Parallel Processing Letters

DEMAND–DRIVEN DATA FLOW ANALYSIS FOR COMMUNICATION OPTIMIZATION*

Xin Yuan
Department of Computer Science, The University of Pittsburgh
Pittsburgh, PA 15260, USA

Rajiv Gupta
Department of Computer Science, The University of Pittsburgh
Pittsburgh, PA 15260, USA

Rami Melhem
Department of Computer Science, The University of Pittsburgh
Pittsburgh, PA 15260, USA

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

Exhaustive global array data flow analysis for communication optimization is expensive and considered to be impractical for large programs. This paper proposes a demand-driven analysis approach that reduces the analysis cost by computing only the data flow information related to optimizations. In addition, the analysis cost of our scheme can be effectively managed by trading space for time or compromising precision for time.

Keywords: Demand–driven data flow analysis, Global communication optimization

1 Introduction

Compilers for distributed memory machines translate shared memory programs into SPMD programs with explicit message passing [1,8]. An important task for such compilers is to reduce communication costs through optimizations such as message vectorization and redundant communication elimination. Program analysis must be performed to obtain information required for the optimizations. Two different analysis approaches, one based on data dependence analysis [10] and the other using array data flow analysis [7,6], have been proposed. While the data dependence approach is more efficient in terms of the analysis cost, the array data flow analysis approach has the advantage of better precision. Array data flow analysis propagates some form of array section descriptor [6,7]. Due to the complexity of the array

*This research is supported in part by NSF award CCR-9157371 and by AFOSR award F49620-93-1-0023DEF.

section descriptor, the propagation of data flow information can be expensive both in time and space. Furthermore, in traditional data flow analysis techniques, obtaining data flow information at *one* point requires the computation of data flow information at *all* program points. Computing such exhaustive solutions results in *over-analysis* of a program and decreases the efficiency and precision of the analysis.

These problems in array data flow analysis for communication optimization can be alleviated by using a demand driven data flow analysis technique which computes data flow information only when it is needed. One other advantage of the demand driven approach is that the analysis cost can be managed. Intermediate results may be recomputed every time they are requested or by maintaining a result cache, repeated computations may be avoided, therefore, allowing a trade-off between space and time. Demand-driven analysis proceeds by considering the part of a program that is close to the point where the data flow information is needed, yielding better approximations when the analysis is terminated prematurely. Thus, demand-driven analysis also allows trading precision for time.

This paper presents a demand driven array data flow analysis scheme for global communication optimization. Section 2 describes the program representation and introduces the *section communication descriptor* (SCD) which is used to represent communications. Section 3 presents general rules to propagate SCDs in a demand driven manner. Applications of the general rules are discussed in Section 4.

2 Program Representation and Section Communication Descriptor

We consider structured programs that contain conditionals and nested loops. A loop is controlled by a basic induction variable whose value ranges from one to an upper bound with an increment of 1. Array references are restricted to references whose subscripts are affine functions of loop induction variables, that is, references are of the form $X(f_1(\vec{i}), f_2(\vec{i}), ..., f_s(\vec{i}))$, where $\vec{i}$ is a vector representing loop induction variables and $f_k(\vec{i})$, $1 \leq k \leq s$, are affine functions of $\vec{i}$. Note that analyzing communication for irregular problems, that is, array subscripts are not affine functions, requires totally different techniques and is beyond the scope of this paper.

Our algorithm performs interval analysis on each subroutine in a demand driven manner. It uses a variant of Tarjan's intervals [11]. A subroutine is represented by an *interval flow graph*, $G = (N, E)$ with nodes N and edges E. $ROOT$ is a special node in N which is viewed as a header node for a subroutine. For $n \in N$, LEVEL(n) is the loop nesting level of n. The outermost level is level 0 (i.e., LEVEL(ROOT) = 0) and the innermost loop has the highest level number. The analysis requires that there are no *critical edges* which connect a node with multiple outgoing edges to a node with multiple incoming edges. Critical edges can be removed by edge splitting transformation [7]. Fig. 1 shows an example interval flow graph. Node 9 in the example is an additional node inserted by the edge splitting transformation.

The processor space is considered as an unbounded grid of virtual processors which is similar to a *template* in High Performance Fortran (HPF) [9]. All arrays are aligned to the virtual processor grid. In this paper when we refer to communication, we mean communication on the virtual processor grid. Communications in a program are represented by *Section Communication Descriptors* (SCD). The SCD is an extension of array section descriptor. It describes an array region and the source–destination relation in the communication. A SCD is defined as $< N, D, M >$, where N is an array name, D is a region descriptor and M is a descriptor describing the source-destination relation.

```
ALIGN (i, j) with VPROCS(i, j) :: x, y, z
ALIGN (i, j) with VPROCS(2*j, i+1) :: w
ALIGN (i) with VPROCS(i, 1) :: a, b
```

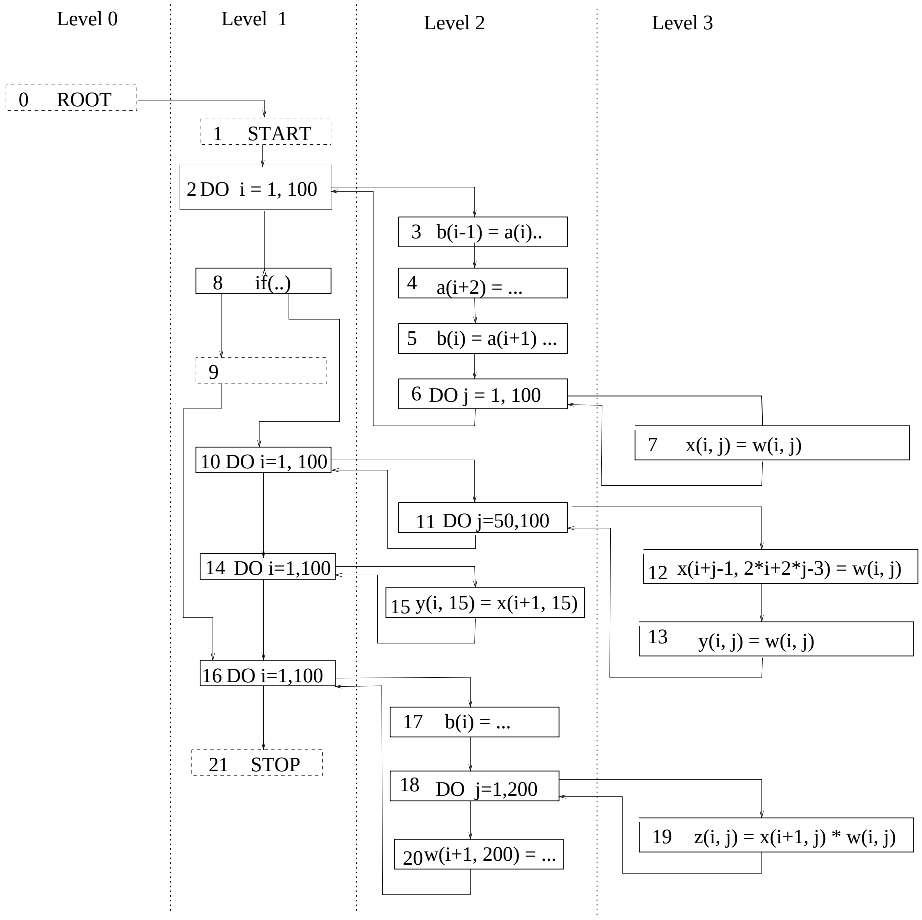


Fig. 1: An example of an interval flow graph.

The *bounded regular section descriptor* (BRSD) [2] is used to describe the array region. As discussed in [2], set operations can be efficiently performed on BRSDs. The region D is 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 used in HPF.

The source-destination mapping M is denoted as a pair $< src, dst >$. The source, src, is a vector whose elements are of the form $\alpha * i + \beta$, where α and β are invariants and i is a loop index variable. The destination, dst, is a vector whose elements are of the form $\sum_{j=1}^{n} \alpha_j * i_j + \beta_j$, where α_j's and β_j's are invariants and i_j's are loop index variables. Notation $M = \top$ denotes arbitrary mappings.

Demand driven array data flow analysis obtains information by propagating SCDs. The propagation starts with SCDs that represent communications in an assignment statement. Next, we will discuss how to calculate SCDs for assignment statements. To calculate the communication requirement of an assignment statement, the compiler must first know the owner of each array element. We assume that all arrays are aligned to the virtual processor grid using 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 can be specified by an *alignment matrix* A and an *alignment offset vector* $\vec{\alpha}$. That is, the virtual processor $\vec{v}$ that owns the array element $\vec{d}$ satisfies the equation $\vec{v} = A\vec{d} + \vec{\alpha}$. Consider array w in Fig. 1, the alignment matrix and the offset vector are as follows:

$$A_w = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix},\ \vec{\alpha}_w = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Once the ownership information is given, SCDs for a statement can be computed from the program structure. We assume that the *owner computes* rule is used. The owner computes rule requires each array element on the *rhs* of an assignment statement to be sent to the processor that owns the *lhs*. Let us consider each component in a $SCD =< N, D, M >$ for an assignment statement. Since subscript expressions in array references are affine functions of loop indices, array references can be expressed as $N(G\vec{i} + \vec{g})$, where N is the array name, G is a matrix and $\vec{g}$ is a vector. We call G the *data access matrix* and $\vec{g}$ the *access offset vector*. Let the array reference in lhs to be $x(G_l\vec{i} + \vec{g_l})$, the array reference in rhs to be $y(G_r\vec{i} + \vec{g_r})$. We have $N = y$ and $D = (G_r\vec{i} + \vec{g_r})$. To obtain the mapping $M =< src, dst >$, we must consider the alignment of arrays x and y. Let A_x, and $\vec{\alpha}_x$ be the alignment matrix and the alignment offset vector for array x, A_y and $\vec{\alpha}_y$ be the alignment matrix and the alignment offset vector for array y. The source processor src, which represents processors owning the rhs, and the destination processor dst, which represents processors owning the lhs, are given by following equations.

$$src = A_y(G_r\vec{i} + \vec{g_r}) + \vec{\alpha}_y, \qquad dst = A_x(G_l\vec{i} + \vec{g_l}) + \vec{\alpha}_x$$

Consider the communication for the statement in node 12 in Fig. 1. The compiler determines the following data access matrices, access offset vectors, alignment matrices, alignment vectors, and the SCD describing the communication.

$$A_x = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ \vec{\alpha}_x = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, A_w = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix},\ \vec{\alpha}_w = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

$$G_l = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix},\ \vec{g_l} = \begin{pmatrix} -1 \\ -3 \end{pmatrix}, G_r = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ \vec{g_r} = \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$

$$SCD =< w, (i, j), < (2 * j, i + 1), (i + j - 1, 2 * i + 2 * j - 3) >>.$$

Set operations on SCDs are needed in our analysis. In most cases operations involve SCDs with the same mapping relation or with the mapping relation in one SCD being a subset of another SCD. Operations on SCDs with *unrelated* mappings results in conservative approximations or list representation. We define two mapping relations M_1 and M_2 to be unrelated if and only if M_1 does not include M_2 and M_2 does not include M_1. Next we describe the operations.

Mapping Subset Testing. Testing whether a mapping $M_1 =< s_1, d_1 >$ is a subset of another mapping $M_2 =< s_2, d_2 >$ is an important operation. It is done by checking if equations $s_1 = s_2$ and $d_1 = d_2$ have a valid solution. Here, variables in M_2 are treated as variables in the equation system while variables in M_1 are treated as constants. For example, to determine whether $M_1 =< (1, i), (1, i + 1) >$ is a subset of $M_2 =< (i, j), (i, j + 1) >$, the system of equations $(i = 1, j = k, i = 1, j + 1 = k + 1)$ is solved, where the variable i in M_1 is renamed to k. Since there is a solution $(i = 1, j = k)$, M_1 is a subset of M_2.

Intersection Operation. The intersection of two SCDs represents the elements common to the array sections that have the same mapping relation.

$< N_1, D_1, M_1 > \cap < N_2, D_2, M_2 >$
$= \phi$, if $N_1 \neq N_2$ or M_1 and M_2 have no relation
$= < N_1, D_1 \cap D_2, M_1 >$, if $N_1 = N_2$ and $M_1 \subseteq M_2$
$= < N_1, D_1 \cap D_2, M_2 >$, if $N_1 = N_2$ and $M_1 \supseteq M_2$

Difference Operation. The difference operation causes a part of the array region associated with the first operand to be invalidated. We only consider the case when the mapping relation for the second operand is $\top$, which means arbitrary mapping. This is because in our analysis the difference operation is used only when elements in the descriptor are killed by some definitions.

$< N_1, D_1, M_1 > - < N_2, D_2, \top >$
$= < N_1, D_1, M_1 >$, if $N_1 \neq N_2$
$= < N_1, D_1 - D_2, M_1 >$, if $N_1 = N_2$.

Union Operation. The union of two SCDs represents the elements that can be in either part of the array sections. This operation is given by:

$< N_1, D_1, M_1 > \cup < N_2, D_2, M_2 >$
$= < N_1, D_1 \cup D2, M1 >$, if $N_1 = N_2$ and $M_1 = M_2$
$=$ list$(< N_1, D_1, M_1 >, < N_2, D_2, M_2 >)$, otherwise.

3 Demand-Driven SCD Propagation

Communication optimization opportunities can be uncovered by propagating SCDs globally. Backward propagation of SCDs can find the earliest points where the communication can be placed, while forward propagation can find the latest points where the communications are alive. In this section, we present general rules to propagate SCDs in a demand driven manner through an interval flow graph. These rules propagate SCDs to their earliest or latest points. In practice, however, a compiler may choose to terminate SCD propagations prematurely to save analysis time. Since forward and backward propagation are similar, we will focus on backward propagation of SCDs.

The demand-driven analysis propagates SCDs in a direction that is the reverse of the direction in traditional exhaustive interval analysis. During the propagation, a SCD may be expanded when it is propagated out of a loop. When a set of elements in a SCD is killed inside a loop, the set is propagated into the loop to determine the exact points where the elements are killed. Thus, there are two types of propagations. During *upward* propagation SCDs are propagated from higher to lower levels and may need to be expanded. During *downward* propagation SCDs are propagated from lower to higher levels and may need to be shrunk.

The form of a *propagation request* is $< S, n, [UP|DOWN], level, cnum >$, where S is a SCD, n is a node in a flow graph, constants UP and $DOWN$ indicate whether the request is an upward propagation or a downward propagation, $level$ indicates at which level the request is and the value $cnum$ indicates which child node of n has triggered the request. A special value -1 for $cnum$ is used to indicate the beginning of a propagation. Hence, a compiler can start a propagation by placing an upward request with $cnum = -1$. A downward propagation is triggered automatically when a loop header node is encountered.

For efficiency reasons a node is processed only when all of its successors have been processed. This guarantees that each node will be processed once for each propagation since each interval is an acyclic flow graph. When all successors of a node n place propagation requests, an action on node n is triggered and a SCD is propagated from node n to all of its predecessors.

RULE 0: Upward propagation at initial node. As we mentioned earlier, the compiler starts an upward propagation from node s by placing the request, $< S, s, UP, level, cnum >$, with $cnum = -1$. Here, S is the SCD descriptor to be propagated. This request is processed by propagating S to all the predecessors of node s as shown in the following code. In the code, the function $pred(s)$ returns the set of all predecessors of node s.

REQUEST($< SCD, s, UP, level, cnum >$):
  if ($cnum = -1$) **then**
    forall $m \in pred(s)$, where s is m's jth child **do**
      REQUEST($< SCD, m, UP, level, j >$)

RULE 1: Upward propagation at a regular node. Regular nodes include all nodes other than loop header nodes. Requests on a regular node trigger an action based on SCDs and the local information. In the following code, functions *action* and *local* depend on the type of optimization being performed.

REQUEST($< S_1, n, UP, level, 1 >$) $\wedge$ ... $\wedge$ REQUEST($< S_k, n, UP, level, k >$) :
S = $S_1 \cap ... \cap S_k$
action(S, local(n))
if $(S - kill_n \neq \phi)$ **then**
forall $m \in pred(n)$, where n is m's jth child **do**
REQUEST($< S - kill_n, m, UP, level, j >$)

A response to requests in a node n occurs only when all of its successors have been processed. This guarantees that in an acyclic flow graph each node will only be processed once. A more aggressive scheme can propagate a request through a node without checking whether all its successors are processed. In that scheme, however, nodes may need to be processed multiple times to obtain final solutions.

RULE 2: Upward propagation at a same level loop header node. Here we consider a request $< S, n, UP, level, cnum >$ such that n is a loop header and $Level(n) = level$. Processing the node requires calculating the summary information, K_n, for the interval, performing an action based on S and K_n, propagating a SCD past the loop and triggering a downward propagation into the loop body.

The summary function can be calculated either before hand or in a demand driven manner. Later in this section we will describe an algorithm to calculate the summary. K_n is the summary information of the loop representing all variables killed in the interval. Note that a loop header can only have one successor besides the entry edge into the loop body. The *cnum* in the downward request is set to -1 to indicate that it is the start of a downward propagation.

REQUEST($< S, n, UP, level, cnum >$):
if ((n is a header) and (LEVEL(n) = level)) **then**
calculate summary K_n
action(S, K_n)
if $(S - K_n \neq \phi)$ **then**
forall $m \in pred(n)$, where n is m's jth child **do**
REQUEST($< S - K_n, m, UP, level, j >$)
if $(S \cap K_n \neq \phi)$ **then**
REQUEST($< S \cap K_n, n, DOWN, level, -1 >$)

RULE 3: Upward propagation at a lower level loop header node. Here we consider a request $< S, n, UP, level, cnum >$ such that n is a loop header and $Level(n) < level$. Once a request reaches the loop header. The request is expanded before propagated into the lower level. At the same time, this request triggers a downward propagation for communications that must stay inside the loop. In the code, we assume that the loop index variable is i with bounds *low* and *high*.

REQUEST($< S, n, UP, level, cnum >$):
if ((n is a header) and ($LEVEL(n) < level$)) **then**
calculate the summary, K_n, of loop n
outside = $expand(S, i, low : high) - \cup_{def} expand(def, i, low : high)$
inside = $expand(S, i, low : high) \cap \cup_{def} expand(def, i, low : high)$
if (outside $\neq \phi$) **then**
forall $m \in pred(n)$, where n is m's jth child **do**
REQUEST($< outside, m, UP, level - 1, j >$)
if (inside $\neq \phi$) **then**
REQUEST($< inside, n, DOWN, level - 1, -1 >$)

The variable *outside* represents elements that are propagated out of the loop, while *inside* represents the elements that are killed within the loop. The expansion function has similar definition as in [7]. For a SCD descriptor S, $expand(S, i, low : high)$ is a function which replaces all single data item references $\alpha * i + \beta$ used in any array section descriptor D in S by a triple $\alpha * low + \beta : \alpha * high + \beta : \alpha$. The set def includes all definitions that are sources of flow-dependences to the array region propagated.

RULE 4: Downward propagation at initial node. A downward propagation starts from a loop header node with $cnum = -1$ and ends at the same node with $cnum \neq -1$. In the downward propagation, the loop's index variable i is treated as a constant. Hence, SCDs that are propagated into the loop body must be changed to be the available SCDs for iteration i, that is, we must subtract the elements killed in iterations $i + 1$ to $high$ in backward propagation. This propagation prepares the downward propagation into the loop body by shrinking SCDs.

REQUEST($< S, n, DOWN, level, cnum >$):
if ($cnum = -1$) **then**
calculate the summary of loop n;
ite = $S - \cup_{def} expand(def, i, i + 1 : high)$
REQUEST($< ite, l, DOWN, level + 1, 1 >$);

RULE 5: Downward propagation at a regular node. For regular node, the downward propagation is similar to the upward propagation.

REQUEST($< S_1, n, DOWN, level, 1 >$) $\wedge$ ...
... $\wedge$ REQUEST($< S_k, n, DOWN, level, k >$):
S = $S_1 \cap ... \cap S_k$
action(S, local(n))
if ($S - kill_n \neq \phi$) **then**
forall $m \in pred(n)$, where n is m's jth child **do**
REQUEST($< S - kill_n, m, DOWN, level, j >$)

RULE 6: Downward propagation at a same level loop header node. When a downward propagation reaches a loop header, if the loop header is in lower level, then the propagation stops; otherwise, the loop header is in the same level as the request, and we must propagate an appropriate SCD past the loop header and generate a downward propagation into the nested loop.

REQUEST($< S, n, DOWN, level, cnum >$):
  if ($LEVEL(n) < level$) **then** STOP
  if (n is a header) and (LEVEL(n) = level) **then**
    calculate summary, K_n, for loop T(n)
    action(S, K_n);
    if ($S - K_n \neq \phi$) **then**
      forall $m \in pred(n)$, where n is m's jth child **do**
        REQUEST($< S - K_n, m, DOWN, level, j >$)
    if ($S \cap K_n \neq \phi$) **then**
      REQUEST($< S \cap K_n, n, DOWN, level, -1 >$);

Let us consider propagating the communication for array w in node 19 in Fig. 1. The positions and values of SCDs during the propagation are shown in Fig. 2 (a). The sequence of SCDs and rules used to generate the SCDs are shown in Fig. 2 (b).

During SCD propagations, summary information of an interval is needed when a loop header is encountered. Next we describe the computation of summary information. This algorithm can be invoked when the need for summary information arises. We use the calculation of kill set of the interval K_n as an example. Let $kill(i)$ be the variables killed in node i, K_{in} and K_{out} be the variables killed before and after the node respectively. The algorithm in Fig. 3 propagates data flow information from the tail node to the header node in an interval using the following data flow equations:

$$K_{out}(n) = \cup_{s \in succ(n)} K_{in}(s),$$
$$K_{in}(n) = kill(n) \cup K_{out}(n).$$

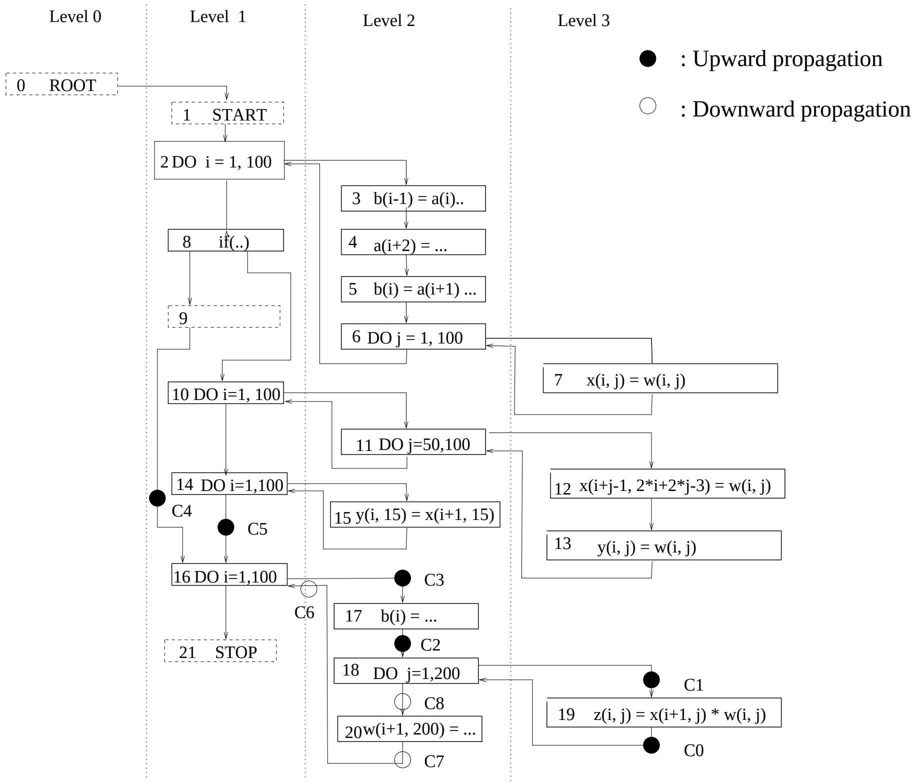
When an inner loop header is encountered, a recursive call is issued to get the summary information for the inner interval. Once the loop header is reached, the kill set is expanded to be used by the outer loop.

4 Demand-Driven Optimization

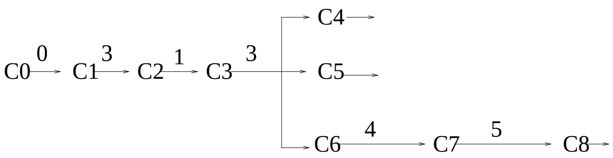
The general SCD propagation rules presented in the preceding section can be easily adapted for specific communication optimizations. Two applications, *message vectorization* and *redundant communication elimination*, are discussed next.

Message vectorization tries to hoist communications out of a loop body so that instead of sending large number of small messages inside the loop body, a smaller number of large messages are communicated outside the loop. This optimization can be done by propagating SCDs for an assignment statement in backward direction in the flow graph. Since in message vectorization, communications are hoisted out of a loop, only *upward* propagation is needed.

$C0 = (< w, (i,j), < (2*j, i+1), (i,j) >>, 19, UP, 3, -1)$
$C1 = (< w, (i,j), < (2*j, i+1), (i,j) >>, 18, UP, 3, 1)$
$C2 = (< w, (i, 1:200:1), < (2*j, i+1), (i,j) >>, 17, UP, 2, 1)$
$C3 = (< w, (i, 1:200:1), < (2*j, i+1), (i,j) >>, 16, UP, 2, 1)$
$C4 = (< w, \{(1:100:1, 1:199:1), (1,200)\}, < (2*j, i+1), (i,j) >>, 9, UP, 1, 1)$
$C5 = (< w, \{(1:100:1, 1:199:1), (1,200)\}, < (2*j, i+1), (i,j) >>, 14, UP, 1, 1)$
$C6 = (< w, (2:100:1, 200), < (2*j, i+1), (i,j) >>, 16, DOWN, 1, -1)$
$C7 = (< w, (2:i+1:1, 200), < (2*j, i+1), (i,j) >>, 20, DOWN, 2, 1)$
$C8 = (< w, (2:i:1, 200), < (2*j, i+1), (i,j) >>, 18, DOWN, 2, 1)$



(a)



(b)

Fig. 2: Propagating SCDs

Summary_kill(n)
  $K_{out}(tail) = \phi$
  forall $m \in T(n)$ and in backward order **do**
    if m is a loop header **then**
      $K_{out}(m) = \cup_{s \in succ(m)} K_{in}(s)$
      $K_{in}(m) =$ summary_kill(m) $\cup K_{out}(m)$
    else
      $K_{out}(m) = \cup_{s \in succ(m)} K_{in}(s)$
      $K_{in}(m) = kill(m) \cup K_{out}(m)$
  return(expand($K_{in}(header)$, i, low:high))

Fig. 3: Summary kill calculation.

For the compiler to perform message vectorization for a statement at node n in an interval flow graph, the compiler will calculate the SCD, S, for the statement, and initiate a propagation with $REQUEST(S, n, UP, LEVEL(n), -1)$. Rules are modified so that the downward propagation is not triggered. The parts of SCDs that can be propagated out of the loop (not killed within the loop body) represent the communications that can be vectorized. The communications of these parts will be placed immediately preceding the loop by a local action. The communication of elements that are killed inside loops stay at the point preceding the reference.

As an example, consider vectorizing the communication of array w in node 19 in Fig. 2. For communications described by SCD $C0$, communication $< w, \{(1:100, 1:199), (1, 200)\}, < (2*j, i+1), (i, j) >>$ can be propagated out of the outermost loop (node 16) while communication $< w, (2:100, 200), < (2*j, i+1), (i, j) >>$ can be propagated out of the inner loop (node 18). Hence, the communication for array w in statement node 19 can be vectorized by placing the communication $< w, \{(1:100, 1:199), (1, 200)\}, < (2*j, i+1), (i, j) >>$ before node 16 and communication $< w, (2:100, 200), < (2*j, i+1), (i, j) >>$ before node 18.

After message vectorization, communications can be further optimized by *redundant communication elimination*. Redundant communication elimination can also be done by propagating SCDs. Two different schemes can be used for this optimization. One scheme propagates a SCD to be eliminated to find communications that can subsume the SCD. The other method propagates a SCD to identify other communications that can be subsumed. These two schemes use similar propagation method, we only consider the second scheme in the remainder of this section.

To find communications that can be subsumed by a communication S at node n, the compiler initiates $REQUEST(< S, n, UP, LEVEL(n), -1)$. During the propagation, when a communication C that is a subset of S is found, the communication C is marked as redundant. The propagation stops when S is killed or ROOT is reached. The propagation follows the rules discussed in last section. However, downward propagation triggered by rule 6 is not needed. The local action at each node will place the communication for the elements that are killed in the node immediately following the node. Local action at a branch node may also place communications at its successors when two different SCD sets are propagated to the branch node. For example, let a branch node n receive $REQUEST(< S_1, n, UP, level, 1 >)$ and $REQUEST(< S_2, n, UP, level, 2 >)$ from its successors n_1 and n_2, respectively. The local action at node n will place communication $S_1 - S_1 \cap S_2$ before node n_1 and communication $S_2 - S_1 \cap S_2$ before node n_2. The redundant communication elimination can be combined with the message vectorization phase.

In summary, we have shown that the demand driven data flow analysis proposed in this paper can be used to perform communication optimizations. These algorithms have been implemented in a global communication optimizer and have been shown to be effective both in terms of analysis cost and communication optimizations [12].

References

1. S. P. Amarasinghe and M. S. Lam, Communication optimization and code generation for distributed memory machine, in *Proc. ACM SIGPLAN Conf. on Programming Languages Design and Implementation*, Albuquerque, NM, 1993, 126–138.
2. D. Callahan and K. Kennedy, Analysis of interprocedural side effects in a parallel programming environment, *J. Parallel and Distributed Comput.* **5** (1988) 517–550.
3. S. Chakrabarti, M. Gupta and J. Choi, Global communication analysis and optimization, in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Philadelphia, PA, 1996, 68–78.
4. J.F. Collard, d. Barthou and P. Feautrier, Fuzzy array dataflow analysis, in *Proc. 5th ACM SIGPLAN Symp. on Principle & Practice of Parallel Programming*, Santa Barbara, CA, 1995, 92–101.
5. E. Duesterwald, R. Gupta and M. L. Soffa, Demand-driven computation of interprocedural data flow, in *Proc. Symposium on Principles of Programming Languages*, San Francisco, CA, 1995, 37–48.
6. C. Gong, R. Gupta and R. Melhem, Compilation techniques for optimizing communication on distributed-memory systems, in *Proc. International Conf. on Parallel Processing*, St. Charles, IL, 1993, Vol. II, 39–46.
7. M. Gupta, E. Schonberg and H. Srinivasan, A unified framework for optimizing communication in data-parallel programs, *IEEE Trans. on Parallel and Distributed Systems* **7** (1996) 689–704.
8. S. Hiranandani, K. Kennedy and C. Tseng, Compiling Fortran D for MIMD distributed–memory machines, *Communications of the ACM* **35** (1992) 66–80.
9. High Performance Fortran Forum, High performance Fortran language specification, Version 1.0 Technique Report CRPC-TR92225, Rice University, 1993.
10. K. Kennedy and N. Nedeljkovic, Combining dependence and data-flow analyses to optimize communication, in *Proc. 9th International Parallel Processing Symposium*, Santa Barbara, CA, 1995, 340–346.
11. R.E. Tarjan, Testing flow graph reducibility, *J. Comput. System Sci.* **9** (1974) 355–365.
12. X. Yuan, R. Gupta and R. Melhem, "An Array Data Flow Analysis Based Communication Optimizer." *Technical Report*, TR-97-06, Department of Computer Science, University of Pittsburgh, 1997.