Global Communication Analysis and Optimization

Soumen Chakrabarti* Manish Gupta† Jong-Deok Choi†

Abstract

Reducing communication cost is crucial to achieving good performance on scalable parallel machines. This paper presents a new compiler algorithm for global analysis and optimization of communication in data-parallel programs. Our algorithm is distinct from existing approaches in that rather than handling loop-nests and array references one by one, it considers all communication in a procedure and their interactions under different placements before making a final decision on the placement of any communication. It exploits the flexibility resulting from this advanced analysis to eliminate redundancy, reduce the number of messages, and reduce contention for cache and communication buffers, all in a unified framework. In contrast, single loop-nest analysis often retains redundant communication, and more aggressive dataflow analysis on array sections can generate too many messages or cache and buffer contention. The algorithm has been implemented in the IBM pHPF compiler for High Performance Fortran. During compilation, the number of messages per processor goes down by as much as a factor of nine for some HPF programs. We present performance results for the IBM SP2 and a network of Sparc workstations (NOW) connected by a Myrinet switch. In many cases, the communication cost is reduced by a factor of two.

1 Introduction

Distributed memory architectures provide a cost-effective method of building scalable parallel computers. However, the absence of global address space, and the resulting need for explicit message passing makes these machines difficult to program. This has motivated the design of languages like High Performance Fortran (HPF) [9], which allow the programmer to write sequential or shared-memory parallel programs that are annotated with directives specifying data decomposition. The compilers for these languages are responsible for partitioning the computation, and generating the communication necessary to fetch values of non-local data referenced by a processor [15, 30, 4, 3, 5, 12].

Accessing remote data is usually orders of magnitude slower than accessing local data. This gap is growing because CPU performance is out-growing network performance, CPU's are running relatively independent multiprogrammed operating systems, and commodity networks are being found more cost-effective. As a result, communication startup overheads tend to be astronomical on most distributed memory machines, although reasonable bandwidth can be supported for sufficiently large messages [25, 24]. Thus compilers must reduce the *number* as well as the *volume* of messages. This can improve performance on shared memory machines as well, because fewer messages translate into fewer synchronization events [26, 22, 13].

Consequently, communication optimization has been extensively researched, from local single loop-nest to global and even interprocedural optimizations. The earliest and most commonly used optimizations include message vectorization [15, 30], using collective communication [11, 20], message coalescing [15], and exploiting pipelined communication [15, 12], all within the scope of a single loop nest. Local analysis of array accesses based on dependence testing alone often retains redundant communication. Naturally, the next step was the use of dataflow analysis, e.g.. using precise array dataflow analysis to detect redundant communication within a loop nest [3], and those using global dataflow analysis for redundancy elimination across loop nests as well. These include dataflow analysis over array sections for regular computations [10, 14, 17, 18] and over entire arrays for irregular computations [27, 1]. Typically, the technique is to move communication to the earliest possible point dictated by data dependency and control flow. Superficially, this appears to give the additional benefit of maximum overlap between CPU and network activity. Recently, it has been pointed out that communication that is scheduled too eagerly can lead to problems like contention (which reduce effective network bandwidth) and excessive buffer requirement (which upsets the computation's cache and thereby degrades performance) [18, 21]. This is similar to the issue of register pressure [19]. However, a more striking fact we point out is that earliest placement can also lead to valuable opportunities being missed for reducing the number of messages or eliminating partial redundancy, making it a sub-optimal strategy even in the absence of resource constraints.

There is thus a clear need for global *scheduling* of communication. In this paper we present a novel compiler algorithm that includes and extends all the optimizations mentioned above. Our algorithm derives from static single assignment analysis, array dependence analysis, and the recently introduced data availability analysis [14], which is extended to detect compatibility of communication patterns in addition to redundancy. We differ significantly from existing research in that the position of communication code for each remote access is not decided independent of other remote accesses; instead, the positions are decided in an interdependent and global manner. The algorithm achieves both redundancy elimination and message combining globally, and is able to reduce the number of messages to an extent that is not achievable with any previous approach.

*Computer Science Division, U. C. Berkeley, CA 94720. Partly supported by ARPA/DOD (DABT63-92-C-0026), DOE (DE-FG03-94ER25206), and NSF (CCR-9210260, CDA-8722788 and CDA-9401156). Part of the work was done while the author was visiting IBM Research. Email: soumen@cs.berkeley.edu.

†IBM T. J. Watson Research Center, Yorktown Heights, P. O. Box 704, NY 10598. Email: {mgupta,jdchoi}@watson.ibm.com.

Our algorithm has been implemented in the IBM pHPF prototype compiler [12]. We report results from a preliminary study of some well-known HPF programs. The performance gains are impressive. Reduction in static message count can be up to a factor of almost nine. Time spent in communication is reduced in many cases by a factor of two or more. We believe that these are also the first results from any implementation of redundant message elimination across different loop nests, and add significant experimental experience to research on communication optimization.

2 Motivating codes

We motivate the need for our proposed optimizations by analyzing a series of real-life F90/HPF source codes. Specifically, we demonstrate the following.

- Redundancy elimination is useful, but often not enough to reduce the number of messages. This is crucial for our target architectures, especially for synchronous and collective communication.
- In fact, the traditional mechanisms of redundancy elimination can sometimes *prevent* the compiler from generating the best communication code.
- The well-known redundancy elimination technique of earliest communication placement is sensitive to minor syntactic differences in the high-level source, and may produce suboptimal code.

In the code fragments that we present, we will elide actual operations and show each RHS as a list of variables accessed. Frequently we deal with F90 style shift operations that involve nearest-neighbor communication (NNC); we show this pictorially using arrows. For simplicity, the combinable messages in our examples have identical patterns on the processor template; in reality, combining is feasible when one pattern is a subset of another.

2.1 Beyond redundancy elimination

Redundancy elimination seeks to avoid unnecessary repetitions of communication for the same data. Often programs exhibit similar communication patterns involving different data as well. Combining those communications to use fewer messages is a crucial goal on current multicomputers like the SP2 as the message startup costs are large.

In the NPAC `gravity` code[1], all 2-d arrays are of dimension `(ny,nz)` distributed `(BLOCK, BLOCK)` and all 3-d arrays are of dimension `(nx,ny,nz)` distributed `(*,BLOCK,BLOCK)`. A simplified form of the code is shown in Figure 1. In this code it is easy to detect that the NNC for `g` and `glast` can be combined, as can be the `sum` operations. Thus we can combine the eight NN messages into four and the eight global sums into two parallel sets of four global sums.

2.2 Earliest placement may hurt

While in the last example we merely needed a combining pass after earliest placement, in general the situation could be more complicated. In particular, redundancy elimination via earliest placement can prevent the combining possibilities from being exploited. To demonstrate this, we study the NCAR shallow water code, which has NN message pattern. As discussed in [12], the message coalescing optimization implemented in the pHPF compiler allows the diagonal communication to be subsumed by an augmented form of the NNC along the two axes. A simplified form of the original code is shown in Figure 2. If no redundant message elimination is done across different loop nests, there would be 20 exchanges generated per processor, following the subsumption of diagonal communication by message-coalescing. Earliest placement will move up a communication as far as possible within the loop, communicating data right after definition. 14 array sections will be communicated per processor. In contrast, using message combining as the guiding profit motive, we get a schedule with only 8 exchanges per processor, in which placement of communication is not at the earliest point detected by dataflow analysis. (The IBM compiler already optimizes diagonal communication like `p`$\nearrow$ by subsuming it using `p`$\uparrow$ and `p`$\rightarrow$. This is reflected by the message counts.)

2.3 Syntax sensitivity

Since earliest placement pushes communication close to the production of the data value, placement is sensitive to the structure of intervals containing the production. As a case in point, consider the semantically equivalent codes in the three columns of Figure 3. Suppose arrays a and b have identical layout, say blocked. Using earliest placement, the messages for the two arrays can be combined in the third column whereas they cannot in the second code. Even if the programmer were careful enough to write the code in the first column, intermediate passes of compilation may destroy the interval containment. In fact, the current IBM HPF scalarizer [12] will translate the F90-style source to the scalarized form in the second column. If loop fusion can be performed before this analysis, as in this case, the problem can be avoided. But this is not always possible [28, § 9.2]. Thus, limited communication analysis at a single loop-nest level or a rigid placement policy may not work well. Our framework, by not relying on any restricted placement (like earliest or latest) but evaluating many choices globally, proves to be a much more robust strategy that is not easily perturbed by minor syntactic differences.

3 Network performance

By profiling our target networks, we justify why global message scheduling is necessary, and what reasonable simplifying assumptions can be made about the optimization problem. We pick two platforms: the IBM SP2 with a custom network, and a network of Sparc workstations (NOW) connected by a commodity network (Myrinet). IBM's message passing library MPL and MPICH[2] are used for communication. Details of the networks can be found in [25, 24, 16]. We want to measure how large messages are rewarded by the network, while estimating the local buffer-copy cost to collect small messages. Figure 5 shows the profiling code and results.

The top curve shows the bandwidth of local `bcopy` as a function of buffer size. The bottom curve plots network bandwidth as a function of message length, based on the time that the receiver waits for completion.

As long as the buffers fit in cache, we can ignore the overhead of `bcopy`. Fortunately, most of the message startup amortization benefits occur at message sizes much smaller

[1] URL `http://www.npac.syr.edu/hpfa`.

[2] MPICH is an implementation of the MPI standard.

```
Timestep loop:
    glast(:,:) = g(1,:.:)
    for i = 2 to nx − 1
        ··· = g(i,:,:)↑←↓→
        ··· = sum(g(i,ny,:)), sum(g(i,ny − 1,:)), sum(g(i,0,:)), sum(g(i,1,:))
        ··· = glast(:,:)↑←↓→
        ··· = sum(glast(ny,:)), sum(glast(ny − 1,:)), sum(glast(0,:)), sum(glast(1,:))
        glast(:,:) = g(i,:,:)
        g(i,:,:) = ···
```

Figure 1: A simplified form of the NPAC `gravity` code illustrating the need for combining communication.

Source code	Earliest placement	Combined placement
Loop:	Loop:	Loop:
	<u>COMM →p,v ↑p,u ←u ↓v</u>	<u>COMM →p,v ↑p,u ←u ↓v</u>
`cu = p→`	`cu = p→`	`cu = p→`
	<u>COMM ←cu ↑cu</u>	
`cv = p↑`	`cv = p↑`	`cv = p↑`
	<u>COMM →cv ↓cv</u>	
`z = u↑, v→, p→, p↑, p↗`	`z = u↑, v→, p→, p↑, p↗`	`z = u↑, v→, p→, p↑, p↗`
	<u>COMM ↓z ←z</u>	
`h = u←, v↓`	`h = u←, v↓`	`h = u←, v↓`
	<u>COMM →h ↑h</u>	<u>COMM →cv,h ↓z,cv ←z,cu ↑cu,h</u>
`unew = z↓, h→, cv→, cv↓, cv↘`	`unew = z↓, h→, cv→, cv↓, cv↘`	`unew = z↓, h→, cv→, cv↓, cv↘`
`vnew = z←, h↑, cu←, cu↑, cu↖`	`vnew = z←, h↑, cu←, cu↑, cu↖`	`vnew = z←, h↑, cu←, cu↑, cu↖`
`pnew = cu←, cv↓`	`pnew = cu←, cv↓`	`pnew = cu←, cv↓`

Figure 2: The NCAR `shallow` benchmark illustrating that redundancy elimination via earliest placement may lose valuable opportunity for message combining.

F90 Source code	Scalarized code	Hand coded F77
	do $i = 1 : n$	
	$a(i) = 3$	do $i = 1 : n$
distribute a, b, c :: (BLOCK)	<u>COMM Earliest(a)</u>	$a(i) = 3$
$a = 3$	do $i = 1 : n$	$b(i) = 4$
$b = 4$	$b(i) = 4$	<u>COMM Earliest(a), Earliest(b)</u>
$c(2 : n) = a(1 : n - 1) + b(1 : n - 1)$	<u>COMM Earliest(b)</u>	do $i = 2 : n$
	do $i = 2 : n$	$c(i) = a(i - 1) + b(i - 1)$
	$c(i) = a(i - 1) + b(i - 1)$	

Figure 3: Syntax sensitivity of earliest placement.

S	Source code	CommSet(S) after			
		Candidate marking	Subset elimination	Redundancy elimination	Earliest placement
	distribute a, b, c, d :: (BLOCK,*)				
1	$b(:, 1 : n : 2) = 1$	b_1			b_1
2	$b(:, 2 : n : 2) = 2$	b_1, b_2			b_2
3	if (cond)				
4	$a = 3$				a_2
5	else				
6	$a = 4$				a_2
7	endif	a_1, a_2, b_1, b_2	a_1, a_2, b_1, b_2	$\{a_2, b_2\}$	
8	do $i = 2 : n$				
9	do $j = 1 : n : 2$				
10	$c(i, j) = a(i - 1, j) + b(i - 1, j)$ // use a_1, b_1				
11	do $j = 1 : n$				
12	$d(i, j) = a(i - 1, j) - b(i - 1, j)$ // use a_2, b_2				

Figure 4: Running example for analysis and optimization steps. Code for each communication entry is executed *after* executing the statement. The notation $\{a_2, b_2\}$ means the messages for these accesses can be combined. The results of traditional earliest placement is shown in the last column for comparison.

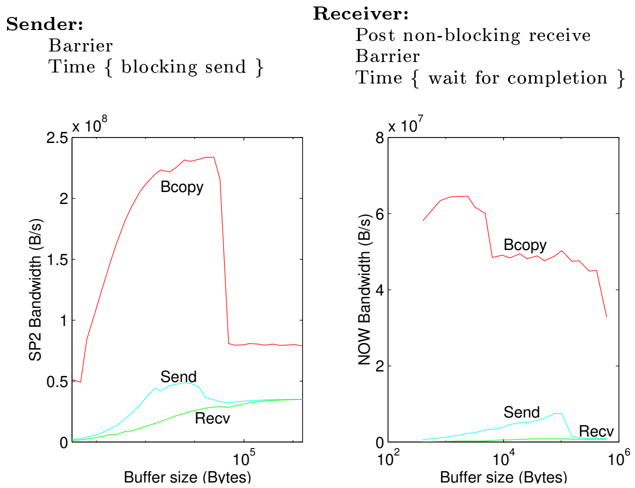


Figure 5: Buffer copying and network bandwidth studies on the IBM SP2 using MPL and the Berkeley NOW using MPICH. The x-axis is to a log scale.

the cache limit, for both machines. Given typical cache sizes, we believe this is a fairly general feature. It may be important to suppress combining communication from different large non-contiguous array sections. E.g., for the SP2, `bcopy` bandwidth is barely twice message bandwidth beyond cache size.

The middle curve shows bandwidth computed using the time the sender takes to inject the message. While the injection bandwidth is much lower than `bcopy`, it is larger than receive bandwidth for certain message sizes. Our algorithm permits additional techniques like Give-n-Take to be used to overlap this latency with code at the sender [27]. We did not implement this because the potential gain was not clear in our architectures and bulk-synchronous SPMD model. It also depends on the co-processor and network software. E.g., the implementors of MPL minimize co-processor assistance because it is much slower than the CPU, and the channel between the CPU and the co-processor is slow [24].

4 Compiler algorithms

In this section we describe the algorithm for placing communication code. This analysis is done after the compiler has performed transformations like loop distribution and loop interchange to increase opportunities for moving communication outside loops [12].

1. For each RHS expression that may need communication, identify the earliest (§4.3) and latest (§4.2) safe position to place that communication. This is typically done using a backward and forward dataflow approach with array section descriptors or bit-vectors. We find it more efficient to exploit the SSA def-use information already computed in an earlier phase [8, 6], refined by array dependence-testing [29].

2. For each non-local reference, identify a set of candidate positions, any one of which can be potentially chosen as the final point to emit a call to a message-passing runtime routine (§4.4).

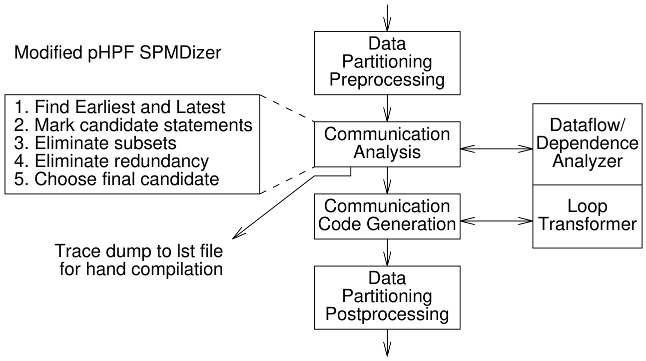


Figure 6: Prototype modifications to the IBM pHPF SPMDizer.

3. Perform the "array-section" analog of common subexpression elimination: detect and eliminate subsumed communication (§4.6).

4. For the remaining communication, choose one from the set of candidate placements. In the prototype we do this in two substeps that will be explained later (§4.5 and §4.7).

The above algorithms have been added to a prototype version of the pHPF compiler as shown in Figure 6. Throughout this section, we will use the code in Figure 4 as a running example to illustrate the operation of the steps of the algorithm.

4.1 Representation and notation

We represent the program using the *augmented* control flow graph (CFG), which makes loop (interval) structure more explicit than the standard CFG by placing *preheader* and *postexit* nodes [2, 23]. These extra nodes also provide convenient locations for summarizing dataflow information for the loop.

The CFG is a directed graph where each *node* is a basic block, a sequence of statements without jumps. A statement S may have a use u or def d of an array variable. d can be either a "regular" def found in the source code, or a ϕ-def inserted during conversion to SSA form. All regular array defs are *preserving*. We refer interchangeably to a use, def, statement, or the node containing them. The node containing S is called $\mathsf{CfgNode}(S)$. When we say communication is placed *at d* we mean immediately *after d*.

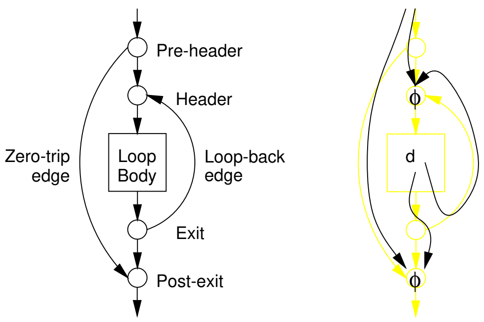


Figure 7: The augmented control flow graph.

A *path* $\pi : v_0 \xrightarrow{+} v_j$ from v_0 to v_j is a non-empty node sequence (v_i) with edges (v_{i-1}, v_i), $1 \leq i \leq j$; we also call π a backward path or *backpath* from v_j to v_0. π *bypasses* v if v does not occur on π. Possibly empty paths are denoted $v_0 \xrightarrow{*} v_j$. Two paths are *non-overlapping* if they are node-disjoint. Non-empty paths $\pi_1 : v_0 \xrightarrow{+} v_j$, $\pi_2 : w_0 \xrightarrow{+} w_k$ converge at z if $v_0 \neq w_0$, $v_j = z = w_k$, and $(v_p = w_q) \Rightarrow (p = j \vee q = k)$.

Loops are named L. Every loop has a well-defined *nesting level* called $\mathsf{NL}(L)$: this is the number of loops containing it. $\mathsf{NL}(v)$ for node v is defined likewise. L or v is *deep* or *shallow* according as NL is large or small. The common nesting level $\mathsf{CNL}(u, v)$ of two nodes u and v is the NL of the deepest loop containing them both. Every loop L has a single *preheader* node, $\mathsf{PreHdr}(L)$, and there is an edge from $\mathsf{PreHdr}(L)$ to $\mathsf{Hdr}(L)$, the *header* node. $\mathsf{PreHdr}(L)$ dominates all nodes in L. There is a *postexit* node for each distinct loop exit target. Each postexit node of L has an incoming edge, called *zero-trip* edge, from $\mathsf{PreHdr}(L)$ (along with the original loop-exiting edges). See Figure 7.

L has a ϕ-def at $\mathsf{Hdr}(L)$, called ϕ_{Hdr}, for each variable defined in the loop or in a loop transitively nested in L. ϕ_{Hdr} has two parameters, r_{pre} and r_{post}, such that there exists a backpath from r_{pre} to `ENTRY` that bypasses all nodes in the loop, and there exists a path from any node in the loop to r_{post} which never takes an exit edge out of the loop.

Each postexit node of a loop L has a ϕ-def, called ϕ_{Exit}, for each variable defined in the loop or in a loop transitively nested in it. Because of ϕ_{Exit}, a definition d can reach a use u only through a definition d' at a level $\mathsf{CNL}(d, u)$. d' can possibly be d only if $\mathsf{CNL}(d, u) = \mathsf{NL}(d)$; otherwise, d' is a ϕ-def at a level $\mathsf{CNL}(d, u)$.

4.2 Identifying the latest position

We describe how the compiler finds $\mathsf{Latest}(u)$, the latest point to place communication for u which is as shallow as possible. This follows from standard communication analysis in which communication is placed just before the outermost loop in which there is no true dependence on u, and is placed just before the statement containing u if no such loop exists [30, 15, 12].

Given a use u, let d range over the reaching regular defs of u. Consider some d. Observe that it is never necessary to place communication for u deeper than at $\mathsf{CNL}(d, u)$. Given d and u, we can compute all possible direction vectors (each is a $\mathsf{CNL}(d, u)$-dimensional vector) [28]. These vectors are used in IsArrayDep in Figure 8(d). Let $\mathsf{DepLevel}(d, u) = \max_\ell\{\mathsf{IsArrayDep}(d, u, \ell)\}$.

Because of the dependency at level $\mathsf{DepLevel}(d, u)$, communication for u cannot be moved outside loop level $\mathsf{DepLevel}(d, u)$. The overall communication level for use u, denoted $\mathsf{CommLevel}(u)$, is set to $\max_d\{\mathsf{DepLevel}(d, u)\}$. Finally, to place communication, we check $\mathsf{CommLevel}(u)$: if $\mathsf{CommLevel}(u) = \mathsf{NL}(u)$, communication is placed immediately before the statement containing u[3]; if $\mathsf{CommLevel}(u) < \mathsf{NL}(u)$, communication is placed in the loop preheader of the loop at level $(\mathsf{CommLevel}(u) + 1)$ that contains u. Note that $\mathsf{CommLevel}(u) > \mathsf{NL}(u)$ is not possible, and that by construction $\mathsf{Latest}(u)$ dominates u.

[3] In this case no vectorization has been possible.

```
a. Earliest(u)
     For each def d of use u in depth-first preorder traversal:
       If Test(d, u) then return d.
--------------------------------------------------------------
b. Test(d, u)
     If d is a φ-def, say d = φ(..., r_i, ...)
       For each φ-parameter r_i
         visit[·] = 0, visit[d] = 1
         Let c_i = Rcount(Reaching(r_i), u, CNL(d, u), visit)
       If two or more c_i's are positive
         Return TRUE
     else (d is a regular def)
       If IsArrayDep(d, u, CNL(d, u))
         Return TRUE.
--------------------------------------------------------------
c. Rcount(d, u, l, visit)
     If d is a φ-def, say d = φ(..., r_i, ...)
       If visit[d] return 0
       visit[d] = 1
       Return Σ_i Rcount(Reaching(r_i), u, l, visit)
     else (d is a regular def)
       If IsArrayDep(d, u, l)
         Return 1
       else if d is a preserving def
         Return Rcount(Reaching(d), u, l, visit)
       else return 0.
--------------------------------------------------------------
d. IsArrayDep(d, u, ℓ)
     If d is the pseudo-def at ENTRY then return TRUE
     If ℓ > CNL(d, u) then return FALSE
     If ∃ direction vector v⃗ = (v_1, ..., v_CNL(d,u)) such that
       • v_i = 0, for i ∈ {1, ..., ℓ − 1}, and
       • v_ℓ ≥ 0
     then return TRUE
     else return FALSE
```

Figure 8: (a) Pseudocode for iterating over reaching defs of u. (b) Pseudocode for testing a def d to identify if d is the earliest communication placement point. (c) Pseudocode for recursively counting the number of incoming edges at ϕ-defs or preserving regular defs that bear possible dependences. (In our SSA implementation, there is a pseudo-def at `ENTRY` for each variable accessed in the routine, which simplifies dataflow analyses.) (d) Routine to check array dependencies at the leaf defs.

4.3 Identifying the earliest position

We now describe how compute $\mathsf{Earliest}(u)$ for use u. Typically, dataflow analysis with array sections marks a set of nodes as "earliest" such that a copy of the communication code has to be placed at *all* these points. This is acceptable if each array section is communicated using a separate runtime call, but for our purposes, this greatly complicates code generation. In different control flow paths, communication for u may be combined with different references, making it impossible to generate a single version of the original computation containing u. The resulting code expansion can be enormous.

Therefore, we restrict our search to the *single* earliest position that dominates the use. Our experience with benchmarks, albeit limited, suggests that further sophistication is often unnecessary. The pseudocode for computing $\mathsf{Earliest}(u)$ for a use u is shown in Figure 8.

Claim 4.1 $\mathsf{Earliest}(u)$ *returns the earliest single dominating communication point d_1 for use u.*

In Figure 4, $\mathsf{Earliest}(a_1) = \mathsf{Earliest}(a_2) = 7$. Traditional array dataflow analysis, which does not insist on dominating

e. Mark candidates:
$c = \mathsf{CfgNode}(\mathsf{Latest}(u))$
While $c \neq \mathsf{CfgNode}(\mathsf{Earliest}(u))$ do
Mark all statements up to $\mathsf{Latest}(u)$ in basic block c
$c = \mathsf{DomTreeParent}(c)$
Mark all statements between $\mathsf{Earliest}(u)$ and $\mathsf{Latest}(u)$ in $\mathsf{CfgNode}(\mathsf{Earliest}(u))$.

f. Eliminate redundancy:
Repeat until no progress:
Find statement S and $c_1, c_2 \in \mathsf{CommSet}(S)$ such that c_2 subsumes c_1
For all S' dominated by S
disable c_1 in $\mathsf{CommSet}(S')$

g. GreedyChoose:
Let $\mathsf{StmtSet}(c) = \{S : c \in \mathsf{CommSet}(S)\}$
Consider entries c in increasing order of $|\mathsf{StmtSet}(c)|$:
For each $S \in \mathsf{StmtSet}(c)$, count the number of entries in $\mathsf{CommSet}(S)$ with which c can *combine* (see text)
Pick S with the highest count to place c
Delete c from $\mathsf{CommSet}(S')$ for all $S' \neq S$
Place each group of combined entries at the latest position common to the candidate placements of the entries it contains, including entries disabled during redundancy elimination.

Figure 9: Pseudocode for communication placement. (e) Pseudocode for marking all candidate statements for communication placement. (f) Pseudocode for global redundancy elimination. (g) Simple greedy heuristic to choose a final position from the set of candidates.

defs [14], would lead to $\mathsf{Earliest}'(a_1) = \mathsf{Earliest}'(a_2) = \{4, 6\}$. In both cases, a_2 subsumes a_1. We prove Claim 4.1 using the following three lemmas. We defer their proofs to the appendix.

Lemma 4.2 *d_1 dominates u.*

Lemma 4.3 *Let n_3 be any proper dominator-tree ancestor of d_1. Then there exists a regular def d_2 such that* $\mathsf{IsArrayDep}(d_2, u, \mathsf{CNL}(d_1, u))$ *returns* `TRUE` *and a path $d_2 \xrightarrow{*} d_1 \xrightarrow{+} u$ that bypasses n_3.*

Lemma 4.4 *There is no regular def d_4 along a path $d_1 \xrightarrow{+} d_4 \xrightarrow{+} u$ such that* $\mathsf{IsArrayDep}(d_4, u, \mathsf{CNL}(d_4, u))$ *returns* `TRUE`*, and there is a path from d_4 to u that bypasses d_1.*

Proof of Claim 4.1. Observe that only a node that dominates u can serve as a single communication point for u. Lemma 4.2 says that $d_1 = \mathsf{Earliest}(u)$ dominates u. Consider all dominator-tree ancestors of u. From this set, Lemma 4.3 rules out all nodes that strictly dominate d_1 as unsafe. Finally, Lemma 4.4 implies that d_1 is a safe communication point for u. ∎

4.4 Generating candidate positions

Since any safe position to insert a single copy of communication for use u must dominate u, the set of candidate positions has a very simple characterization in terms of the following claims. We omit the proofs.

Claim 4.5 *Starting at the basic block containing* $\mathsf{Latest}(u)$*, if we follow parent links in the dominator tree of the CFG, we will reach the basic block containing* $\mathsf{Earliest}(u)$.

Claim 4.6 *The statements marked in the basic blocks encountered during the dominator tree traversal from* $c(\mathsf{Latest}(u))$ *up to* $c(\mathsf{Earliest}(u))$ *are exactly those that are single candidate positions for communication placement for use u.*

Our algorithm for finding candidate placements of communication is thus extremely simple, and shown in Figure 9(e). In our example (Figure 4), statements 3, 4, 5, and 6 are not candidates for b_1 and b_2 because they do not dominate those uses.

4.5 Subset elimination

Our current algorithm gives priority to reducing the volume and number of messages over exploiting overlap benefits or reducing contention for buffers and cache. Given this simplification, we can preclude a large number of candidate positions without compromising the quality of the solution. Specifically, let $\mathsf{CommSet}(S)$ denote all communication entries associated with the statement S. A given entry can occur in the CommSet of many statements. If for statements S_1 and S_2 we have $\mathsf{CommSet}(S_1) \subset \mathsf{CommSet}(S_2)$, we can reset $\mathsf{CommSet}(S_1) = \emptyset$ without losing opportunities for combining or redundancy elimination. For example, in Figure 4, the CommSet of statements 1 and 2 can be safely set to $\emptyset$. In the case that $\mathsf{CommSet}(S_1) = \mathsf{CommSet}(S_2)$, either set may be emptied at this stage, because the actual choice governing the placement of communication would be made in the final step (§4.7).

4.6 Redundancy elimination

Typically, earlier approaches eliminated redundancy by examining the list of communications placed before each statement, and check each pair of entries to see if one subsumes the other. This test is based on the Available Section Descriptor (ASD) representation of communication [14]. Briefly, an ASD consists of a pair $\langle D, M \rangle$, where D represents the data (scalar variable or an array section) being communicated, and M is a mapping function that maps data to the processors which receive that data. A communication $\langle D_1, M_1 \rangle$ is made redundant by another communication $\langle D_2, M_2 \rangle$ if $D_1 \subseteq D_2$, and $M_1(D_1) \subseteq M_2(D_1)$.

In our case, since there can be many entries for a reference, we have to propagate the redundancy information globally. The pseudocode for eliminating redundant communication in the context of our current framework is shown in Figure 9(f). The modification is that in each step examining a statement S, the subsumed communication entry is cleared not only from $\mathsf{CommSet}(S)$ but from all statements S' such that S dominates S'. (The dominance ordering prevents a cycle of deletions.) We iterate over statements and communication pairs until no more elimination occurs.

Claim 4.7 *The subset and redundancy elimination steps are safe, i.e., the remaining communication entries are sufficient.*

One implication of the above ordering of eliminations is noteworthy. Consider our running example (Figure 4), specifically the communication due to the uses b_1 and b_2 ($\mathsf{ASD}(b_1) \subset \mathsf{ASD}(b_2)$). Since $\mathsf{Earliest}(b_1) = 1 \neq \mathsf{Earliest}(b_2) = 2$, an initial test of redundancy based on earliest placement, followed by candidate marking and subset elimination will not catch the redundancy. Thus, by choosing a later (than the earliest possible) placement for b_1, we are able to eliminate that communication completely. In contrast, the solution proposed in [14] would move each communication to the earliest point, and reduce the communication for b_2 to $\mathsf{ASD}(b_2) - \mathsf{ASD}(b_1)$, while the communication for b_1 would remain unchanged. The solution obtained with our current method is superior because it reduces the communication startup overhead, and it makes code generation much simpler.

4.7 Choosing from the candidates

At this stage we can still have a communication entry c in multiple CommSet's, and we have to arbitrate in favor of one. The goal is to minimize the total communication cost. In the common message cost model using fixed overhead per message and bandwidth, minimizing the cost is $\mathcal{NP}$-hard (also see §6). In practice, simple greedy heuristics work quite well; see Figure 9(g). It is similar to Click's global code motion heuristic [7]: consider the most constrained communication entry next, and put it where it is compatible in communication pattern (as shown by the test below) with the largest number of other candidate communication. A more refined heuristic would use estimates of message sizes and consider the communication cost if the current entry were combined with a given set of entries.

The entries in the CommSet of each statement can now be partitioned into groups, each group consisting of one or more entries which will be combined into a single aggregate communication operation. Any flexibility still available in placing this aggregate can be used to push this communication later if reducing contention for buffers and cache is more important than overlap benefits (as is folk truism for the SP2), or push it earlier if the situation were reversed. Our algorithm places communication for each group at the *latest* position common to the possible placements of each entry in that group (including positions disabled during the previous step for redundancy elimination). Deferring the placement decision until this final step enables our algorithm to take advantage of any possible placement that leads to redundancy elimination or combining benefits, without the drawback of unnecessary movement of communication that uses up more resources or degrades performance.

Criteria for communication compatibility. While in principle, code for any arbitrary communications can be combined into code for a single (and potentially complex) communication operation, we are interested in combining messages only when the startup overheads associated with all but one of them can be eliminated, leading to improved performance. Hence, we view two communications as compatible for combining if the associated sender-receiver relationships are identical or one is a subset of the other.

Thus, communications for $\langle D_1, M_1 \rangle$ and $\langle D_2, M_2 \rangle$ are combined only if $M_1 = M_2$ or $M_1 \subset M_2$. The combined communication is given by $\langle D_1 \cup D_2, M_2 \rangle$. In order to ensure better performance and for simplicity of code generation, we impose the following additional constraints on combining.

- The combined data size of $D_1 \cup D_2$ must be below a threshold (based on our study reported in §3, currently set to 20 KB for SP2), beyond which combining messages leads to diminishing returns or even worse performance. When data sizes are unknown, the compiler uses rules of thumb like assuming that NNC and reductions (where volume of data communicated is significantly lower than that involved in computation) are operating within the range suitable for combining.
- The size of $D_1 \cup D_2$, as approximated by a single section descriptor (array sections are not closed under the union operation), should not exceed the combined size of D_1 and D_2 by more than a small constant. This descriptor for $D_1 \cup D_2$ refers to identical sections of different arrays if D_1 and D_2 correspond to different arrays, and to a single array otherwise.

The check for $M_1 \subseteq M_2$ is done in the virtual processor space of template positions, as described in [14]. However, we have incorporated extensions to check for equality of mappings in physical processor space for nearest-neighbor communication and for mappings to a constant processor position [14].

4.8 Code generation

As shown in Figure 6, the step after communication analysis and optimization is to insert communication code in the form of subroutine calls to the pHPF runtime library routines, which in turn invoke MPL/MPI. The runtime library provides a high-level interface through which the compiler specifies the data being communicated in the form of array sections, and the runtime system takes care of packing and unpacking of data. For NNC, data is received in overlap regions [30] surrounding the local portion of the arrays. For other kinds of communication involving arrays, data is received into a buffer that is allocated dynamically, and the array reference that led to this communication is replaced by a reference to the buffer.

Redundant message elimination for NNC requires no further change to code generation. For other forms of communication, code generation has been modified to ensure that the array reference corresponding to eliminated communication is also replaced by a reference to the buffer holding non-local data, and that this buffer is deallocated only after its last use is over.

Combining messages for different arrays requires changes in code generation and the HPF runtime library routines. The data being sent or received is still represented by a single section descriptor, but now has a list of arrays associated with it. Correspondingly, the runtime routines now have to take on additional responsibilities of packing and unpacking data for the multiple array sections. Our benchmarks currently emit calls to a rudimentary runtime library with these features, but this has not been integrated into the compiler yet.

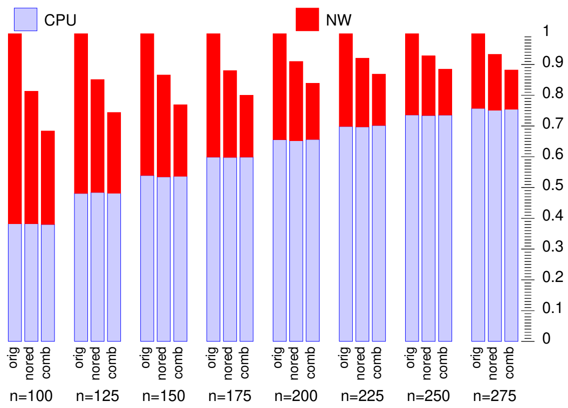
5 Performance

The analysis described in this paper has been implemented in the pHPF compiler. In order to study the potential performance benefits before the code generator and the run-time library could be modified to take advantage of the superior communication placement, we emitted scalarized code annotated with human readable communication entries after the analysis and optimization pass of the compiler. The table in Figure 10 shows some compile-time statistics of the reduction in the number of static call sites to the communication library. Static message counts are reduced by a factor of roughly 2–9.

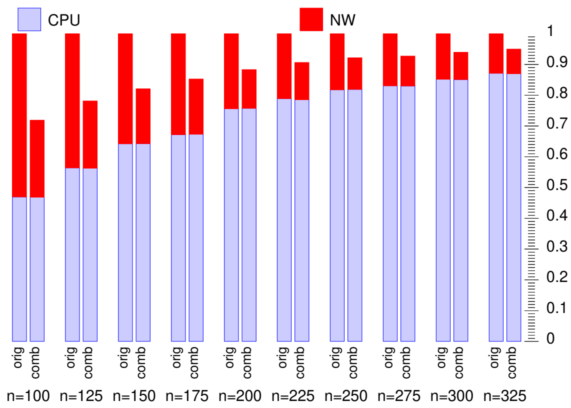
The trace emitted was then used to generate C programs with calls to MPL/MPICH message passing libraries. This

Benchmark Routine	shallow main	gravity main	gravity main	trimesh normdot	trimesh gauss	hydflo flux	hydflo hydro
Comm Type	NNC	NNC	SUM	NNC	NNC	NNC	NNC
Original (orig)	20	8	8	24	13	52	12
+ Redundancy elimination (nored)	14	8	8	24	13	30	12
+ Combined messages (comb)	8	4	2	4	4	6	6

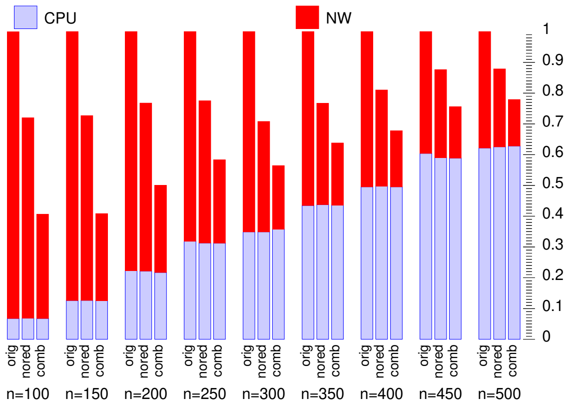
(a) SP2 **shallow** $P = 25$, $n \times n$, 50 runs



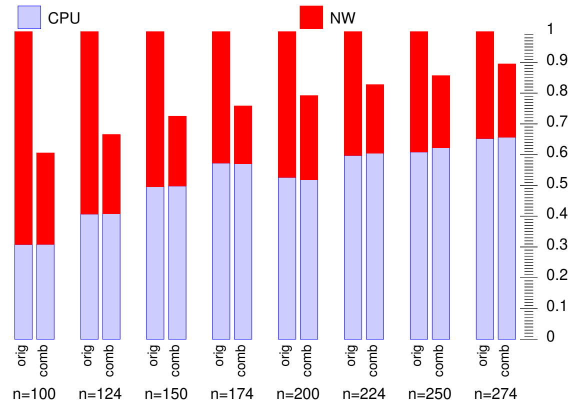
(b) SP2 **gravity** $P = 25$, $n \times n \times n$, 50 runs



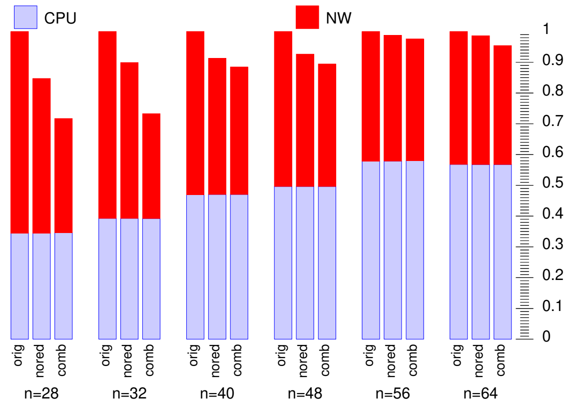
(c) NOW **shallow** $P = 8$, $n \times n$, 20 runs



(d) NOW **gravity** $P = 8$, $n \times n \times n$, 5 runs



(e) NOW **hydflo** $P = 8$, $5 \times n \times n \times n$, 5 runs



(f) NOW **trimesh** $P = 8$, $n \times n \times n$, 5 runs

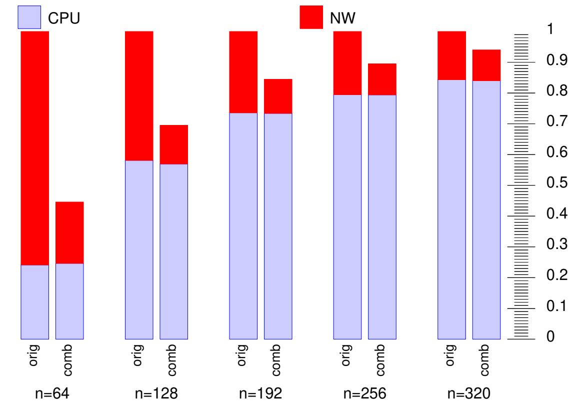


Figure 10: Performance analysis of the new algorithm: compile-time message counts and normalized running times.

enabled us to study performance improvements not only on the IBM SP2 but also on a network of workstations (NOW) consisting of Sparc workstations connected by a Myrinet switch. For each benchmark, the compiler generated two or three versions of code. The baseline pulls communication into outermost possible loops but does not detect redundancy or perform message scheduling. The next version uses earliest placement for redundancy elimination but does not perform message scheduling or combining. The final version uses the new algorithm. (Note: `gravity` and `trimesh` have no redundancy.) On the SP2, all codes were compiled using the IBM `xlc` compiler. On the NOW, we used `SUNWspro` compiler `cc`. Optimization `-O3` was used.

We report the results in Figure 10. Each diagram mentions the number of processors and the number of runs over which the median performance is reported. In each bar-chart the x-axis is the problem size. For each size two or three bars are plotted, one for each version of generated code. The y-axis is normalized so that the original code has unit running time, and the dark segment representing network cost shortens as optimizations are applied. These measurements were made with overlap disabled to clearly account for CPU and network activity. All floating point operations are on `double` (eight bytes). `shallow` and `trimesh` involve 2-d $n \times n$ arrays distributed `(BLOCK,BLOCK)`; `shallow` has 13 and `trimesh` has over 25 such arrays. Thus the problem sizes are realistic in that they occupy several MBytes. Communication time is reduced by a factor of 2–3. This typically translates to 10–40% overall running time reduction. `gravity` uses a 3-d $n \times n \times n$ array distributed `(*,BLOCK,BLOCK)`. Thus memory needed even at moderate n is quite staggering, and the graphs again show 10–40% overall gain in this reasonable size range. `hydflo` uses eight $5 \times (n+2)^3$ arrays. Therefore even for small n, the memory requirement is enormous, which affects the size range shown. Finally, the SP2 network has lower overhead and higher bandwidth than the NOW[4], which is evident from the higher overall performance gains on NOW compared to SP2, although the reduction in communication cost alone is roughly proportionate.

6 Extensions

The basic idea of exploiting flexibility in communication placement is rather general. Although for the sake of practicality our current prototype makes some justified simplifications, it would be interesting to extend the work in two ways. Currently, network architecture is undergoing considerable flux. If the CPU-network overlap can be exploited more effectively in future generation machines, the compiler could obtain better performance by considering the trade-offs between enhancing the overlap and reducing the number of messages and buffer contention. In particular, the simple subset elimination step (§4.5) would have to be dropped, as it could easily degrade the quality of the solution. In fact, the general problem becomes intractable when all of these conflicting optimizations are considered. The other direction comprises enhancements for handling special communication patterns like reductions.

6.1 General models

In a well-known model of communication cost, the problem of optimally selecting final candidates is $\mathcal{NP}$-hard, justifying our heuristic approach. A runtime call to the communication library in general leads to a many-to-many communication pattern. Let the inverse bandwidth of the network be scaled to one, and the message startup cost be C in these units. The cost of this pattern to a given processor is C times the total number of distinct processors that it sends to or receives from, plus the total volume of data that it sends or receives. Ignoring CPU-network overlap in our bulk-synchronous model, the cost of a pattern is the maximum cost over all processors, and the cost of a set of patterns is the sum of their costs. Unfortunately, we can show the following.

Claim 6.1 *Picking one candidate position for each reference, such that the total cost of all patterns is minimized, is $\mathcal{NP}$-hard. Specifically, there is an approximation-preserving reduction from* chromatic number.

Thus it is unlikely that our problem can be solved near-optimally in the worst case in polynomial time. Like many other $\mathcal{NP}$-hard problems, the optimization problem can be formulated as an integer linear program (ILP). Furthermore, several additional constraints can be incorporated into the ILP, including overlap between CPU and network and message buffer and cache constraints. Profile information would be crucial to specify this ILP and solve it to adequate precision.

6.2 Special communication patterns

Reduction communication is dealt with in a special way in the compiler since communication requirement is inverted, in a sense, for reductions. Whereas ordinary statements require communication to fill in remote values before computation can proceed, for reduction the computation occurs first (for the partial reduction operation on individual processors), followed by communication for the global reduction operation that must be completed before the use. Our preliminary prototype does not do reduction candidate marking yet. For communications which are marked as reductions, we need to employ a reversed SSA analysis, i.e., iterating through reached uses of a given definition to determine the latest point at which communication may be safely placed. Conceptually this is identical to the framework in this paper, but the implementation is left for future work. The current implementation does allow reduction communications placed at the same point to be combined, as in `gravity`.

7 Conclusion

We have presented an algorithm for global optimization of communication code placement in compilers for data-parallel languages like HPF. Modern parallel architectures greatly reward dealing with remote accesses throughout a program in an interdependent manner rather than naively generating messages for each of them. We achieve precisely this enabling optimization. In particular, we explore later placements of communication that preserve the benefits of redundancy elimination (normally obtained by moving communication earlier), reduce the wastage of resources like buffers for non-local data, and improve performance due to other factors like fewer messages. Preliminary performance results obtained on some HPF benchmarks show significant reduction in communication costs and overall improvements in performance of those programs on the IBM SP2 and

[4]Note that we use MPI on both machines, not Active Messages.

a cluster of Sparcs connected by Myrinet. In the future, we will conduct performance studies to investigate the desirability of including partial redundancy elimination as well into our framework. Another area for future work is interprocedural analysis; we believe that the application of our algorithm across procedure boundaries can often lead to further improvements in performance.

Acknowledgements. We wish to thank Edith Schonberg, Harini Srinivasan, Ko-Yang Wang, Charles Koelbel, Thomas Brandes, Ajay Sethi, Saman Amarasinghe, and Chau-Wen Tseng for helpful discussions, and Rich Martin and Lok Liu for help with NOW measurements.

A Appendix: Proofs

Proof of Lemma 4.2. (By contradiction.) We assume d_1 is not the pseudo-def at ENTRY, since the latter dominates all nodes in the CFG. Let $\ell_1 = \mathsf{NL}(d_1)$, and L_1 be the loop containing d_1. Note that $\ell_1 \leq \mathsf{NL}(u)$ because Earliest will never flag a d_1 with $\mathsf{NL}(d_1) > \mathsf{NL}(u)$. Assume d_1 does not dominate u. Then there exist two or more paths: one from ENTRY to u that bypasses d_1, and another from d_1 to u. If $\mathsf{NL}(u) = \mathsf{NL}(d_1)$, these two paths imply that there exists a ϕ-def at level ℓ_1 with (at least) two parameters, r_1 and r_2, such that there exist two non-overlapping backpaths: one from r_1 to d_1, and the other from r_2 to the pseudo-def at ENTRY that bypasses d_1. (Because of the zero-trip edges, we can ignore other loops nested in L_1.). That there is such a ϕ-def at level ℓ_1 still holds if $\mathsf{NL}(u) > \mathsf{NL}(d_1)$, because the preheader node of each loop containing u dominates u, and the two (or more) paths converge at the preheader node which is at level ℓ_1, at the latest. Test is called on at least one of these ϕ-defs, say p, before d_1 during the traversal of $\mathsf{Earliest}(u)$, starting from u. During execution of $\mathsf{Test}(p, u)$, Rcount gets called on defs $\mathsf{Reaching}(r_1)$ and $\mathsf{Reaching}(r_2)$, with nesting level $\mathsf{CNL}(p, u) = \ell_1$. The call at $\mathsf{Reaching}(r_1)$ returns a positive number, because some recursive call inspects d_1. Similarly the call at $\mathsf{Reaching}(r_2)$ also returns a positive number, because some recursive call inspects ENTRY. Since at least two invocations of Rcount return a positive numbers, the ϕ-def, not d_1, will be returned as $\mathsf{Earliest}(u)$ if d_1 does not dominate u, a contradiction. ∎

Proof of Lemma 4.3. If d_1 is the pseudo-def at ENTRY, there is nothing to prove. Also, if d_1 is a regular def, $\mathsf{IsArrayDep}(d_1, u, \mathsf{CNL}(d_1, u))$ must hold for d_1 to be returned as $\mathsf{Earliest}(u)$, in which case d_1 serves as the definition d_2 in the statement of the lemma. Therefore, we can assume d_1 is a ϕ-def.

By design, $\mathsf{Test}(d_1)$ returned TRUE because at least two Rcount calls on the ϕ-parameters of d_1 returned positive counts. But because of the visit[] array, no def is accounted more than once. Therefore the two positive counts can be attributed to two node-disjoint backpaths to two distinct regular defs (one of which could be ENTRY). At most one of these paths contain n_3. Let d_2 be some regular def on the other path such that $\mathsf{IsArrayDep}(d_2, u, \mathsf{CNL}(d_1, u)) =$ TRUE. Then there is a $d_2 \xrightarrow{+} u$ path bypassing n_3. ∎

Proof of Lemma 4.4. (By contradiction.) Assume there exists such a d_4. According to SSA construction, two cases can occur: either (1) d_4, as well as d_1, dominates u, or (2) d_4 has a path, bypassing d_1, from it to u through one or more ϕ-defs that dominate u.

Case 1. If d_4 dominates u, d_4 cannot also dominate d_1. Otherwise, there exists a path from ENTRY to d_4 to u that bypasses d_1 (second condition in the lemma), in which case d_1 cannot dominate u, contradicting Lemma 4.2. Therefore, d_1 dominates d_4 (note that if both d_4 and d_1 dominate u, one of them must dominate the other), which in turn dominates u. Thus $\mathsf{Test}(d_4, u)$ is called before $\mathsf{Test}(d_1, u)$ by $\mathsf{Earliest}(u)$. $\mathsf{Test}(d_4, u) =$ TRUE because $\mathsf{IsArrayDep}(d_4, u, \mathsf{CNL}(d_4, u)) =$ TRUE, so d_4 will get returned as $\mathsf{Earliest}(u)$; a contradiction.

Case 2. In the second case, d_1 dominates the ϕ-defs. If not, then d_1 would not dominate u either, (contrary to Lemma 4.2) because there is a path $d_4 \xrightarrow{+} \phi \xrightarrow{+} u$ avoiding d_1. Hence, these ϕ-defs are dominated by d_1 and are visited before d_1 by $\mathsf{Earliest}(u)$. It follows, from a similar argument in the proof of Lemma 4.2, that these two paths converge at some node at level $\mathsf{CNL}(d_4, u)$, creating a ϕ-def at level $\mathsf{CNL}(d_4, u)$. This ϕ-node has (at least) two parameters, r_1 and r_2, such that there exist two non-overlapping paths: one from d_4 to r_1, and the other from d_1 to r_2. When applied to r_1, Rcount returns positive, possibly because of d_4, which satisfies $\mathsf{IsArrayDep}(d_4, u, \mathsf{CNL}(d_4, u))$. When applied to r_2, Rcount returns positive, possibly because of the pseudo-def at ENTRY. Since (at least) two parameters return positive, the ϕ-def, not d_1, is returned by $\mathsf{Earliest}(u)$, another contradiction. ∎

References

[1] G. Agarwal, J. Saltz, and R. Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Programming Language Design and Implementation (PLDI)*, La Jolla, CA, June 1995.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the ptran analysis system for multiprocessing. *Proc. ACM 1987 International Conference on Supercomputing*, 1987. Also published in *Journal of Parallel and Distributed Computing*, Oct., 1988, 5(5) pages 617-640.

[3] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Programming Language Design and Implementation (PLDI)*, Albuquerque, NM, June 1993. ACM SIGPLAN.

[4] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. A compilation approach for Fortran 90D/HPF compilers on distributed memory MIMD computers. In *Proc. Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, Aug. 1993.

[5] T. Brandes. ADAPTOR: A compilation system for data-parallel Fortran programs. In C. W. Kessler, editor, *Automatic parallelization – new approaches to code generation, data distribution, and performance prediction.* Vieweg Advanced Studies in Computer Science, Vieweg, Wiesbaden, Jan. 1994.

[6] J.-D. Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, Feb. 1994.

[7] C. Click. Global code motion global value numbering. In *Programming Language Design and Implementation (PLDI)*, pages 246–257. ACM SIGPLAN, 1995.

[8] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[9] H. P. F. Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[10] E. Granston and A. Veidenbaum. Detecting redundant accesses to array data. In *Proc. Supercomputing '91*, pages 854–965, 1991.

[11] M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication on multicomputers. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[12] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, K. Wang, D. Shields, W.-M. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proc. Supercomputing '95*, San Diego, CA, Dec. 1995.

[13] M. Gupta and E. Schonberg. Static analysis to reduce synchronization costs in data-parallel programs. In *Principles of Programming Languages (POPL)*, St. Petersburg Beach, FL, Jan. 1996.

[14] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs. Technical Report RC 19872(87937) 12/14/94, IBM Research, 1994. To appear in *IEEE Transactions on Parallel and Distributed Systems*.

[15] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[16] K. Keeton, T. Anderson, and D. Patterson. LogP quantified: The case for low-overhead local area networks. In *Proc. Hot Interconnects III: A Symposium on High Performance Interconnects*, Stanford, CA, Aug. 1995.

[17] K. Kennedy and N. Nedeljkovic. Combining dependence and data-flow analyses to optimize communication. In *International Parallel Processing Symposium*. IEEE, 1995.

[18] K. Kennedy and A. Sethi. A constraint-based communication placement framework. Technical Report CRPC-TR95515-S, CRPC, Rice University, 1995.

[19] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Programming Language Design and Implementation (PLDI)*, San Francisco, CA, June 1992.

[20] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[21] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 62–73. ACM SIGPLAN, 1992.

[22] M. O'Boyle and F. Bodin. Compiler reduction of synchronization in shared virtual memory systems. In *Proc. 9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[23] V. Sarkar. The PTRAN parallel programming system. *Parallel Functional Programming Languages and Compilers*, pages 309–391, 1991.

[24] M. Snir et al. The communication software and parallel environment of the IBM SP2. *IBM Systems Journal*, 34(2):205–221, 1995.

[25] C. Stunkel et al. The SP2 high performance switch. *IBM Systems Journal*, 34(2):185–204, 1995.

[26] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Principles and Practice of Parallel Programming (PPoPP)*, Santa Barbara, CA, July 1995.

[27] R. v Hanxleden and K. Kennedy. Give-n-Take—a balanced code placement framework. In *Programming Language Design and Implementation (PLDI)*, Orlando, FL, June 1994. ACM SIGPLAN.

[28] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[29] M. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, Apr. 1987.

[30] H. Zima, H. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.