Compiler Analysis to Support Compiled Communication for HPF–like programs

Xin Yuan[*]
Dept. Of Computer Science
Florida State University
Tallahassee, FL 32306

Rajiv Gupta and Rami Melhem
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15213

Abstract

By managing network resources at compile time, the compiled communication *technique greatly improves the communication performance for communication patterns that are known at compile time. In order to support compiled communication, the compiler must estimate the runtime physical connection requirement (*physical communication*) of a program and partition the program into phases such that the underlying network can support the communications within each phase. Traditional communication analysis techniques represent the communication requirement in logical forms (*logical communication*) and are insufficient for compiled communication. In this paper, we describe the compiler algorithms that derive physical communications from logical communications and present a communication phase analysis algorithm that partitions a program into phases. These algorithms are implemented and evaluated in our* E–SUIF *compiler.*

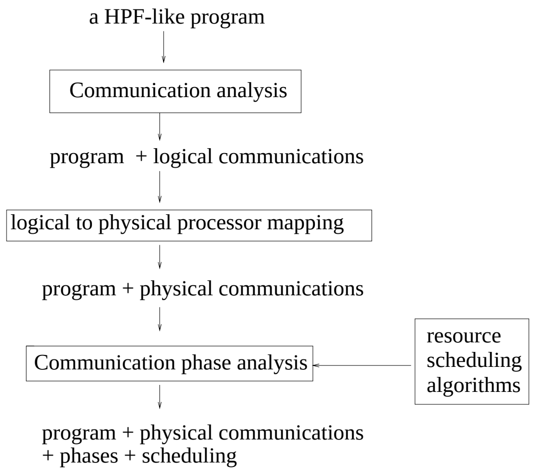
a HPF-like program
Communication analysis
program + logical communications
logical to physical processor mapping
program + physical communications
Communication phase analysis
resource scheduling algorithms
program + physical communications + phases + scheduling

Figure 1. The E–SUIF compiler

1 Introduction

Recently, researchers have shown that the communication performance of dense matrix computation applications can be greatly improved by allowing network resources to be managed by the compiler and using the *compiled communication* technique [2, 5, 7]. In compiled communication, the compiler determines the communication requirement of a program and manages network resources statically to support efficient communications for the program. A number of compiler issues must be addressed in order to apply the compiled communication technique. First, traditional communication analysis techniques [4, 6, 8] represent the communications in logical forms (*logical communications*), such as Available Section Descriptor (ASD) [4], Section Communication Descriptor (SCD) [8] and a linear algebra framework [6]. While these descriptors contain sufficient information for the compiler to perform a number of high–level communication optimizations, they are insufficient for compiled communication which utilizes the knowledge of runtime physical connection requirement (*physical communication*) of a program. Deriving physical communications from logical communications is a non–trivial task and usually requires approximations due to the variables whose values are unknown at the compile time. Second, due to the limited network resources, the compiler must partition a program into phases such that each phase contains fixed, pre-determined communication patterns that the underlying network can support. Since the network must reconfigure at phase boundaries, to obtain high performance, the compiler must incorporate as many communications as possible without exceeding the network capacity in each phase to reduce the number of reconfigurations at runtime.

We have addressed these issues in the *E–SUIF* compiler, an extension of the Stanford SUIF compiler [1]. *E–SUIF* supports compiled communication for HPF–like programs on optical Time–Division–Multiplexing (TDM) networks. While *E–SUIF* targets optical TDM networks, most of the

[*]This work was performed when the author was at the Department of Computer Science, University of Pittsburgh.

techniques for supporting compiled communication can be applied to other types of networks. Figure 1 shows the major components in *E–SUIF*. The first phase in *E–SUIF* is a traditional communication analyzer that analyzes the logical communication requirement of a program and performs a number of high–level communication optimizations. The second phase, *logical to physical processor mapping*, derives physical communications from logical communications. The component *resource scheduling* in *E–SUIF* determines whether a set of physical communications can be supported by the underlying network, and if the communications can be supported, how the network resources are scheduled to support the communications. The third phase, *communication phase analysis*, utilizes the resource scheduling algorithms to partition the program into phases such that communications in each phase can be supported by the underlying network and that there is minimal number of phases at runtime. The *E–SUIF* compiler outputs a program with physical communications, phases and the resource scheduling for each phase. In this paper, we will discuss the techniques used in *E–SUIF*.

The rest of the paper is organized as follows. Section 2 briefly introduces the background. Section 3 describes algorithms to derive physical communications from SCDs. Section 4 presents the communication phase analysis algorithm. Section 5 evaluates the performance of the algorithms and Section 6 concludes the paper.

2 Background

We have previously developed a traditional communication analyzer based on a demand driven dataflow analysis technique [8]. The work in this paper is built on top of the analyzer. In this section, we will describe the programming model and the data flow communication descriptor, Section Communication Descriptor (SCD), used in the analyzer.

2.1 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 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. However, our compiler 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 data space to the corresponding point $\vec{v}$ on 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 $v \ mod \ (p*b)/b$. For cyclic distribution, $b = 1$. For block distribution, $b = N/p$, where N is the size of the dimension. We will use 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.

```
    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
             $C_1 :< y, (i), < (1, i+1, 2) \rightarrow (2*j, i+2, 1), \perp >, \perp >$
(3)          x(i, j) = y(i)...
```

Figure 2. Representing logical communications using SCDs

2.2 Section Communication Descriptor

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 the array name N and the second component is the array region descriptor D. The third component is the communication mapping descriptor CM of the form $< src \rightarrow dst, qual >$, which describes the source–destination relationship of the communication. Finally the fourth component is a qualifier descriptor Q, which specifies the iterations within a loop when the communication is performed. Detail about SCD can be found in [8]. Figure 2 is a simple example showing how SCDs can represent logical communications. Assuming the *owner computes* rule, communication C_1 represents the logical communication required by the statement in line (3), that is, moving array element $y(i)$ from the logical processor that owns $y(i)$ to the logical processor that owns $x(i, j)$. This communication can be vectorized and placed out of the loop. The vectorized communication can be represented as $< y, (1:5:1), < (1, i+1, 2) \rightarrow (2*j, i+2, 1), j = 1:5:1 >, \perp >$.

3 Logical to physical processor mapping

The SCD descriptor represents the communication in a logical form and does not provide sufficient information to perform compiled communication that requires the knowledge of the detailed connection requirement of a program. This section describes 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 and use *processor grid* to denote both physical and logical processor grids.

3.1 One–Dimensional arrays and one–dimensional processor grids

Let us consider the case where the distributed array and the processor grid are one-dimensional. Given a $SCD = < N = A, D, CM = < src \rightarrow dst, qual >, Q >$, let $src = \alpha * i + \beta$ where $\alpha \neq 0$, $dst = \gamma * i + \delta$ where $\gamma \neq 0$, and $qual = \perp$. The cases where $\alpha = 0$, $\gamma = 0$ or $qual \neq \perp$ will be considered later when multi-dimensional arrays and processor grids are discussed. Let the alignment matrix and the offset vector for array A be M_A and v_A and the distribution of the virtual processor template be block–cyclic(b, p). It can be easily shown that the communication of element $A[n]$ requires the connection from physical processor $(M_A * n + v_A)\ mod\ (p * b)/b$ to physical processor $(\gamma * (M_A * n + v_A - \beta)/\alpha + \delta)\ mod\ (p * b)/b$. The physical communication pattern for the SCD can be obtained by considering all elements in D. However, computing the physical communication of a SCD using this brute–force method is both inefficient and, sometimes, infeasible when D cannot be determined at the compile time.

Next we will show that given the block–cyclic(b, p) distribution of the template, examining at most p^2b^2 elements in D is sufficient to determine the physical communication pattern. To illustrate this fact, let us examine the communications in some detail. We will use notation $s \rightarrow d$ to represent a communication from s to d. Let $D = l : u : s$. The source processors of the logical communication can be obtained by mapping D to the virtual processor grid. Since the mapping function is linear, the set of source processors can be represented as a triple $vs_l : vs_u : vs_s$, that is, $\{vs_l,\ vs_l + vs_s,\ vs_l + 2 * vs_s,\ ...,\ vs_u\}$. The corresponding destination can be computed by solving the equations $vs_l + i * vs_s = CM.src$, where $i = 0, 1, 2, ...$, and replacing the solution in $CM.dst = \gamma * i + \delta$. Hence, the destination processors on the virtual processor grid can also be represented as a triple $vd_l : vd_u : vd_s$, where $vd_l = \gamma * ((vs_l - \beta)/\alpha) + \delta$, $vd_u = \gamma * ((vs_u - \beta)/\alpha) + \delta$ and $vd_s = \gamma * vs_s/\alpha$. Thus, communications on the virtual processor grid can be represented as $vs_l \rightarrow vd_l : vs_u \rightarrow vd_u : vs_s \rightarrow vd_s$, that is, by the set

$$\{vs_l \rightarrow vd_l, vs_l + vs_s \rightarrow vd_l + vd_s, ..., vs_u \rightarrow vd_u\}.$$

Communications on physical processors are obtained by mapping virtual processors onto physical processors. Let the data distribution of the template to be block–cyclic(b, p). Let a point e in the virtual processor grid correspond to (pp, o), where $pp = e\ mod\ (p * b)/b$ is the physical processor that contains e, and $o = e\ mod\ b$ is the offset of e within a block in the processor. Let (pp_k, o_k) correspond to $l + k * s$, where $k = 0, 1, ...$ It can be easily shown that

$$pp_i = pp_j \wedge o_i = o_j \Rightarrow pp_{i+1} = pp_{j+1} \wedge o_{i+1} = o_{j+1}$$

In the (pp, o) space, there are p choices for pp and b choices for o. Thus, there exists a k, $k \leq p * b$, such that $pp_k = p_0$ and $o_k = o_0$, which is a repetition point.

To map logical communication $\{vs_l \rightarrow vd_l, vs_l + vs_s \rightarrow vd_l + vd_s, ..., vs_u \rightarrow vd_u\}$ to physical communication, let (spp_i, so_i) correspond to $vs_l + i * vs_s$ and (dpp_i, do_i) to $vd_l + i * vd_s$. Following previous discussion, there are p choices for *spp* and *dpp*, and b choices for so and do. There exists k, $k \leq p^2b^2$, such that both source and destination processors, and thus the communication pattern, will repeat themselves at the kth element in the data region where all four components in $(spp, so) \rightarrow (dpp, do)$ repeat. The following lemma summarizes this conclusion.

Lemma: Let the template be distributed using block–cyclic(b, p) and $SCD = < A, D = l : u : s, CM = < src \rightarrow dst, qual >, Q >$. Assuming that 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 connection requirement as the communication for $A[l + (m - k) * s]$.

Proof: Follows from above discussions. □

The implication of the lemma is that the algorithm to determine the communication pattern for a SCD can stop when the repetition point occurs. Figure 3 shows the algorithm. The algorithm first checks whether the SCD can be processed. If the SCD does not contain sufficient information, the physical communication is approximated by the All–to–All communication. Otherwise, the algorithm will consider each element in D until the repetition point is found or all elements in D are considered. The algorithm has a time complexity of $O(p^2b^2)$ and can be easily extended to handle the case when the source array has different distribution from the destination array.

3.2 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 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 connections.

The algorithm first checks 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

```
Compute_1D_pattern($D$, $CM.src$, $CM.dst$)

  Let $D = l : u : s$
  Let $CM.src = \alpha * i + \beta$. $CM.dst = \gamma * i + \delta$
  if ($l$, $\alpha$, $\beta$, $\gamma$ or $\delta$ contain variables) then
    return all–to–all connections
  end if
  if ($s$ contains variables) then $s = 1$
  $pattern = \phi$
  for each element $i$ in $D$ do
    $pattern = pattern + comm.\ of\ element\ i$
    if (communication repeated) then
      return $pattern$
    end if
  end for
```

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

```
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_1D_pattern($D[i]$,
        $CM.src[sd]$, $CM.dst[dd]$)
    else
      1dpattern = compute_1D_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_1D_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

pattern for the SCD is approximated by all–to–all connections. If the algorithm can determine the correlation, it computes the communication patterns for all the correlated dimensions using the algorithm to compute communication patterns for 1–dimensional arrays. Other dimensions in the source processor grid are mapped either to a constant processor or to a range of processors that can be determined by the array elements involved, while other dimensions in the destination processor grid are mapped either to a constant processor or to a range of processors that can be determined by the mapping qualifier, $CM.qual$. The algorithm considers all these cases and cross–products all the patterns in different dimensions to obtain the final communication pattern.

Consider the physical communication for the vectorized communication, $< y, (1 : 5 : 1), < src = (1, i + 1, 2), dst = (2 * j, i + 2, 1), qual = \{j = 1 : 5 : 1\} >, \perp>$ in 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)). From the array alignment, the algorithm determines that dimension 1 in the 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 $\{0 \rightarrow 1, 1 \rightarrow 0, 0 \rightarrow 0, 1 \rightarrow 1\}$ with $sd = 1$ and $dd = 1$ is obtained. Thus the communication list becomes $\{(*, 1, *) \rightarrow (*, 0, *), (*, 0, *) \rightarrow (*, 1, *), (*, 1, *) \rightarrow (*, 1, *), (*, 0, *) \rightarrow (*, 0, *)\}$ 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 $\{(0, 1, 1) \rightarrow (*, 0, *), (0, 0, 1) \rightarrow (*, 1, *), (0, 0, 1) \rightarrow (*, 0, *), (0, 1, 1) \rightarrow (*, 1, *)\}$. Considering the $qual$ in CM, the dimension 0 of the destination processor can be either 0 or 1. The dimension 2 of the destination is always mapped to processor 0. Thus, the final physical communication is given by $\{(0, 1, 1) \rightarrow (0, 0, 0),\ (0, 1, 1) \rightarrow (1, 0, 0),\ (0, 1, 1) \rightarrow (0, 1, 0),\ (0, 1, 1) \rightarrow (1, 1, 0),\ (0, 0, 1) \rightarrow (0, 0, 0),\ (0, 0, 1) \rightarrow (1, 0, 0),\ (0, 0, 1) \rightarrow (0, 1, 0),\ (0, 0, 1) \rightarrow (1, 1, 0)\}$.

4 Communication Phase analysis

The communication phase analysis is carried out in a recursive manner on the high level SUIF representation of a program [1]. SUIF represents a program in a hierarchical manner. A SUIF representation of a program contains a list of nodes, which may in turn contain sub–lists. The nodes that contain sub–lists are called *composite nodes*. The communication phase analysis algorithm associates two variables, $pattern$, which represents the communication pattern that is exposed from the sub–lists, and $kill\_phase$, which indicates whether the sub–lists contain phases, for each *composite node*.

```
Communication_Phase_Analysis(list)
Input: $list$: a list of SUIF nodes
Output: $pattern$: comm. pattern exposed out of the list
          $kill\_phase$: phases within the list?

Analyze comm. phases for the sub–lists for each node.
$c\_pattern = \perp, kill\_phase = 0$
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 $n$.
      $c\_pattern = \perp, kill\_phase = 1$
    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

The algorithm to analyze communication phases in a program (a list of SUIF nodes) is shown in Figure 5. The algorithm takes advantage of the assumption that the program is a structured program and computes the data flow information through bottom–up traversal of program structures. Thus, it 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 bottom–up traversal of the SUIF program accumulates the communications in the innermost loops first, and thus captures the communication locality. After all sub–lists are analyzed, the program becomes a straight line program, whose nodes are annotated with physical communications, $pattern$ and $kill\_phase$. The algorithm examines all these annotations in each node from back to front accumulating communications. When the accumulated communications exceed network capacity, a phase will be generated. Another case to create a phase is when a $kill\_phase$ annotation is encountered, which indicates there are phases in the sub–lists.

Figure 6 shows an example of the communication phase analysis. The program in the example contains six communications, $C0$, $C1$, $C2$, $C3$, $C4$, and $C5$, an IF structure and a DO structure. The communication phase analysis algorithm first analyzes the sub–lists in the IF and DO

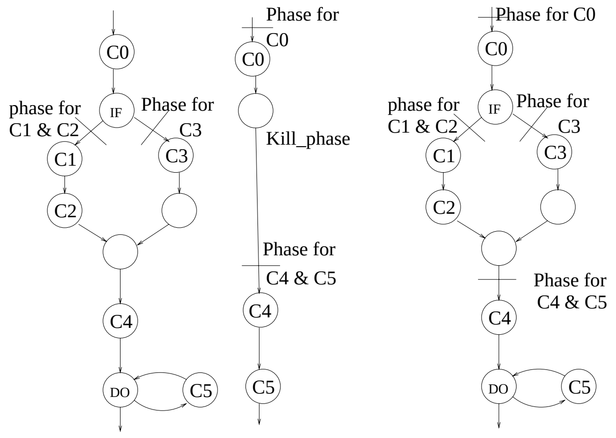

Figure 6. An example

structures. Assuming the combination of $C1$ and $C2$ can be supported by the underlying network, while combining communications $C1$, $C2$ and $C3$ exceeds the network capacity, which results in the two phases in the IF branches and the $kill\_phase$ is set for the IF header node. Assuming that all communications of $C5$ within the DO loop can be supported by the underlying network, Figure 6 (a) shows the result after the sub–lists are analyzed. The algorithm analyzes the list by considering each node and 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 6 (c) shows the final result of the communication phase analysis for this example.

5 Performance evaluation

This section evaluates the performance of the compiler algorithms. In the evaluation, we assume that the underlying network is a 8×8 torus with a maximum multiplexing degree of 10. That is, each link in the network can support up to 10 channels.

Programs from the HPF benchmark suite at Syracuse University [3] are used to evaluate the algorithms. The benchmarks and their descriptions are listed in Table 1. Table 2 breaks down the compiler analysis time (in the unit of seconds). The table shows the overall compile time, the time for logical communication analysis and the time for physical communication analysis and phase analysis. The overall compile time includes the time to load and store the program in addition to the analysis time. The time for physical communication analysis and phase analysis accounts for a significant portion of the overall compile time for all the

Prog.	Description
0013	2-D Potts Model Simulation using Metropolis Heatbath
0014	2-D Binary Phase Quenching of Cahn Hilliard Cook Equation
0022	Gaussian Elimination - NPAC Benchmark
0025	N-Body Force Calculation - NPAC Benchmark
0039	Segmented Bitonic Sort
0041	Wavelet Image Processing
0053	Hopfield Neural Network

Table 1. Benchmarks and their descriptions

prog.	size (lines)	all (sec)	logical (sec)	physical & phase (sec)
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

programs. However, for small size programs as the benchmarks used, the analysis time is not significant.

E–SUIF estimates the set of physical connections in each phase and uses a resource scheduling algorithm to determine the multiplexing degree needed in the network to support all connections in the phase simultaneously. 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 in actual executions. The number of connections and the multiplexing degree in each phase during execution are obtained by accumulating the connections within each phase at runtime. For most programs, the analysis results match the actual program executions. For the programs where approximations occur, the approximation for the multiplexing degree is more precise than the approximation for the number of connections as shown in benchmark 0022. Since the multiplexing degree determines the communication performance for a communication pattern, the approximation for the multiplexing degree is more important for compiled communication on optical TDM networks.

6 Conclusion

In this paper, we have presented the compiler analysis technique used in the *E–SUIF* compiler to support compiled communication. Specifically, we described algorithms that computes physical communications from the Section Communication Descriptors (SCDs), which represent logical communications, presented a communication phase analysis algorithm that partitions a program into phases so that the communications in each phase can be supported by the underlying network, and evaluated these algorithms. Our results show that the compiled communication technique can be efficiently implemented.

prog.	ave. conn. per phase			ave. degree		
	act.	comp.	percent	act.	comp.	percent
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

References

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

[2] 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.

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

[4] 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.

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

[6] 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.

[7] X. Yuan "Dynamic and Compiled Communication in Optical Time-Division-Multiplexed Point-to-point Networks." *Ph.D dissertation*, Dept. of Computer Science, University of Pittsburgh, 1998.

[8] 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.