysis component builds a global $c{-}vector$ and a global $p{-}matrix$.

Rather than the above partitioning rules we currently use, other potential strategies exist. For instance, it is also practical to partition phases mainly according to outermost loops and decision points. We can also start from identifying each communication operation from the source code and treating each of them as a tiny communication phase. Then we might merge these tiny communication phases to construct additionally coarse phases until the granularity of phases reaches a desired threshold. Our current design can be extended to allow different phase partitioning policies.

Besides supporting phase partitioning, information from control and data flow analysis, especially when the

destinations are re-calculated, is crucial to determine if the communications are static, persistent, or dynamic.

### 4.4. Manipulating Patterns

A communication phase can more precisely represent an active connection *working set*. One strategy to determine the communication phases in the program is to assume that each loop in the application is a phase and to manipulate these initial phase decisions to group the communications into new phases that are best suited to the capacity of the network. For instance, we may want to combine two adjacent phases if the network capacity is large enough to hold both of them; we may want to remove some infrequently used connections if the newly combined phase is slightly beyond the capacity of the network. We define four core operations required to deal with different communication phases:

**Merge** combines the $p-matrices$ and $c-vectors$ of two adjacent phases of a communication pattern into a new phase. If no communication weights are included, this is equivalent to an OR operation.

**Filter** removes connections below a threshold from a phase of a communication pattern.

**Unwrap** extracts the loop body from the control to facilitate other operations such as phase merging.

**Collective to point-to-point** (C2P) decomposes collective operations into a matrix of point-to-point operations. This allows explicit overlaps between the $p-matrix$ and $c-vector$ to be explored.

Because these representations do not include temporal information, it is not possible to sub-divide a communication phase into multiple phases. However, the analysis may revert back to the initial loop partitions to rebuild simpler phases. Additionally, other operations are omitted when they can be created from groups of existing transformations, (e.g. delete can be implemented by a combination of merge and filter).

## 5. Results

We have developed an experimental compiler that implements the compilation framework described in Section 3. Our compilation framework provides three key contributions beyond current approaches in the literature. In Section 5.1 we examine the impact of considering persistent communications in addition to previous approaches which consider only static and dynamic communications. This knowledge is leveraged in the compiler to determine the communication pattern for these benchmark applications in Section 5.2.

### 5.1. NAS Parallel Benchmark Results

We used our experimental compiler to profile the communication statistics of the NAS Parallel Benchmarks v2.4.1. The percentage of static, persistent, and dynamic communications are shown for point-to-point operations in Table 3 and for collective operations in Table 4. All of the data obtained for these charts were acquired through compile-time analysis alone with the exception of IS and FT's collective operations which were added through prior 128 node traces.

**Table 3. The percentages of different point-to-point communications in NAS benchmarks.**

| Benchmark | Static | Persistent | Dynamic |
|---|---|---|---|
| **IS** | 100% | 0% | 0% |
| **CG** | 100% | 0% | 0% |
| **MG** | 0% | 100% | 0% |
| **BT** | 0% | 100% | 0% |
| **SP** | 0% | 100% | 0% |
| **LU** | 100% | 0% | 0% |

For the point-to-point operations, IS, CG and LU contain only static communications. MG, BT, SP contain only persistent communications. For BT and SP, the destination set for each node is calculated prior to all point-to-point communications and are used through application completion. For MG, there are two communication stages. In each stage, the destination set for each node is calculated prior to the communications and is retained until each stage completes.

In the case of collective operations both IS and FT contain all-to-all communications. While the number of total nodes increases, the all-to-all communications dominates the message volume. So the ratio of static communications (e.g. point-to-point) to collective communications becomes very low and almost can be ignored.

**Table 4. The percentages of different collective communications in NAS benchmarks.**

| Benchmark | Static | Persistent | Dynamic |
|---|---|---|---|
| **IS** | 0.4% | 0% | 99.6% |
| **CG** | 100% | 0% | 0% |
| **MG** | 100% | 0% | 0% |
| **EP** | 100% | 0% | 0% |
| **FT** | 0% | 100% | 0% |
| **BT** | 100% | 0% | 0% |
| **SP** | 100% | 0% | 0% |
| **LU** | 100% | 0% | 0% |

### 5.2. Identifying Communication Patterns

To show the capability of our compiler to identify communication patterns, we consider one application LBMHD (Lattice Boltzmann model of magneto-hydrodynamics) and the IS, LU, MG and CG benchmark programs from the NAS parallel benchmark suite. While information for the entire benchmark suite can be generated, we demonstrate only 4 due to space limitations.

**IS:** The compiler identified communication pattern for IS has been shown in Table 2 and Figure 5. The weights in $c-vectors$ and $p-matrices$ are binary values. In phase 0, collective operations `AR`, `AA`, and `AV` are executed with no point-to-point communication. This is similar for phase 1. Phase 2 combines the `RD` collective operation with the point-to-point matrix shown

in Figure 5. By using Merge and Unwrap operations, phase 0 and 1 can be combined.

**LBMHD:** The communication pattern of application LBMHD is shown in Figure 8 and Figure 9. Our compiler identified a single communication phase. Because the number of processors $N$ and the processor rank $rank$ are not known at compile-time, we generate a formula list in Figure 8 to describe the $p-matrix$. Each processor has a set of four other processors with which it communicates. Since $N$ and $rank$ are the only symbols in the expression, this matrix is considered statically known as it may be entirely resolved at load-time to execution. Thus it is possible to entirely configure the network for LBMHD based on compiler analysis prior to execution. Figure 9 shows the pattern from a run on 64 processors.

$$\begin{pmatrix} ((\lfloor rank/N_y \rfloor + 1) \mod N_x + (rank - \lfloor rank/N_y \rfloor * N_y) \\ ((\lfloor rank/N_y \rfloor - 1) \mod N_x + (rank - \lfloor rank/N_y \rfloor * N_y) \\ \lfloor rank/N_y \rfloor * N_y + ((rank - \lfloor rank/N_y \rfloor * N_y) + 1) \mod N_y \\ \lfloor rank/N_y \rfloor * N_y + ((rank - \lfloor rank/N_y \rfloor * N_y) - 1) \mod N_y \end{pmatrix}$$

**Figure 8. LBMHD $p-matrix$ described by a formula list where $N = N_x * N_y$.**

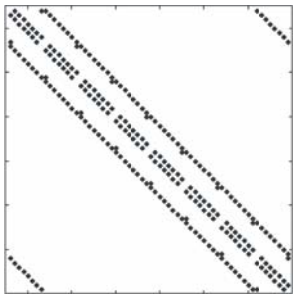


**Figure 9. LBMHD $p-matrix$ for 64 processors.**

While compiling any NAS parallel benchmarks, the total number of processors, referred as $N$ here, has to be given as a build parameter. Therefore, $N$ is known at compile-time. We show the analysis results for CG and MG running on $N = 128$ processors.

**CG:** The compiler initially identified two communication phases from the source code. It turns out that each of these phases are identical and can be merged. The $p-matrix$ for the compiler predicted communication pattern is shown in Figure 10.

**MG:** When analyzing MG, the compiler discovered that the communication destinations depend on run-time input data. However, after the topologies are determined, they are used for an extended period of time. Hence the communication pattern is persistent. Therefore, it is only possible to construct symbolic $p-matrices$ or formula lists from the source code. However, the input data for the program is provided by a pre-generated input file, it is possible to use these values to construct deterministic $p-matrices$.

Using values in a particular input file in compile-time analysis, We constructed 12 p-matrices shown

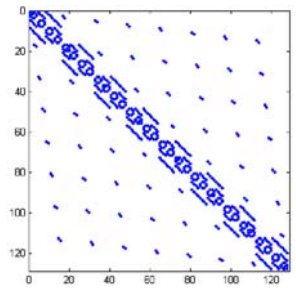
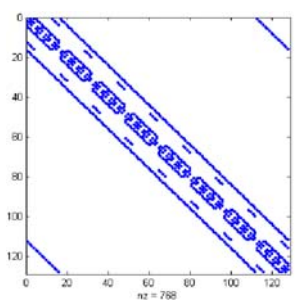
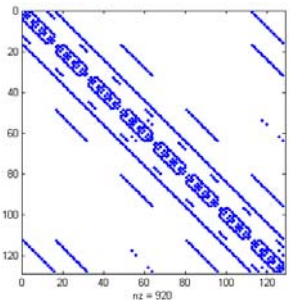


**Figure 10. The $p-matrix$ of CG (128 nodes).**

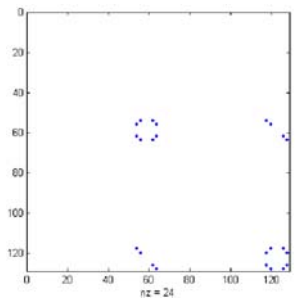
in Figure 11. $p-matrices$ 0, 1, 6-10 correspond to Figure 11(a), $p-matrices$ 2 and 11 correspond to Figure 11(b), $p-matrix$ 3 corresponds to Figure 11(c), and $p-matrix$ 4 corresponds to Figure 11(d). $p-matrix$ 5 is empty because the branches containing zero point-to-point communications were taken in the decision points in this case.



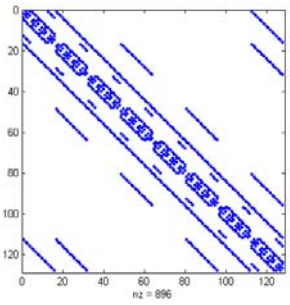
(a) $p-matrices$ 0,1,6-10.



(b) $p-matrices$ 2,11.



(c) $p-matrix$ 3.



(d) $p-matrix$ 4.

**Figure 11. Predicted $p-matrices$ of MG.**

The experimental compiler is also capable of automatically generating traces to help verify the detected communication patterns. First the compiler identifies the communication pattern. Second, it inserts trace generation instructions for communication operations. The resulting MPI program is annotated with trace generation statements. This code can be compiled and executed normally to generate trace files. This tech-

nique was used to verify our results for the LBMHD, CG, and MG benchmarks.

## 6. Conclusions

In this paper we have presented a compiler framework analysis of communication operations for parallel application. This framework can identify communication patterns directly from source code. Additionally, the compiler framework can automatically insert network configuration instructions and/or trace generation instructions directly into the application for setting up network configurations and collecting traces, respectively. An experimental compiler has been implemented based on the SUIF compiler infrastructure.

One of the main contributions of this work is the development of a communication pattern representation scheme for our experimental compiler. This scheme is flexible and can be easily tailored to many types of communication analysis. It also provides the power to easily manipulate the granularity of communication phases and/or translate collective communications to point-to-point communications using a set of proposed operations on the patterns.

Applying our experimental compiler on the NAS parallel benchmark suite, we found a large portion of communications which are classified as dynamic [8] to actually be persistent. This provides opportunities for pre-configuring network to reduce communication overhead.

Potential future work includes enhancing the control and data flow analysis to more accurately detect persistent communications within an application. Additionally, the compiler can be used with a variety of different types of network topologies to evaluate the performance impact of predicting the communication pattern at compile-time. Finally, this idea can be expanded for non-message passing software layers such as distributed shared memory architectures that give the appearance of shared memory but require actual messages to traverse the network.

## References


[1] D. Bailey, T. Harris, W. Sahpir, and R. van der Wijngaart. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, December 1995.

[2] K. J. Barker, A. Benner, R. Hoare, A. Hoisie, A. K. Jones, D. J. Kerbyson, D. Li, R. Melhem, R. Rajamony, E. Schenfeld, S. Shao, C. Stunkel, and P. A. Walker. On the feasibility of optical circuit switching for high performance computing systems. In *Proc. of SC*, 2005.

[3] F. Cappello and C. Germain. Toward high communication performance through compiled communications on a circuit switched interconnection network. In *Proc. of the Int. Symp. on High Performance Computer Architecture (HPCA)*, pages 44–53, 1995.

[4] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. *ACM Computer Architecture News*, 21(2):2–13, May 1993.

[5] H. G. Dietz and T. Mattox. Compiler techniques for flat neighborhood networks. In *Proc. of 13th Int. Wrokshop on Languages and Compilers for Parallel Computing*, 2000.

[6] Z. Ding, R. Hoare, A. Jones, D. Li, S. Shao, S. Tung, J. Zheng, and R. Melhem. Switch design to enable predictive multiplexed switching in multiprocessor networks. In *Proc. of the Int. Parallel and Distributed Procssing Symp. (IPDPS)*, 2005.

[7] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. Margan Kaufmann, 2003.

[8] A. Faraj and X. Yuan. Communication characteristics in the NAS parallel benchmarks. In *Proc. of the Parallel and Distributed Computing and Systems Conf. (PDCS)*, 2002.

[9] J. Fernandez, E. Frachtenberg, and F. Petrini. Bcs-mpi: A new approach in the system software design for large-scale parallel computers. In *Proc. of the ACM/IEEE Conf. on SC*, 2003.

[10] S.-Y. Ho and N.-W. Lin. Static analysis of communication structures in parallel programs. In *Proc. of the Int. Computer Symp.(ICS)*, pages 215–221, 2002.

[11] D. Lahaut and C. Germain. Static communcations in parallel scientific programs. In *Proc. of PARLE*, 1994.

[12] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier. An architecture and compiler for scalable on-chip communication. *IEEE Trans. on Very Large Scale Integration Systems (TVLSI)*, 12(4):711–726, July 2004.

[13] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995.

[14] J. Shalf, S. Kamil, L. Oliker, and D. Skinner. Analyzing ultra-scale application communication requirements for a reconfigurable hybird interconnect. In *Proc. of SC*, 2005.

[15] D. Shires, L. Pollock, and S. Sprenkle. Program flow graph construction for static analysis of mpi programs. In *Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications(PDPTA)*, June 1999.

[16] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *Journal of Parallel and Distributed Computing*, 63(9):853–865, September 2003.

[17] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarsinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[18] X. Yuan, R. Melhem, and R. Gupta. Compiled communication for all-optical TDM networks. In *Proc. of SC*, 1996.