Algorithms for Supporting Compiled Communication *

Xin Yuan[†]
Dept. of Computer Science
Florida State University
Tallahassee, FL 32306
xyuan@cs.fsu.edu

Rami Melhem
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
melhem@cs.pitt.edu

Rajiv Gupta
Dept. of Computer Science
University of Arizona
Tucson, AZ 85721
gupta@cs.arizona.edu

Abstract

In this paper, we investigate the compiler algorithms to support compiled communication in multiprocessor environments and study the benefits of compiled communication assuming that the underlying network is an all–optical Time–Division–Multiplexing (TDM) network. We present an experimental compiler, *E–SUIF*, that supports compiled communication for High Performance Fortran (HPF) like programs on all–optical TDM networks, describe and evaluate the compiler algorithms used in *E–SUIF*. We further demonstrate the effectiveness of compiled communication on all–optical TDM networks by comparing the performance of compiled communication with that of a traditional communication method using a number of application programs.

Keywords: Compiled Communication, Communication Optimization, Compilation Techniques for Distributed Memory Machines, All-optical Networks.

*This research was supported in part by NSF grants: CCR-9904943, CCR-0073482 and ANI-0106706.
[†]Preferred Address: Xin Yuan, 254 Love Building, Department of Computer Science, Florida State University, Tallahassee, FL 32306, USA. Phone: (850)644-9133. Fax: (850)644-0058. Email: xyuan@cs.fsu.edu.

1 Introduction

In traditional high performance computing systems, communication optimizations are performed either in the compiler by considering the communication requirement of the programs or in the communication library by exploiting architectural features of the underlying network to realize communication routines efficiently. In both cases, the optimizations are inherently limited. Specifically, the optimizations in the compiler typically cannot exploit architectural features of the network since the communication library usually hides the network details, while the optimizations in the communication library cannot be performed across communication patterns since the communication library does not have the information about the sequence of communication patterns in an application program. Compiled communication overcomes these limitations and exploits more optimization opportunities.

In compiled communication, the compiler analyzes a program to determine its communication requirement. The compiler then uses the knowledge of the underlying network and the communication requirements to manage network resources statically. Since a typical network has limited resources and cannot efficiently support arbitrary communication patterns, the compiler must partition the program into *phases* such that each phase contains a fixed, pre–determined pattern that can be supported efficiently. The compiler then manages the communications within each phase statically and inserts code at phase boundaries to reconfigure the network to support the communications within each phase.

Compiled communication offers many advantages over traditional communication methods. First, by managing network resources at compile time, some runtime communication overheads, such as buffer management, can be eliminated. Second, compiled communication can use *long–lived* connections for communication and amortize the startup overhead over a number of messages. Third, compiled communication can improve network resource utilization by using off–line resource management algorithms. Last but not the least, compiled communication can optimize arbitrary communication patterns (when the patterns can be determined at the compile time) instead of a set of predefined collective communications supported by a communication library. Generally, compiled communication allows broader program optimization in comparison to library based communication since compiled communication is not limited to individual communication. The limitation of compiled communication is that it is effective only for the communication patterns that are known at compile time. Studies [17] have shown that more than 99% of communication patterns in scientific programs can be determined completely or parametrically at compile time. Only less than 1% are unknown at compile time. Thus, using the compiled communication technique is likely to improve the overall communication performance for scientific programs.

All–optical interconnection networks are promising networks for future parallel computers. Multiplexing techniques, such as *time–division multiplexing* (TDM) [19, 25, 27] and *wavelength–division multiplexing* (WDM) [3], are typically used to exploit the large bandwidth in optical networks. While all–optical networks have the potential to provide large bandwidth, *dynamic all–optical communication* requires an all–optical path to be established before the data transmission starts. This connection management task places strict demands on the control of the interconnection network. Specifically, the network control, be it centralized or distributed, is usually performed in the electronic domain and thus is very slow in comparison to the large bandwidth supported by optical data paths. In this paper, we attempt to use the compiled communication technique to reduce the connection management overhead for the communication patterns that can be determined at compile time.

We study the compiler algorithms for supporting compiled communication in multiprocessor environments and demonstrate the benefits of compiled communication on all–optical TDM networks. We present our experimental compiler, *E–SUIF*, which is an extension of the Stanford SUIF(Stanford University Intermediate Format) compiler [1]. E–SUIF supports compiled communication for High Performance Fortran (HPF) like programs on all–optical TDM networks. Although E–SUIF targets all–optical TDM networks, most of the compiler algorithms can also apply to other types of networks.

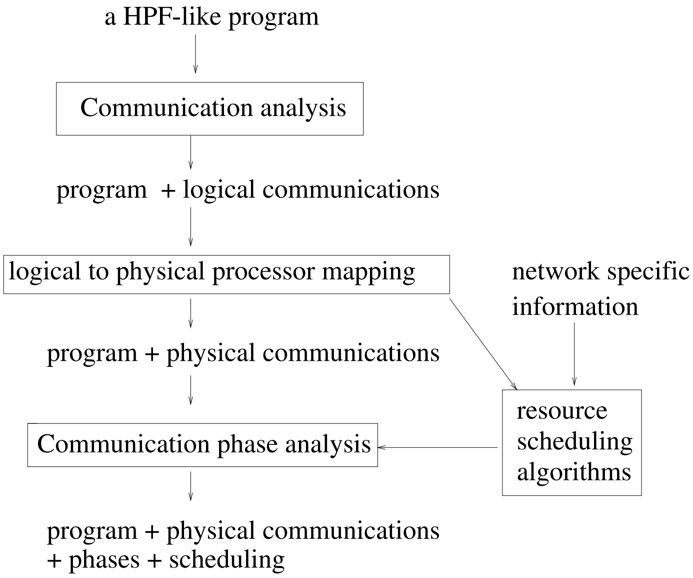

Figure 1: The E–SUIF compiler

The major components in E–SUIF are shown in Figure 1. The first component of *E–SUIF*, *communication analysis*, analyzes the logical communication requirement of a program and performs a number of high–level communication optimizations, such as message vectorization. The analyzer represents the logical communication requirement using Section Communication Descriptors (SCDs) [24]. The second component, *logical to physical processor mapping*, derives physical communications, that is, communications on physical processors, from SCDs. The *resource scheduling algorithms* take network specific information and communication patterns in application programs as inputs and determine whether a communication pattern can be supported by the underlying network, and if the pattern can be supported, how the network resources are allocated to support the communication pattern. In *E–SUIF*, network resources are communication channels in optical TDM networks. Notice that this module is architectural dependent. For different networks, different network resources can be managed by the compiler and different optimizations can be applied in compiled communication. The third component, *communication phase analysis*, utilizes the resource scheduling algorithms to partition the program into phases such that communications in each phase can be supported efficiently by the underlying network and that there is a minimal number of phases at runtime. The *E–SUIF* compiler outputs a program with physical communications, phases and the resource scheduling for each phase, and thus, supports the compiled communication technique.

In this paper, we describe and evaluate the compiler algorithms used in *E–SUIF*. We further demonstrate the benefits of compiled communication on all–optical TDM networks by comparing the performance of compiled communication with that of a traditional communication method using a number of application programs. Notice that there are many issues in optimizing parallel applications, such as the mapping of processing and data into physical processes. In this paper, however, we focus on optimizing communication through compiled communication and assume that those issues are addressed by other parts of the compiler.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 introduces the programming model. Section 4 presents the communication analysis algorithms. Section 5 introduces the communication phase analysis algorithms. Section 6 evaluates the compiler algorithms. Section 7 reports the performance study of compiled communication on all–optical TDM networks. Section 8 concludes the paper.

2 Related work

Many projects have focused on reducing the communication overheads in the software messaging layer [8, 9, 18]. While this approach is beneficial for all types of communications, it does not expose architectural dependent optimization opportunities to the compiler.

Many parallel compiler projects tried to improve communication performance by generating efficient communication code for distributed memory machines [1, 2, 7, 10, 12, 21]. To simplify the compilation, these compilers use the dynamic communication model and do not exploit the potential of compiled communication. Communication analysis and optimization has been applied in parallel compilers. Early approaches optimize communications in a single loop nest using data dependence information [1, 12]. Later, data flow analysis techniques have been developed to obtain information for global communication optimizations [7, 10, 15, 24]. However, the analysis only obtains logical communication information, which is insufficient for compiled communication. The interaction between other components, such as program partitioning and data mapping, and the communication sub-system, has also been investigated [22], which offers another dimension for communication optimization.

Research on compiled communication has drawn the attention of a number of research groups [4, 6, 11, 16]. In [4], the compiler applies the compiled communication technique specifically to the stencil communication pattern. In [16], a special purpose machine is designed to support compiled communication. Compiled communication was proposed for a general purpose machine using a multistage interconnection network [6]. However, since the multi–stage interconnection network without multiplexing has very limited capacity to support connections, compiled communication results in excessive synchronization overhead. The work in [11] proposed to amortize the startup overhead using long–lived connections and to perform architecture–dependent communication optimizations. However, the compiler in [11] can only recognize a very limited number of communication patterns. *E–SUIF* is more powerful in physical communication analysis.

The *E–SUIF* compiler is built on top of the Stanford SUIF compiler [1]. The SUIF compiler provides an excellent compiler infrastructure for Fortran and C languages. However, it does not support compiled communication. In this work, we incorporate all necessary algorithms to support compiled communication, including communication analysis, resource scheduling, and communication phase analysis, into the SUIF compiler.

3 Programming Model

We consider structured HPF–like programs, which contain conditionals and nested loops, but no arbitrary goto statements. The array subscripts are assumed to be of the form $\alpha * i + \beta$, where α and β are invariants and i is a loop index variable. The programmer explicitly

```
          ALIGN (i, j) with VPROCS(2*j, i+2, 1) :: x
          ALIGN (i) with VPROCS(1, i+1, 2) :: y
(1)       DO 100 i = 1, 5
(2)         DO 100 j = 1, 5
              Comm : [y, (i), [src = (1, i + 1, 2), dst = (2 * j, i + 2, 1), NULL], NULL]
(3) 100       x(i, j) = y(i) + 1
```

Figure 2: A HPF-like program

specifies the data alignments and distributions. To simplify the discussion, we assume in this paper that all arrays are aligned to a single virtual processor grid template, and the data distribution is specified through the distribution of the template. For example, in the program in Figure 2, shared arrays x and y are aligned to $VPROCS$. *E–SUIF* handles multiple virtual processor grids.

Arrays are aligned to the virtual processor grid by simple affine functions. The alignments allowed are scaling, axis alignment and offset alignment. The mapping from a point $\vec{d}$ in the data space to the corresponding point $\vec{v}$ in the virtual processor grid is specified by an alignment matrix M and an alignment offset vector $\vec{\alpha}$, that is, $\vec{v} = M\vec{d} + \vec{\alpha}$. The distribution of the virtual processor grid can be cyclic, block or block–cyclic. Assuming that there are p processors in a dimension, and the block size of that dimension is b, the virtual processor v is in physical processor $\frac{v \ mod \ (p*b)}{b}$. For cyclic distribution, $b = 1$. For block distribution, $b = N/p$, where N is the size of the dimension. We will use the notation block–cyclic(b, p) to denote the block–cyclic distribution with block size of b over p processors for a specific dimension of a distributed array.

4 Communication analysis

To support compiled communication, the compiler must have the knowledge of the physical communication requirement of a program, that is, communication patterns on the physical processors. In the rest of the paper, we will use *physical communication* to denote communication patterns on the physical processors and *logical communication* to denote logical communication patterns on the logical processor grids. *E–SUIF* obtains physical communication requirement of a program in two steps. In the first step, *logical communication analysis* analyzes the logical communication requirement of the program and carries out a number of high level communication optimizations. In the second step, the *physical communication analysis* derives physical communications from logical communications. Next, we will discuss these two steps.

4.1 Logical communication analysis

E–SUIF uses a demand driven communication analyzer [24] to analyze the logical communication requirement of a program. The analyzer performs message vectorization, global redundant communication elimination and global message scheduling [7] optimizations and represents logical communications using *Section Communication Descriptors* (SCDs). In the rest of this subsection, we will describe SCD and how SCDs are used to represent logical communications. Details about the communication optimizations and the analyzer can be found in [24].

A *Section Communication Descriptor* (SCD) describes a logical communication by specifying the source array region involved in the communication and the communication pattern for each element in the array region. A $SCD = [N, D, CM, Q]$ consists of four components. The first component is an array name N and the second component is the array region descriptor D. The third component is the communication mapping descriptor CM that describes the source–destination relationship of the logical communication. Finally, the fourth component is a qualifier descriptor Q that specifies iterations during which communication is performed.

D represents the source array region involved in the communication using *bounded regular section descriptor* (BRSD)[5]. It contains a vector of subscript values. Each element in the vector is either (1) an expression of the form $\alpha * i + \beta$, where α and β are invariants and i is a loop index variable, or (2) a triple $l : u : s$, where l, u and s are invariants. The triple, $l : u : s$, defines a set of values, $\{l,\ l+s,\ l+2s,\ ...,\ u\}$, as in the HPF array statement.

The source–destination mapping CM is denoted as $[src \rightarrow dst, qual]$. Both src and dst are vectors whose elements are of the form $\alpha * i + \beta$, where α and β are invariants and i is a loop index variable. The pair src and dst specifies the logical source and destination relation of the communication. That is, given a logical processor of the source, which can be computed from the source data element in D, the logical processor of the destination can be calculated through this relation. To specify the broadcast communication, the *mapping qualifier* list, $qual$, is introduced. $Qual$ contains a list of range descriptors. Each range descriptor is of the form $i = l : u : s$, where l, u and s are invariants and i is a loop index variable. Each range descriptor specifies the range of a variable in dst but not in src. Notation $qual = NULL$ denotes that no mapping qualifier is needed.

The qualifier Q is a range descriptor of the form $i = l : u : s$, where i is the loop index variable of the loop that directly encloses the SCD. This qualifier indicates the iterations of the loop in which the SCD is performed. If the SCD is to be performed in every iteration in the loop, $Q = NULL$. Q will be referred to as the *communication qualifier*.

Consider the example in Figure 2. The assignment statement at line (3) requires communication. Let us assume the *owner computes* rule, which requires that the processor that owns the array element in the left hand side of an assignment statement performs the assignment and that shared array elements in the right hand side of the assignment statement must be transmitted to the processor that owns the left hand side array element. The communication for moving $y(i)$ from its owner to the processor that owns $x(i,j)$ can be represented as $[y, (i), [src = (1, i+1, 2), dst = (2*j, i+2, 1), NULL], NULL]$. This communication can be vectorized and moved out of the loops. When the communication is hoisted out of the j loop in line (2), the analyzer will detect that there is a broadcast for each element $y(i)$ and put $j = 1 : 5 : 1$ in the mapping qualifier to represent the broadcast. When the communication is then hoisted out the the i loop in line (1), the array elements are aggregated as $1 : 5 : 1$.

As a result, the communication is placed outside the loop and is represented as

$$SCD = [y, (1:5:1), [src = (1, i+1, 2), dst = (2*j, i+2, 1), qual = \{j = 1:5:1\}], NULL].$$

4.2 Physical communication analysis

This subsection describes the algorithms to compute physical communications from SCDs. We assume that the physical processor grid has the same number of dimensions as the logical processor grid. *Processor grid* will denote both physical and logical processor grids. Notice that this is not a restriction because a dimension in the physical processor grid can always be collapsed by assigning a single physical processor to that dimension. Notice also that calculating physical communication does not take the network topology into consideration.

4.2.1 One–Dimensional arrays and one–dimensional processor grids

Let us consider the case where the distributed array and the processor grid are one-dimensional. We are interested in deriving physical communication from a $SCD = [N = A, D, CM = [src \rightarrow dst, qual], Q]$, where $src = \alpha * i + \beta$ and $dst = \gamma * i + \delta$. We assume that $\alpha \neq 0$, $\gamma \neq 0$, and $qual = NULL$. The cases where $\alpha = 0$, $\gamma = 0$ or $qual \neq NULL$ will be considered later when multi-dimensional arrays and processor grids are discussed. Let the alignment matrix and the offset vector be M_A and v_A, that is, element $A[n]$ is owned by logical processor $M_A * n + v_A$. Assuming that the distribution of the logical processor template is block–cyclic(b, p), the physical source processor for communicating $A[n]$ is given by $\frac{(M_A*n+v_A) \ mod \ (p*b)}{b}$. The logical destination processor can be computed by first solving the equation $(M_A * n + v_A) = \alpha * i + \beta$ to obtain $i = \frac{(M_A*n+v_A-\beta)}{\alpha}$ and then replacing the value of i in dst to obtain the logical destination processor $\gamma * (M_A * n + v_A - \beta)/\alpha + \delta$. Thus, the physical destination processor is given by

$$\frac{\gamma*(M_A*n+v_A-\beta)/\alpha+\delta) \ mod \ (p*b)}{b}.$$

The array region D may need to be expanded using the communication qualifier Q or using the range for a loop index variable when the communication is inside a loop. The physical

```
Compute_1–dimensional_pattern(D, CM.src, CM.dst)

  Let D = l : u : s, CM.src = α * i + β, CM.dst = γ * i + δ
  if (l contains variables) then
    return all–to–all connections
  end if
  if (α, β, γ or δ are variables) then
    return all–to–all connections
  end if
  if (s contains variables) then s = 1
  pattern = φ
  for each element i in D do
    pattern = pattern + communication of element i
    if (communication repeated) then
      return pattern
    end if
  end for
```

Figure 3: Algorithm for 1-dimensional arrays and 1-dimensional processor grids

communication pattern for the SCD can be obtained by considering all elements in the array region D. Computing the physical communication of a SCD using this brute–force method, however, is both inefficient and, sometimes, infeasible when D cannot be determined at compile time. Fortunately, for a general block–cyclic(b, p) distribution, we do not need to examine every element in D to determine the physical communication as shown in the following lemma (the proof of the lemma can be found in [26]).

Lemma: Assume that the template is distributed using the block–cyclic(b, p) distribution. Let $SCD = [A, D = l : u : s, CM = [src \rightarrow dst, qual], Q]$, assuming u is infinite, there exists a k, $k \leq p^2b^2$, such that the communication for all $m \geq k$, $A[l + m * s]$ has the same source and destination as the communication for $A[l + (m - k) * s]$.

The implication of the lemma is that the physical communication pattern for a SCD can be determined by examining the communications for at most p^2b^2 elements. In addition, when the upper bound of D is unknown, the communication pattern can be approximated by considering all elements up to the repetition point.

Figure 3 shows the algorithm to compute the physical communication pattern for a 1–dimensional array and an 1–dimensional virtual processor grid. Given $D = l : u : s$ and

Compute communication pattern(SCD)
  Let $SCD = [A, D, CM, Q]$
  if (the form of CM cannot be processed) **then**
    return all-to-all connections
  end if
  $pattern = \{(*, *, ..., *) \rightarrow (*, *, ..., *)\}$
  for each dimension i in array A **do**
    Let sd be the corresponding dimension in source processor grids.
    Let dd be the corresponding dimension in destination processor grids.
    if (dd exists) **then**
      1dpattern = compute_1-dimensional_pattern($D[i]$, $CM.src[sd]$, $CM.dst[dd]$)
    else
      1dpattern = compute_1-dimensional_pattern($D[i]$, $CM.src[sd]$, $\perp$)
    end if
    pattern = cross_product(pattern, 1dpattern)
  end for
  pattern = source_processor_constants(pattern)
  for each element i in the mapping qualifier **do**
    Let dd be the corresponding destination processor dimension.
    1dpattern = compute_1-dimensional_pattern($CM.qual[i]$, $\perp$, $CM.dst[dd]$)
    pattern = cross_product(pattern, 1dpattern)
  end for
  pattern = destination_processor_constants(pattern)
  return pattern

Figure 4: Algorithm for multi–dimensional arrays

$CM = [\alpha * i + \beta \rightarrow \gamma * i + \delta, NULL]$, if l contains variables or if any of α, β, γ or δ is a symbolic constant, the communication is approximated with all–to–all connections. When s contains variables, it will be approximated by 1, that is, D is approximated by a superset $l : u : 1$. After the testing and approximations, the algorithm accumulates the connection requirement for communicating each element in D until the repetition point is reached. The algorithm in Figure 3 is an $O(p^2b^2)$ algorithm that computes physical communications for general block–cyclic(b, p) distributions. For block distributions, a more efficient $O(p^2)$ algorithm can be designed.

4.2.2 Multi–dimensional arrays and multi–dimensional processor grids

The algorithm to compute physical communications for multi–dimensional arrays and multi–dimensional processor grids is given in Figure 4. In the algorithm, we use the notion $\perp$

to represent a "don't care" parameter. In an n–dimensional processor grid, a processor is represented by an n–dimensional coordinate $(p_1, p_2, ..., p_n)$. The algorithm determines all pairs of source and destination processors that require communication by reducing the problem into computing 1–dimensional communication sub–problems.

The first step in the algorithm is to check whether the mapping relation can be processed. If one loop induction variable occurs in two or more dimensions in $CM.src$ or $CM.dst$, the algorithm cannot find the correlation between dimensions in source and destination processors, and the communication pattern for the SCD is approximated by all–to–all connections.

If the SCD passes the mapping relation test, the algorithm initializes the communication $pattern$ to be $(*, *, ..., *) \rightarrow (*, *, ..., *)$. The wild card * in $pattern$ means that the particular dimension has not been processed. The algorithm then determines for each dimension in the data space the corresponding dimension sd in the source processor grid. If it does not exist, the data dimension is not distributed and does not need be considered. If there exists such a dimension, the algorithm then tries to find the corresponding dimension dd in the destination processor grid by checking whether there is a dimension dd such that $CM.dst[dd]$ contains the same looping index variable as the source dimension $CM.src[sd]$. If such dimension exists, the algorithm computes 1-dimensional communication pattern between dimension sd in the source processor and dimension dd in the destination processor, then cross–products the 1-dimensional communication pattern into the n-dimensional communication pattern. Here, the cross–product operation is similar to the cross product of sets except that specific dimensions are involved in the operation. When dd does not exist, the algorithm determines a degenerated 1-dimensional pattern, where only source processors are considered, and cross-products the degenerate 1-dimensional pattern into the communication pattern. This is the case when $\gamma = 0$ and the destination is mapped to a constant logical processor. After all dimensions in the data space are considered, there may still exist dimensions in the source processor (in the virtual processor grid) that have not been considered. These dimensions should be constants that can be determined from the alignment matrix and the alignment offset vector. This is the case when $\alpha = 0$ and the source is mapped to a constant logical processor. The algorithm fills in the constants in the source processors. Dimensions in destination processor may not be fully considered yet. When $CM.qual \neq NULL$, the algorithm finds for each item in $CM.qual$ the corresponding dimension in the destination processor, computes all possible processors in that dimension and cross–products the list into the communication list. Finally, the algorithm fills in all constant dimensions in the destination. The constant dimensions in the processor grids corresponds to the cases where $CM.src[sd] = \alpha * i + \beta$ with $\alpha = 0$ and $CM.dst[dd] = \gamma * i + \delta$ with $\gamma = 0$.

Consider the example in Figure 2. Let us assume that the virtual processor grid, $VPROCS$, is distributed as (block–cyclic(2,2), block–cyclic(2,2), block–cyclic(1,1)). As discussed earlier, the logical communication is represented as

$$[y, (1:5:1), [src = (1, i+1, 2), dst = (2*j, i+2, 1), qual = \{j = 1:5:1\}], NULL].$$

The physical communication for this SCD is computed as follows. First consider the dimension 0 in the array y. From the alignment, the algorithm knows that dimension 1 in the virtual processor grid corresponds to this dimension in the data space. Checking dst in CM, the algorithm finds that dimension 1 in destination corresponds to dimension 1 in source processors. Applying the 1-Dimensional mapping algorithm, an 1–dimensional communication pattern $\{0 \rightarrow 1, 1 \rightarrow 0\}$ with $ss = 1$ and $dd = 1$ is obtained. Thus the communication list becomes $\{(*, 1, *) \rightarrow (*, 0, *), (*, 0, *) \rightarrow (*, 1, *)\}$ after taking the cross product with the 1–dimensional pattern. Next, the other dimensions in source processors, including dimension 0 that is always mapped to processor 0 and dimension 2 that is always mapped to processor 1 are considered. After filling in the physical processor in these dimensions in source processors, the communication pattern becomes $\{(0, 1, 1) \rightarrow (*, 0, *), (0, 0, 1) \rightarrow (*, 1, *)\}$. Considering the $qual$ in M, the dimension 0 of the destination processor can be either 0 or 1. Applying the cross product operation, the new communication list $\{(0, 1, 1) \rightarrow (0, 0, *),\ (0, 1, 1) \rightarrow (1, 0, *),\ (0, 0, 1) \rightarrow (0, 1, *),\ (0, 0, 1) \rightarrow (1, 1, *)\}$ is obtained. Finally, the dimension 2 in the destination processor is always mapped to processor 0. Thus, the final physical communication is given by

$$\{(0,1,1) \rightarrow (0,0,0), (0,1,1) \rightarrow (1,0,0), (0,0,1) \rightarrow (0,1,0), (0,0,1) \rightarrow (1,1,0)\}.$$

There are several levels of approximations in computing the physical communication. First, when the algorithm cannot correlate the source and destination processor dimensions from the mapping relation, the algorithm uses an approximation of all–to–all connections. If the mapping relation contains sufficient information to distinguish the relation of the source and destination processor dimensions, computing the communication pattern for a multi-dimensional array reduces to computing 1-dimensional communication patterns, thus the approximations within each dimension are isolated to that dimension and will not affect the patterns in other dimensions. Using this multi-level approximation scheme, some useful information is obtained even when the compiler does not have complete information for a communication.

5 Communication Phase analysis

An all–optical TDM network can only support a limited multiplexing degree and thus, may not be able to support an arbitrary communication pattern without incurring large control overheads. Thus, the compiler must partition a program into phases such that each phase contains communications that can be effectively supported by the underlying networks. This task is called *communication phase analysis*. *E–SUIF* utilizes the resource management algorithms we developed in [23] to perform communication phase analysis. Given a physical communication pattern and an all–optical TDM networks architecture, these algorithms determine whether the communication pattern can be supported by the network. If the network can support the communication pattern, the algorithms also determine the channel assignment to support the communication pattern. The goal of communication phase analysis is to partition the program into phases such that each phase contains communications that can be effectively supported and to minimize the number of phases in the program to reduce the synchronization overhead.

The communication phase analysis is carried out in a recursive manner on the high level SUIF representation of a program. SUIF represents a program in a hierarchical manner. A procedure contains a list of SUIF nodes, where each node can be of different types and may contain sub–lists. Some SUIF node types are TREE_FOR, TREE_LOOP, TREE_IF, TREE_BLOCK and TREE_INSTR. A TREE_FOR node represents a for–loop structure. It contains four sub–lists, *lb_list* which contains the nodes to compute the lower bound, *ub_list* which contains the nodes to compute the upper bound, *step_list* which contains the nodes to compute the step, and *body* which contains the loop body. A TREE_LOOP node represents a while–loop structure. It contains two sub–lists, *test* and *body*. A TREE_IF node represents an if–then–else structure. It contains three sub–lists, *header* which is the test part, *then_part* which contains the nodes in the then part, and *else_part*. A TREE_BLOCK node represents a block of statements, it contains a sub–list *body*. A TREE_INSTR nodes represents a statement.

In the analysis, two variables, *pattern* and *kill_phase*, are associated with each *composite node* that contains sub–lists. The variable *pattern* records the communication pattern that is exposed from its sub–lists. The variable *kill_phase* has a boolean value indicating whether its sub–lists contain communication phases. The communication phase analysis algorithm in *E–SUIF* is shown in Figure 5. Given a list of SUIF nodes, which is the SUIF program representation, the algorithm first recursively examines the sub–lists of all nodes and annotates the composite nodes with *pattern* and *kill_phase*, then considers the phases in the list. This post–order traversal of the SUIF program accumulates the communications in the innermost loops first, and thus captures the communication locality when it exists and is supported by the underlying network.

After all sub–lists are analyzed, the program becomes a straight line program (list), whose nodes are annotated with *pattern* and *kill_phase*. The algorithm examines all these annotations in each node from back to front. A variable *c_pattern* is used to maintain all

```
Communication_Phase_Analysis(list)
Input: list: a list of SUIF nodes
Output: pattern: communication pattern exposed out of the list
        kill_phase: whether there are phases within the list

Analyze communication phases for the sub–lists for each node.
c_pattern = NULL, kill_phase = 0
For each node n in list in backward order do
  if (n is annotated with kill_phase) then
    Generate a new phase for c_pattern after n.
    c_pattern = NULL, kill_phase = 1
  end if
  if (n is annotated with communication pattern a) then
    new_pattern = c_pattern + a
    if (the network can support new_pattern) then
      c_pattern = new_pattern
    else
      Generate a new phase for c_pattern after n.
      c_pattern = a, kill_phase = 1
    end if
  end if
end for
return c_pattern and kill_phase
```

Figure 5: Communication phase analysis algorithm

```
Communication_Phase_Analysis for TREE_IF

Analyze the header list.
Analyze the then_part list.
Analyze the else_part list.
Let comb = the combined communications from the three sub–lists.
If (there are phase changes in the sub–lists) then
  Generate a phase in each sub–list for the communication exposed.
  pattern = NULL, kill_phase = 1
if (the network cannot support comb) then
  Generate a phase in each sub–list for the communication exposed.
  pattern = NULL, kill_phase = 1
else
  pattern = comb, kill_phase = 0
end if
Annotate the TREE_IF node with pattern and kill_phase.
```

Figure 6: Communication phase analysis for TREE_IF nodes

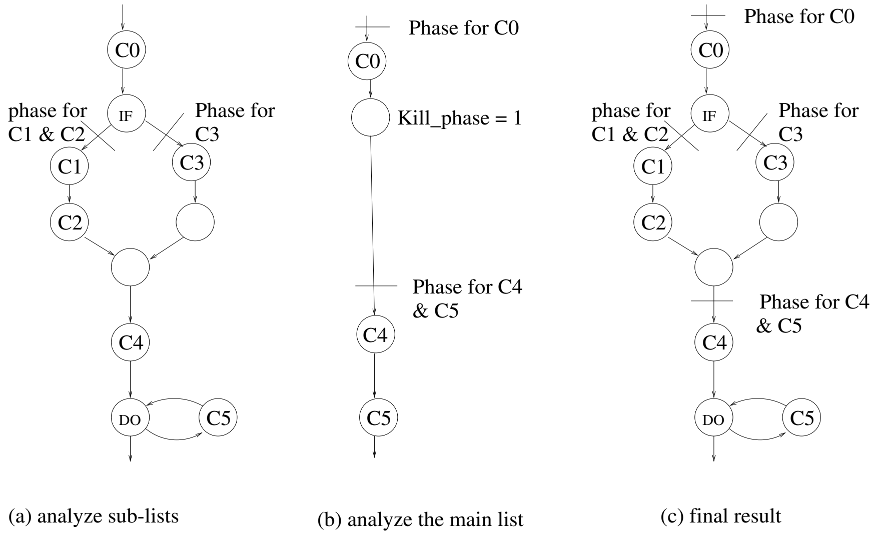

Figure 7: An example of the communication phase analysis

communications currently accumulated. There are two cases when a phase is generated. First, once a *kill_phase* annotation is encountered, which indicates there are phases in the sub–lists, thus, it does not make sense to maintain a phase passing the node since there are phase changes during the execution of the sub–lists, a new phase is created to accommodate the connection requirement after the node. Second, in the cases when adding a new communication pattern into the current (accumulated) pattern exceeds the network capacity, a new communication phase is needed.

Figure 6 describes the algorithm for analyzing the TREE_IF node. The algorithm first computes the phases for the three sub–lists. In the cases when there are phases within the sub–lists and when the network does not have sufficient capacity to support the combined communication, a phase is created in each of the sub–list to accommodate the corresponding communications from that sub–list. Otherwise, the TREE_IF node is annotated with the combined communication indicating the communication requirement of the IF statement. Algorithms to analyze other types of nodes are similar.

Figure 7 shows an example of communication phase analysis. The example contains six communications, $C0$, $C1$, $C2$, $C3$, $C4$, and $C5$, an IF structure and a DO structure. The algorithm first analyzes the sub–lists in the IF and DO structures. Assuming that the com-

Prog.	Description	Distrib.
0001	Solution of 2-D Poisson Equation by ADI	(*, block)
0003	2-D Fast Fourier Transform	(*, block)
0004	NAS EP Benchmark - Tabulation of Random Numbers	(*, block)
0008	2-D Convolution	(*, block)
0009	Accept/Reject for Gaussian Random Number Generation	(block)
0011	Spanning Percolation Cluster Generation in 2-D	(*, block)
0013	2-D Potts Model Simulation using Metropolis Heatbath	(*, block)
0014	2-D Binary Phase Quenching of Cahn Hilliard Cook Equation	(*, block)
0022	Gaussian Elimination - NPAC Benchmark	(*, cyclic)
0025	N-Body Force Calculation - NPAC Benchmark	(block, *)
0039	Segmented Bitonic Sort	(block)
0041	Wavelet Image Processing	(*, block)
0053	Hopfield Neural Network	(*, block)

Table 1: Benchmarks and their descriptions

bination of $C1$ and $C2$ can be supported by the underlying network and that combining communications $C1$, $C2$ and $C3$ exceeds the network capacity, there are two phases in the IF branches and the *kill_phase* is set for the IF header node. Assume also that all communications of $C5$ within the loop can be supported by the underlying network. Figure 7 (a) shows the result after the sub–lists are analyzed. Figure 7 (b) shows the analysis of the main list after sub–lists are considered. The algorithm combines communications $C4$ and $C5$. Since the IF header node is annotated with *kill_phase*. A new phase is generated for communications $C4$ and $C5$ after the IF structure. The algorithm then proceeds to create a phase for communication $C0$. Figure 7 (c) shows the final result.

6 Performance of the compiler algorithms

This section evaluates the performance of the compiler algorithms for compiled communication in *E–SUIF*. We evaluated both the efficiency and the effectiveness of the algorithms. We use the greedy scheduling algorithm [23] in the evaluation and assumed that the underlying network is an 8×8 torus with a maximum multiplexing degree of 10. Note that the network topology, the network size and the multiplexing degree affect the performance of

benchmarks	size (lines)	overall(sec.)	logical(sec.)	physical & phase(sec.)
0001	545	11.33	0.45	8.03
0003	372	24.83	0.50	11.80
0004	599	19.08	0.42	15.02
0008	404	27.08	0.68	13.28
0009	491	46.72	4.45	19.65
0011	439	14.78	0.57	11.37
0013	688	23.08	1.07	17.30
0014	428	15.58	1.03	11.38
0022	496	22.57	0.77	18.35
0025	295	5.77	0.78	3.35
0039	465	16.08	0.38	13.13
0041	579	9.93	0.28	6.62
0053	474	7.39	0.35	4.33

Table 2: Communication phase analysis time

the resource scheduling algorithm, which in turn, affects the communication phase analysis algorithm. The performance results are obtained by running *E–SUIF* on SPARC 5 with 32 MB memory.

We use benchmarks, listed in Table 1, from the HPF benchmark suite [13] at Syracuse University in the evaluation. In the table, the data distributions of the major arrays are obtained from the original benchmark programs. Table 2 breaks down the compiler analysis time (in the unit of seconds), showing the overall compile time, the time for logical communication analysis and the time for physical communication analysis and phase analysis. The time for physical communication analysis and phase analysis accounts for a significant portion of the overall compile time for all the programs. For these small programs the analysis time is not significant. More efficient communication phase analysis algorithms are desirable for large programs.

E–SUIF conservatively estimates the set of physical connections in each phase and uses a resource scheduling algorithm to determine the multiplexing degree needed for each phase. Table 3 shows the precision of the analysis. It compares the average number of connections and the average multiplexing degree in each phase obtained from the compiler with those resulting from actual executions. For most programs, the analysis results match the

benchmark	ave. connections per phase			ave. multiplexing degree		
programs	actual	compiled	percentage	actual	compiled	percentage
0001	564.4	564.4	100%	9.1	9.1	100%
0003	537.6	537.6	100%	8.6	8.6	100%
0004	116.3	116.3	100%	5.5	5.5	100%
0008	562.6	562.6	100%	8.9	8.9	100%
0009	91.2	230.7	39.6%	4.3	6.6	65.1%
0011	126.3	126.3	100%	5.2	5.2	100%
0013	67.3	67.3	100%	3.1	3.1	100%
0014	126.4	126.4	100%	4.0	4.0	100%
0022	13.1	413.2	3%	4.6	8.9	52.7%
0025	80.0	80.0	100%	3.0	3.0	100%
0039	125.7	125.8	99.9%	8.8	8.8	99.9%
0041	556.1	556.1	100%	8.8	8.8	100%
0053	149.2	575.2	25.9%	9.0	9.1	98.9

Table 3: Analysis precision

actual program executions. For the programs where approximations occur, the multiplexing degree approximation is better than the connection approximation as shown in benchmark 0022. This is mainly due to the approximation of the communications that are not vectorized. For such communications, if the underlying network can support all connections in a loop, the phase will contain the loop and will support for all communications in the loop. However, when estimating the average number of connections in each phase, the compiler approximates each individual communication inside the loop with all communications of the loop. Since the multiplexing degree determines the communication performance for a communication pattern, this type of approximation does not hurt the compiled communication performance, although in these cases, dynamic communication may be more efficient than compiled communication.

7 Benefits of compiled communication

In this section, we demonstrate the advantages of compiled communication on optical TDM networks by comparing the performance of compiled communication with that of a traditional dynamic communication method using a number of application programs. The traditional communication method uses a path reservation protocol, called the forward path reservation protocol [25] to reserve a path for each connection before the transmission of data takes place. The forward reservation protocol works as follows. When the source node wants to send data, it first sends a reservation packet towards to destination to establish a lightpath. The reservation fails if the wavelength in some link along the path is not available. Otherwise, the lightpath will be established between the source and the destination, and the destination will notify the source that the lightpath has been established. After that, the source can start sending data.

We use the communication completion time for each communication pattern as the performance metric. For each communication pattern, the communication completion time is defined as the time the program reaches the communication pattern to the time that all messages are received. The runtime communication patterns of a program are obtained by executing the program and collecting the communication traces. These communication patterns are then fed into a cycle level network simulator that simulates both compiled communication and dynamic communication.

We made the following general assumptions in the performance study. The network topology is an 8×8 torus with variable multiplexing degrees. XY routing is used to establish the connections in the torus. We assume that the physical identifiers of the processor follow the row-major numbering and the mapping function between logical processor numbers and physical identifiers is the identity function. Other assumptions that are specific to each experiment will be specified in the experiment.

Two sets of experiments are performed in this study. The first set of experiments uses hand–coded application programs whose communications are highly optimized for the Cray–T3E. These application programs are not HPF–like programs and cannot be handled by the current *E–SUIF*. Thus, this study only demonstrates the potential of compiled communication on these hand–coded programs by assuming a compiler that can accurately analyze the communication requirement of the programs. The second set of experiments uses the

HPF benchmarks described in Section 6 and considers all elements in compiled communication including the imprecision and approximation in communication analysis, connection scheduling and communication phase analysis.

7.1 Hand–coded parallel application programs

This set of programs includes three programs, GS, $TSCF$ and $P3M$. These programs are well designed parallel applications. The communication performance is highly tuned for the Cray–T3E. The GS benchmark uses Gauss–Siedel iterations to solve Laplace equation on a discretized unit square with Dirichlet boundary conditions. The $TSCF$ program simulates the evolution of a self–gravitating system using a self consistent field approach. $P3M$ performs particle–particle particle–mesh simulation. Table 4 describes the static communication patterns that arise in these programs. While GS and TSCF programs contain only one communication pattern each, P3M, which is a much larger program, contains five static communication patterns. All of the above patterns are in the main iterations of the programs.

Pattern	Type	Description
GS	shared array ref.	PEs form a logically linear array, Each PE communicates with two PEs adjacent to it.
TSCF	explicit send/recv	hypercube pattern
P3M 1	data redistrib.	(:block, :block, :block) → (:, :, :block)
P3M 2	data redistrib.	(:, :, :block) → (:block, :block, :)
P3M 3	data redistrib.	(:block, :block, :) → (:, :, :block)
P3M 4	data redistrib.	(:, :, :block) → (:block, :block, :block)
P3M 5	shared array ref.	PEs are logically 3–D array, each PE communicates with 26 PEs surrounding it

Table 4: Communication pattern description.

Table 5 shows the communication time for these communication patterns. Listed in the tables are the size of the problem and the communication times in unit of time slots for compiled communication and dynamic communication. A time slot is assumed to be the time to send a packet of 64 bytes data. For a 100Mbps channel, the duration of the time slot would be the time to transmit the data, $5.12\mu s$, plus a small guard band between time slots. Notice that the software startup overhead at the end hosts is in the order of a few microseconds for small messages in a well tuned messaging system [18], which corresponds to around 1 time slot. In the simulation, we assume that the packet processing time at each node is 1 time slot, which includes the software overheads at the end hosts. For compiled communication, we assume sufficient multiplexing degree to support all the patterns. For dynamic communication, we evaluated the performance of multiplexing degrees of 1, 2, 5 and 10. Notice that compiled communication can use different multiplexing degree at different times for different patterns since the network configuration can change at phase boundary, while dynamic communication can only use a fixed multiplexing degree to handle all communication patterns [19]. The size of the problem affects the message size except for the TSCF program.

Pattern	Problem Size	Compiled Communication	Dynamic Communication			
			$d = 1$	$d = 2$	$d = 5$	$d = 10$
GS	64×64	35	105	118	171	251
	128×128	67	137	154	251	411
	256×256	131	265	304	411	731
TSCF	5120	19	344	268	270	300
P3M 1	$32 \times 32 \times 32$	831	3905	3625	2018	1861
	$64 \times 64 \times 64$	6207	12471	10754	10333	9619
P3M 2,3	$32 \times 32 \times 32$	382	9999	6094	4661	4510
	$64 \times 64 \times 64$	2174	17583	14223	10360	9320
P3M 4	$32 \times 32 \times 32$	457	3309	2356	1766	1722
	$64 \times 64 \times 64$	3369	9161	7674	7805	7122
P3M 5	$32 \times 32 \times 32$	40	583	374	371	480
	$64 \times 64 \times 64$	68	673	457	445	505

Table 5: Communication time for static patterns.

The following observations can be made from the results in Table 5. First, the compiled communication out–performs dynamic communication in all cases. The communication time for dynamic communication was 2 to 30 times greater than that for compiled communication. Larger performance gains are observed for communication with small message sizes (e.g., the TSCF pattern) and dense communication (e.g., the P3M 2 pattern). Second, a large multiplexing degree does not always improve the communication performance for dynamic communication. For example, a multiplexing degree of 1 results in best performance for the pattern in GS. This is because the dynamic control must use a fixed multiplexing degree and is not able to adapt to the optimal multiplexing degree for a given communication pattern.

In this study we observe that for a well designed parallel program, the fine grain communications that result from shared array accesses usually cause sparse communications with small message sizes. For a communication system to efficiently support such communication, the system should have small latency. Optical networks that use dynamic communication incur large startup overhead. Thus, they cannot support this type of communication efficiently. As shown in our simulation results, compiled communication eliminates the startup overhead and performs fine grain communications efficiently. We also observe that data redistributions can result in dense communications with large message sizes. In this case, the control overhead does not significantly affect communication performance. However, dense communication results in more network contentions in the communication system, and the dynamic communication control system may not be able to resolve these contentions efficiently. Our simulation shows that static management of the dense communication patterns results in large performance gains.

7.2 HPF benchmarks

Table 6 shows the communication time (in unit of time slots) of the programs using compiled communication with a maximum multiplexing degree of 10 and dynamic communication with different multiplexing degrees 1, 4, 14, 20. In this experiment, compiled communication out–performs dynamic communication with all multiplexing degrees for most of the programs. The benefits of assigning communication channels statically at compile time and the elimination of the runtime path establishement overhead out–weights the bandwidth losses due to the imprecision of compiler analysis. Performance degradation in compiled communication due to the conservative approximation in compiler analysis is observed in some of the programs. For example, the compiler over–estimates the communication requirement in benchmarks 0009 and 0022. However, since the patterns approximated do not dominate the total communication time, compiled communication still results in lower overall communication time for these two benchmarks. The simple communication phase analysis algorithm implemented in *E–SUIF* always attempts to put as many communications into each phase as possible. This sometimes has negative effects in the communication performance. As can be seen in Table 6, for the benchmark 0013, the compiled communication performs worse than dynamic communication with $d = 1$. The performance of the communication phase analysis can be improved when the communication locality [20] of the program is taken into consideration instead of blindly inserting patterns into each phase.

benchmarks	Compiled Communication	Dynamic Communication			
		$d = 1$	$d = 4$	$d = 14$	$d = 20$
0001	45,624	888,240	357,600	267,360	273,360
0003	752	14,804	5,960	4,456	4,556
0004	1,368	1,920	2,208	3,504	4,224
0008	2,256	44,412	17,880	13,368	13,668
0009	2,394	3,360	3,864	6,132	7,392
0011	105,252	141,052	181,506	372,678	484, 374
0013	166,280	154,080	256,240	779,920	1,086,160
0014	63,400	71,200	125,200	391,200	550,800
0022	3,244,819	6,844,054	6,402,631	6,516,485	6,925,278
0025	29,854	23,440	31,221	61,712	81,958
0039	68,704	115,488	136,390	214,042	261,832
0041	1,504	29,608	11,920	8,912	9,112

Table 6: Communication time for the HPF benchmarks.

In summary, compiled communication achieves high performance for all types of static patterns even after taking the approximations in the compiler analysis algorithms into consideration. Four factors contribute to the performance gain. First, compiled communication eliminates dynamic control overhead. This is most significant for communications with small message sizes. Second, compiled communication takes the whole communication pattern into consideration, while dynamic communication, which considers the connection requests one by one, suffers from the head–of–line effect. Third, the off–line message scheduling algorithm further optimizes the communication efficiency for compiled communication. Fourth, compiled communication allows the system to use various multiplexing degrees for different communication patterns, which allows the system to use optimal multiplexing degree to deal with each particular communication pattern. In dynamic communication, control mechanism for variable multiplexing degrees is too expensive to implement [19]. Fixing the multiplexing degree in dynamic communication performs well for some communication patterns, but poorly for other communication patterns.

8 Conclusion

In this paper, we present the *E–SUIF* compiler that supports compiled communication for all–optical TDM networks, describe the compiler algorithms used in *E–SUIF*, evaluate the compiled algorithms, and compare the performance of compiled communication with that of dynamic communication. Our results show that compiled communication is efficient for all–optical TDM networks. While the techniques in this paper are developed for all–optical TDM networks, compiled communication can also improve the communication performance on other types of networks. Most of the compiler analysis algorithms in *E–SUIF* can also be applied to support compiled communication on other networks. The resource management algorithms, however, need to be tailored for different networks.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam and C. W. Tseng "The SUIF Compiler for Scalable Parallel Machines." *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February, 1995.

[2] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. "The PARADIGM Compiler for Distributed-Memory Multicomputers." in *IEEE Computer*, Vol. 28, No. 10, pages 37-47, Oct. 1995.

[3] C. A. Brackett, "Dense wavelength division multiplexing networks: Principles and applications," *IEEE Journal on Selected Areas of Communications*, Vol. 8, Aug. 1990.

[4] M. Bromley, S. Heller, T. McNerney and G. L. Steele, Jr. "Fortran at Ten Gigaflops: the Connection Machine Convolution Compiler," in *Proc. of SIGPLAN'91 Conf. on Programming Language Design and Implementation.* June, 1991.

[5] D. Callahan and K. Kennedy "Analysis of Interprocedural Side Effects in a Parallel Programming Environment." *J. of Parallel and Distributed Computing*, 5:517-550, 1988.

[6] F. Cappelllo and C. Germain. "Toward high communication performance through compiled communications on a circuit switched interconnection network," in *Proceedings of the Int'l Symp. on High Performance Computer Architecture*, pages 44-53, Jan. 1995.

[7] S. Chakrabarti, M. Gupta and J. Choi "Global Communication Analysis and Optimization." In *Programming Language Design and Implementation*, 1996, pages 68-78.

[8] David Culler, et. al, "The Generic Active Message Interface Specification," *White Paper*, NOW group, UC Berkeley. Available at http://now.cs.berkeley.edu/Papers/Papers/gam_spec.ps.

[9] Thorsten von Eicken, Anindya Basu, Vineet Buch and Werner Vogels "U–Net: A User–Level Network Interface for Parallel and Distributed Computing." *Proc. of the 15th ACM Symposium on Operating Systems Principles*, Dec., 1995.

[10] M. Gupta, E. Schonberg and H. Srinivasan "A Unified Framework for Optimizing Communication in Data-parallel Programs." In *IEEE Trans. on Parallel and Distributed Systems*, Vol. 7, No. 7, pages 689-704, July 1996.

[11] S. Hinrichs. "Compiler Directed Architecture–Dependent Communication Optimization," *Ph.D dissertation*, School of Computer Science, Carnegie Mellon University, 1995.

[12] S. Hiranandani, K. Kennedy and C. Tseng "Compiling Fortran D for MIMD Distributed–memory Machines." *Comm. of the ACM*, 35(8):66-80, August 1992.

[13] "High Performance Fortran Applications (HPFA)." Available at "http://www.npac.syr.edu/hpfa".

[14] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam and N. Shenoy "A Global Communication Optimization Technique Based on Data–Flow Analysis and Linear Algebra." In *the First Merged Symposium IPPS/SPDP*, Orlando, Fl, April 1998.

[15] K. Kennedy, N. Nedeljkovic and A. Sethi "A linear–time algorithm for computing the memory access sequence in data–parallel programs." *Principles & Practice of Parallel Programming*, pages 102–111, 1995.

[16] M. Kumar. "Unique Design Concepts in GF11 and Their Impact on Performance". *IBM Journal of Research and Development*. Vol. 36 No. 6, November 1992.

[17] D. Lahaut and C. Germain, "Static Communications in Parallel Scientific Programs," in *PARLE'94, Parallel Architecture & Languages*, Athen, Greece, July 1994.

[18] S. Pakin, V. Karamcheti and A. A. Chien "Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors." *IEEE Concurrency*, Vol. 5, No. 2, April-June 1997, pages 60-73.

[19] C. Qiao and R. Melhem. "Reducing Communication Latency with Path Multiplexing in Optically Interconnected Multiprocessor Systems," in *IEEE Trans. on Parallel and Distributed Systems*, vol 8, no 2, pp.97-108, 1997.

[20] S. Salisbury, Z. Chen, and R. Melhem, "Modeling Communication Locality in Multiprocessors", *J. of Parallel and Distributed Computing*, vol 56, no 2, pp. 71-98, 1999.

[21] J.M. Stichnoth, D. O'Hallaron and T.R. Gross "Generating Communication for Array Statements: Design, Implementation, and Evaluation." *Journal of Parallel and distributed Computing*, 21(1):150-159, 1994.

[22] R. Subramanian and S. Pande, "Efficient Program Partitioning based on Compiler Controlled Communication", *Fourth International Workshop on High Level Programming Models and Supportive Environments (HIPS'99)*, San Juan, Puerto Rico, April 1999.

[23] X. Yuan, R. Melhem and R. Gupta "Compiled Communication for All–optical TDM Networks," *Supercomputing'96*, Pittsburgh, PA, November 17-22, 1996.

[24] X. Yuan, R. Gupta, and R. Melhem "An Array Data Flow Analysis based Communication Optimizer," *Tenth Annual Workshop on Languages and Compilers for Parallel Computing* (LCPC'97), LNCS 1366, Minneapolis, Minnesota, August 1997.

[25] X. Yuan, R. Melhem and R. Gupta "Distributed Path Reservation Algorithms for Multiplexed All-optical Interconnection Networks." *IEEE Trans. on Computers*, Volume 48, No. 12, pages 1355–1363, Dec. 1999.

[26] X. Yuan, R. Gupta R. Melhem "Compiler Analysis to Support Compiled Communication for HPF-like Programs," *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, April 1999.

[27] Xin Yuan, Rami Melhem and Rajiv Gupta, "Performance of Multi-hop Communications Using Logical Topologies on Optical Torus Networks", *Journal of Parallel and Distributed Computing*, Vol. 61, No. 6, pages 748-766, June 2001.

Biographies

Xin Yuan received a B.S. in Computer Science and Engineering from Shanghai Jiaotong University in 1989, an M.S. degree in Computer Science and Engineering from Shanghai Jiaotong University in 1992, and a Ph.D. degree in Computer Science from the University of Pittsburgh in 1998. He is an assistant professor at the department of Computer Science, Florida State University. His research interests include quality of service routing, optical WDM networks and high performance communication for clusters of workstations. Dr. Yuan is a member of IEEE and ACM.

Rami Melhem received a B.E. in Electrical Engineering from Cairo University in 1976, an M.A. degree in Mathematics and an M.S. degree in Computer Science from the University of Pittsburgh in 1981, and a Ph.D. degree in Computer Science from the University of Pittsburgh in 1983. He was an Assistant Professor at Purdue University prior to joining the faculty of The University of Pittsburgh in 1986, where he is currently a Professor of Computer Science and Electrical Engineering and the Chair of the Computer Science Department. His research interest include Optical Interconnection Networks, Real-Time and Fault-Tolerant Systems, High Performance Computing and Parallel Computer Architectures. Dr. Melhem served on program committees of numerous conferences and workshops and was the general chair for the third International Conference on Massively Parallel Processing Using Optical Interconnections. He was on the editorial board of the IEEE Transactions on Computers and the IEEE Transactions on Parallel and Distributed systems. He is servin on the advisory boards of the IEEE technical committees on Parallel Processing and on Computer Architecture. He is the editor for the Klewer/Plenum Book Series in Computer Science and is on the editorial board of the Computer Architecture Letters. Dr. Melhem is a fellow of IEEE and a member of the ACM.

Rajiv Gupta is a Professor of Computer Science at The University of Arizona. He received the BTech degree (Electrical Engineering, 1982) from the Indian Institute of Technology, New Delhi, India, and the PhD degree (Computer Science, 1987) from the University of Pittsburgh. His primary areas of research interest include performance, power, and memory issues in embedded and superscalar processors, profile guided optimization and program analysis, and instruction level parallelism.

Rajiv received the National Science Foundation's Presidential Young Investigator Award in 1991 and is an IEEE Distinguished Visitor. He is serving as the Program Chair for 2003 SIGPLAN Conference on Programming Language Design and Implementation (PLDI) and 9th International Symposium on High Performance Computer Architecture (HPCA-9). He has served as a program committee member for numerous conferences in the fields of optimizing compilers and computer architecture. He serves as an Associate Editor for Parallel Computing journal and IASTED International Journal of Parallel and Distributed Systems and Networks. Rajiv is a member of ACM and a senior member of IEEE.