Demand-driven Data Flow Analysis for Communication Optimization *

X. Yuan R. Gupta R. Melhem
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{xyuan, gupta, melhem@cs.pitt.edu}

Abstract

Global communication optimization greatly reduces communication costs in programs compiled for a distributed memory system. However, traditional exhaustive global array data-flow analysis for communication optimizations is expensive and considered to be impractical for large programs. This paper tackles the efficiency problem by proposing a demand-driven analysis approach. In comparison to traditional data flow analysis, demand driven approaches reduce the analysis cost by computing only the data flow information related to the optimizations. Furthermore, the cost of demand driven analysis can be managed by trading space for time or compromising precision for time complexity. A new form of section communication descriptor *is introduced to represent communication requirements concisely and to perform set operations efficiently. By globally propagating descriptors, many communication optimization opportunities can be discovered. A general demand driven algorithm to globally propagate the section communication descriptor is described, and applications of the general algorithm for communication optimization are presented.*

1 Introduction

Although distributed memory systems provide scalability for parallel applications, writing programs with explicit message passing is tedious and error prone. This has motivated considerable research towards developing compilers that relieve programmers from the burden of generating communications [12, 2, 3, 10, 16, 18]. Such compilers take sequential or shared memory parallel programs and generate SPMD programs with explicit message passing. One important task for the compilers is to reduce communication costs. Many communication optimizations, such as, message vectorization, redundant communication elimination, message merging, and overlapping communication with computation have been proposed.

It has been demonstrated that global communication optimizations can greatly reduce communication costs[5, 14]. Two different approaches, one based on data dependence analysis [14] and the other using data flow analysis[11, 8], have been proposed. While the data dependence approach is more efficient in terms of its analysis cost, data flow analysis technique has the advantage of better precision. However, the data-flow frameworks [11, 8] typically propagate information represented in some form of array section descriptor. Due to the complexity of the array section descriptors, the propagation of data flow information can be expensive both in time and space. Furthermore, in traditional data flow approaches, 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* in that not all the data flow solutions at all points are useful. Over–analysis may not only decrease the efficiency, but also effect the precision in the context of communication optimization because of the approximations in array data flow analysis.

These problems in data flow analysis for communication optimization can be alleviated by using demand driven data flow analysis technique. Demand-driven data flow analysis computes the data flow information only when necessary. Thus, it reduces the analysis cost by preventing the over-analysis of a program that occurs when part of the analysis effort is spent on collecting superfluous information. Another advantage is that the analysis cost can be managed. Intermediate results may be recomputed every time they are requested (without saving them) to trade time for space. On the other hand, maintaining a *result cache* can avoid repeated computations of intermediate solutions, which trades space for time. In addition, demand-driven analysis considers the part of program that is textually close to the point where the data flow information is needed. This enables the demand-driven analysis to give reasonable approximation when the analysis is terminated prematurely. Thus, demand-driven analysis can also trade precision for time. Efficient demand driven analysis techniques for scalars can be found in [7].

This paper presents demand driven algorithms for global communication optimizations. A new form of *section communication descriptor*(SCD) is introduced to represent the communication requirements. This descriptor enables the compiler to represent communications concisely and to perform the section opera-

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

tions efficiently. The SCD information is propagated in a demand driven manner through the control flow graph to uncover the opportunities for communication optimizations. General rules and an algorithm that implements these rules for SCD propagations are presented. Applications of the general algorithm to specific communication optimizations, including message vectorization and redundant communication elimination, are discussed.

The rest of the paper is organized as follows. Section 2 describes the program representation. Section 3 introduces SCD and its operations. Section 4 details the demand-driven algorithms for propagating SCDs. Section 5 describes the communication optimization algorithms. Section 6 concludes the paper.

2 Program representation

We consider structured programs that contain conditionals and nested loops, but no arbitrary goto statements. A loop is controlled by a basic induction variable and no statement in the loop contains an assignment to this variable. Furthermore, we assume that all loops are normalized, i.e. the induction variable ranges from 1 to an upper bound with an increment of one. The array references nested in loops 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 the vector representing the loop induction variables and $f_k(\vec{i})$, $k = 1..s$, are affine functions of $\vec{i}$.

A general data flow analysis algorithm that considers loop nesting hierarchies is interval analysis[17]. Our algorithm performs the interval analysis in a demand driven manner. A variant of Tarjan's intervals[17] is used. The analysis is done on one *interval flow graph* G = (N, E), with nodes N and edges E. $ROOT \in N$ is the unique root of G, which is viewed as a header node for the entire program. 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 level has the highest level.

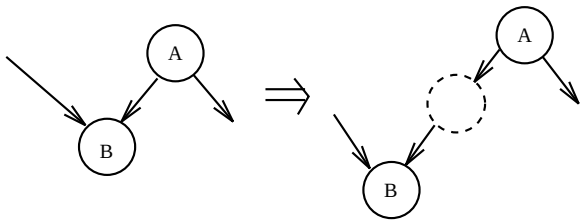


Figure 1: Edge splitting transformation

Let $T(n)$ be the interval whose header is n. For a node m, we define $header(m)$ to be the header of the interval that immediately encloses m. Two special nodes, the header and tail, for each interval are inserted into the control flow graph if necessary, that is, to make the interval single entry and single exit. The analysis requires that there are no *critical edges*, which connect a node with multiple outgoing edges to a node with multiple incoming edges. This can be achieved by edge splitting transformation[11] to the control flow graph as shown in Fig. 1. Fig. 2 shows an example code and its corresponding interval flow graph. Node 9 in the example is an additional node inserted by the edge splitting transformation.

3 Section Communication Descriptor(SCD)

The processor space is considered as an unbounded grid of virtual processors. The abstract processor space is similar to a *template* in High Performance Fortran (HPF) [13], which is a grid over which different arrays are aligned. In the rest of the paper, when we refer to communication, we mean communication on the virtual processor grid.

The Section Communication Descriptor(SCD) is an extension form of an array section descriptor. It describes both an array region and the communication involving the region. A SCD is defined as $< N, D, M >$, where N is an array name, D is the source region of the communication, and M is the descriptor of source-destination mapping. Hence a SCD describes a communication pattern by describing the sources of the communication and the mapping relation of the source and destination.

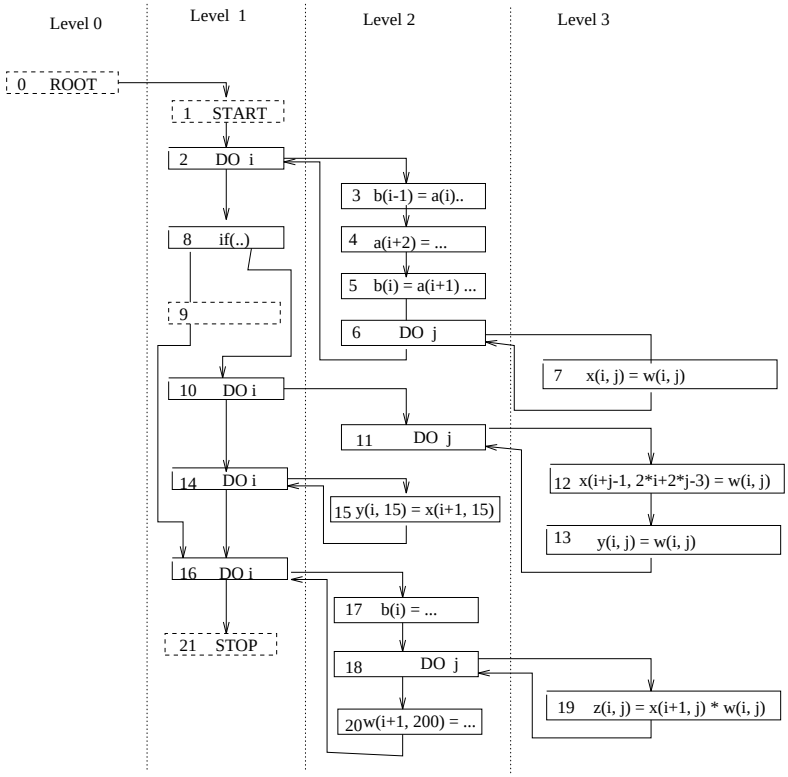
The *bounded regular section descriptor* (BRSD)[4] is used to describe the source region of communications. As discussed in [4], set operations can be efficiently performed over BRSDs. The source region S is a vector of subscript values such that each of its elements is either (1) an expression of the form $\alpha * i + \beta$, where i is a loop index variable and α and β are invariants, (2) a triple $l : u : s$, where l, u and s are invariants, or (3) $\perp$, indicating no information about the subscript value.

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

3.1 Ownership

To calculate the communication requirements of an assignment statement, the compiler must first know the ownership of the arrays, that is, which processor contains which elements. We assume that the arrays are all aligned to a single virtual space by a simple affine function. The alignments allowed are scaling, axis alignment and offset alignment. The mapping from a point $\vec{d}$ in data space to the corresponding point $\vec{v}$ in virtual processor grid can be specified by an alignment matrix M and an alignment offset vector $\vec{\alpha}$. Thus $\vec{v} = M\vec{d} + \vec{\alpha}$. For example, consider the alignments of array w and a in the example program in Fig. 2, the alignment matrices and offset vectors for array w and a are the following:

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




```
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
(s1)  do i = 1, 100
(s2)    b(i-1) = a(i)...
(s3)    a(i+2) = ...
(s4)    b(i) = a(i+1) ...
(s5)    do j = 1, 100
(s6)      x(i, j) = w (i, j)
(s7)    end do
(s8)  end do
(s9)  if (...) then
(s10)   do i = 1, 100
(s11)     do j = 50, 100
(s12)       x(i+j-1, 2*i+2*j-3) = w (i, j)
(s13)       y(i, j) = w(i, j)
(s14)     end do
(s15)   end do
(s16)   do i = 1, 100
(s17)     y(i, 150) = x(i+1, 150)
(s18)   end do
(s19) end if
(s20) do i = 1, 100
(s21)   b(i) = ...
(s22)   do j = 1, 200
(s23)     z(i, j) = x(i+1, j)* w(i, ,j)
(s24)   end do
(s25)   w(i+1, 200) = ...
(s26) end do
```

Figure 2: An example program and its interval flow graph

$$M_a = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \ \vec{\alpha}_a = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

3.2 SCD calculation

Let $\vec{i}$ be the vector of loop indices. When the subscript expressions are affine functions of the loop indices, the 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*. The data access matrix, G, and the data access matrix, $\vec{g}$, describe a mapping from each point in the iteration space to the corresponding point in the data space.

The SCD for each assignment statement can be calculated from the program structure when the communication can be determined statically. In this section, we discuss how the SCD can be calculated from the program when the *owner computes* rule is used. The owner computes rule requires each item referenced on the *rhs* of an assignment statement to be sent to the processor that owns the *lhs*.

Let G_l, $\vec{g}_l$, M_l, $\vec{\alpha}_l$ be the data access matrix, access offset vector, alignment matrix and alignment vector for the left hand side array, and G_r, $\vec{g}_r$, M_r, $\vec{\alpha}_r$ be the corresponding quantities for the right hand side array. The communication required by this statement can be specified by the following relation:

$$Map =< M_r(G_r\vec{i} + \vec{g}_r) + \vec{\alpha}_r, M_l(G_l\vec{i} + \vec{g}_l) + \vec{\alpha}_l >$$

Consider the communication in statement $s12$ for Fig. 2. The compiler can obtain from the program the following the data access matrices, access offset vectors, alignment matrices and alignment vectors.

$$M_x = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \ \vec{\alpha}_x = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

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

$$G_l = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix}, \ \vec{g}_l = \begin{pmatrix} -1 \\ -3 \end{pmatrix}$$

$$G_r = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \ \vec{g}_r = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Thus, source-destination mapping for the statement is $< (2 * j, i + 1), (i + j - 1, 2 * i + 2 * j - 3) >$.

The SCD descriptor representation can capture the communication pattern of the array assignment statement with the following restrictions:

- Each subscript expression of the right hand side array is of the form $\alpha * i + \beta$, where α and β are invariants and i is a loop index variable.

- Each subscript expression of the left hand side array is a linear combination of the loop index variables.

With these restrictions, the source region can be expressed using BRSD in SCD descriptors and the *source* in the mapping relation will be a vector whose elements are of the form $\alpha * i + \beta$, where α and β are constants and i is a loop index variable.

3.3 Operations on SCD

Operations, such as intersection, difference and union, on the SCD descriptors are needed in our analysis. In most cases the 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 approximation or list representation. We define two mapping relations M_1 and M_2 to be unrelated if and only if: (1) M_1 does not include M_2 and (2) M_2 does not include M_1. Next we describe how to test whether a mapping is a subset of another mapping before we describe the operations.

Subset testing. Testing whether a mapping relation is a subset of another mapping relation is an important operation since most of other operations rely on this operation. For example, the equivalence testing can be broken down into two subset testings. Testing that a relation M_1 ($=< s_1, d_1 >$) is a subset of another relation M_2 ($=< s_2, d_2 >$) is done by checking if the equations $s_1 = s_2$ and $d_1 = d_2$ have a valid solution with variables in M_1 treated as constants and variables in M_2 treated as variables. Note that since the elements in s_1 and s_2 are of the form $\alpha * i + \beta$, to determine whether the equations have a valid solution generally takes constant amount of time. For example, to determine whether the mapping relation $M_1 =< (1, i), (1, i + 1) >$ is a subset of $M_2 =< (i, j), (i, j + 1) >$. The system will solve the following equations assuming the variable i in M_1 is renamed to k.

$$i = 1, j = k, i = 1, j + 1 = k + 1$$

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 constituting the common part of their array sections that have the same mapping relation. This operation is given by

$< N_1, D_1, M_1 > \cap < N_2, D_2, M_2 >$
$= \phi$, 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. In our analysis, difference operation only occurs when elements in the descriptor are killed by some definitions. The kill sets are always represented as descriptors with $\top$ mapping function. Hence, this special case is sufficient for our analysis.

$< N_1, D_1, M_1 > - < N_2, D_2, \top >$
$= < N_1, D_1, M_1 >$, 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 their array section. 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

After multiple union operations, the descriptor can be a large list which will hinder the efficiency of the analysis. When such situation occurs, the demand driven algorithm may as well assume the conservative results and terminate the analysis to trade precision for time. Note that all array data flow analysis using BRSD suffer from the same difficulty, since the union operation is not closed in the BRSD. Demand driven analysis can give better approximation than the traditional exhaustive schemes.

4 Demand-driven SCD propagation

Many communication optimization opportunities can be uncovered by propagating SCDs globally. Propagating SCDs backward can find the earliest point to place the communication, while propagating SCDs forward can find the lastest point where the effect of the communication is destroyed. Thus both backward and forward propagation is useful in communication optimization. Since forward and backward propagation are similar, we will only focus on backward propagation of SCDs.

We present the generic demand driven algorithms to propagate SCDs through the interval flow graph representation of a program. In the generic algorithms, the SCDs are propagated until they cannot be propagated, i.e. all the elements in the SCD are killed. However, in practice, the compiler may choose to terminate the propagation prematurely, i.e. when there are still elements in SCDs, to save analysis time. In this case, since the analysis starts from the points that contribute to the optimizations, the points that are textually close to the starting points, where most of the optimization opportunities present, have been considered. This gives the demand driven algorithm the ability to trade precision for time.

The analysis techniques are the reverse of the interval-analysis [9]. Specially, by reversing the information flow associated with program points, we derive a system of request propagation rules. In the propagation, only a single interval is under consideration at a given time. Hence, the propagations are logically done in an acyclic flow graph. During the propagation, a SCD may expand when it is propagated out of a loop. When a set of elements of a SCD is killed inside a loop, the set is propagated into the loop to determine the exact point where the elements are killed. 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 data flow *propagation request* is $< S, n, [UP|DOWN], level, cnum >$, where S is a SCD, n is a node in the flow graph, constants UP and $DOWN$ indicate whether the request is an upward propagation or a downward propagation, *level* indicates at which level is the request and the value *cnum* indicates which child node of n has triggered the request. A special value -1 for *cnum* is used as the indication of the beginning of propagation. Hence, a compiler can start a propagation by placing an upward request with $cnum = -1$. Downward propagation are triggered automatically when loop header nodes are encountered. As a result, downward propagations always start from loop header nodes. 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 the 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. The following subsection describes the propagation rules. We assume that node n has k successors.

4.1 Propagation rules

RULE 0: upward propagation, initial node. As we mentioned earlier, the compiler starts a upward propagation from node s by placing a upward 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
    for all $m \in pred(s)$
      Let s be m's jth child
      request($< SCD, m, UP, level, j >$)

RULE 1: upward propagation, regular node. Here regular node includes all nodes other than loop header nodes and the initial node. The requests on a regular node trigger an action based on SCD sets and the local information and the propagation of a SCD further upwards. In the specification of the rule shown below, functions *action* and *local* are dependent 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
    for all $m \in pred(n)$
      Let n be m's jth child
      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 the final solution.

RULE 2: upward propagation, 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 the action based on S and K_n, propagating some information past above the loop and triggering a downward propagation to propagate the information into the loop nest.

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
      for all $m \in pred(n)$
        Let n be m's jth child
        request($< S - K_n, m, UP, level, j >$)
    if ($S \cap K_n \neq \phi$) then
      request($< S \cap K_n, n, DOWN, level, -1 >$)

The summary function can be calculated either before hand or in demand driven manner. In section 4.2, we will describe the algorithm to calculate the summary. K_n is the summary information of the loops representing the 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* of the downward request is set to -1 to indicate that it is the start of the downward propagation.

RULE 3: upward propagation, 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 propagation in to the lower level. At the same time, this request triggers a downward propagation for the set that must stay in the loops. 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
      for all $m \in pred(n)$
        Let n be m's jth child
        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 the same definition as in [11].

For a SCD descriptor S, $expand(S, k, low : high)$ is a function which replaces all single data item references $\alpha * k + \beta$ used in any array section descriptor D in S by the triple $(\alpha * low + \beta : \alpha * high + \beta : \alpha)$. The set def includes all the definitions that are the sources of an flow-dependence to the array region propagated.

RULE 4: downward propagation, initial node. A downward propagation always starts from a loop header node with a $cnum = -1$ 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 initial available set for iteration i, that is, we must subtract the elements killed in iterations $i+1$ to $high$. This propagation prepares the downward propagation into the loop body by shrinking the SCD.

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

RULE 5: downward propagation: regular node. For regular node, the downward propagation is the similar to the upward propagation rule.

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
    for all $m \in pred(n)$
      Let n be m's jth child
      request($< S - kill_n, m, DOWN, level, j >$)

RULE 6: downward propagation: same level loop header node. When downward propagation reaches a loop header, if the loop header is in lower level, than the propagation stops; otherwise, the loop header is in the same level as the request, and we must propagate appropriate SCD through the loop header and generate further downward propagation request into nested loops.

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
      for all $m \in pred(n)$
        Let n be m's jth child
        request($< S - K_n, m, DOWN, level, j >$);
    if ($S \cap K_n \neq \phi$) then
      request($< S \cap K_n, n, DOWN, level, -1 >$);

4.2 Summary calculation

During the request propagation, the summary information of an interval is needed when a loop header is encountered. In this section, 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 the data flow information from the tail node to the header node in the interval using the following data flow equation:

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

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

Summary_kill(n)
  $K_{out}(tail) = \phi$
  for all $m \in T(n)$ and in backward order
    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))

Figure 3: Summary kill calculation.

4.3 Request propagation algorithm

The demand driven algorithms *uprequest* and *downrequest* that implement the propagation rules are shown in Fig. 4 and Fig. 5 respectively. *Uprequest* takes as its input a request r and propagates r to the earliest points that r can reach. The algorithm first propagates the upward requests and then calls the *downrequest procedure* to propagate the request in the downward direction. A worklist is maintained to store the pending requests. The worklist is initialized with the input request r. All the request currently in the worklist can potentially be propagated further. In each step a request is removed from the worklist and translated according to the propagation rule associated with the node under inspection. The request resulting from this translation is merged with the previous requests and added to the worklist. The algorithm terminates when the worklist becomes empty. Data structures *dset* is used to maintain the downward requests set that are not being processed. This algorithm focuses on the propagation of the request and omits the actions in each node.

4.4 Example

Let us use an example to see how the propagation works. Consider propagating the communication for array w in statement $s23$ in the example program in Fig. 2. The SCD describing the communication is $< w, (i, j) < (2 * j, i + 1), (i, j) >>$.

```
(1)  uprequest(r)
(2)    Let r =< S, s, UP, level, −1 >
(3)    for each m ∈ N do request[m] = φ.
(4)    dset = φ
(5)    /* propagate initial request, RULE 0 */
(6)    for all m ∈ pred(s) do
(7)      Let m be s's jth successor
(8)      request[m] = request[m] +
(9)      {< S, m, UP, level, j >}
(10)     worklist = worklist + {m}
(11)   while worklist ≠ φ do
(12)     remove a node n from worklist
(13)     if n is a loop header then
(14)       Let request[n] =< S, n, UP, level, 1 >
(15)       if LEVEL(n) < level then
(16)         /* rule 3, go to lower level */
(17)         calculate summary K_n
(18)         outside = < expand(S, i, low : high)−
(19)         ∪_def expand(def, i, low : high)
(20)         inside = < expand(S, i, low : high)∩
(21)         ∪_def expand(def, i, low : high)
(22)         if (outside ≠ φ) then
(23)           for all m ∈ pred(n) do
(24)             Let n be m's ith child
(25)             request[m] = request[m] +
(26)             {< outside, m, UP, level − 1, i >};
(27)             worklist = worklist + {m}
(28)         if (inside ≠ φ) then
(29)           dset = dset +
(30)           {< inside, n, DOWN, level − 1, −1 >};
(31)       else /* rule 2, same level */
(32)         calculates K_n for interval T(n)
(33)         if (S − K_n ≠ φ) then
(34)           for all m ∈ pred(n)
(35)             Let n be m's ith child
(36)             request[m] = request[m] +
(37)             {< S − K_n, m, UP, level, i >}
(38)             worklist = worklist + m
(39)         if (S ∩ K_n ≠ φ) then
(40)           dset = dset +
(41)           {< S ∩ K_n, n, DOWN, level, −1 >}
(42)     else /* regular node, rule 1 */
(43)       if all m's successors place a request then
(44)         Let S_1, ..., S_k be the SCD in the request
(45)         S = S_1 ∩ S_2 ... ∩ S_k
(46)         if (S − kill_n ≠ φ) then
(47)           for all m ∈ pred(n)
(48)             Let n be m's ith child
(49)             request[m] = request[m] +
(50)             {< S − kill_n, m, UP, level, i >}
(51)             worklist = worklist + {m}
(52)       else
(53)         if there is a request that can be propagated
(54)           insert m back into the worklist
(55)   while dset ≠ φ do
(56)     /* processing the downward propagation */
(57)     remove r from dset.
(58)     call downrequest(r)
```

Figure 4: Upward request propagation algorithms

```
(1)  downrequest(r)
(2)    Let r =< S, s, DOWN, level, −1 >
(3)    for each m ∈ N do request[m] = φ.
(4)    request[s] = r, worklist ← {s}
(5)    while worklist ≠ φ do
(6)      remove a node n from worklist
(7)      if n is a loop header then
(8)        Let request[n] =< S, n, DOWN, level, cnum >
(9)        if (cnum = −1) then
(10)         /* RULE 4, initial node */
(11)         calculate summary of interval T(n)
(12)         Let node l be the tail node of the interval
(13)         inside = S − ∪_def expand(def, i, i + 1 : high)
(14)         request[l] = {< inside, l, DOWN, level + 1, 1 >}
(15)         worklist = worklist + {l}
(16)       else /* RULE 6 */
(17)         if (LEVEL(n) < level) then return
(18)         else
(19)           calculates the summary, K_n, of interval T(n)
(20)           if (S − K_n ≠ φ) then
(21)             for all m ∈ pred(n)
(22)               Let n be m's jth child
(23)               request[m] = request[m] +
(24)               {< S − K_n, m, UP, level, j >}
(25)               worklist = worklist + m
(26)           if (S ∩ K_n ≠ φ) then
(27)             request[n] = request[n] +
(28)             {< S ∩ K_n, n, DOWN, level, −1 >}
(29)             worklist = worklist + {n}
(30)     else /* regular node, RULE 5 */
(31)       if all m's successors place a request then
(32)         Let S_1, ..., S_k be the SCD in the request
(33)         S = S_1 ∩ S_2 ... ∩ S_k
(34)         if (S − kill_n ≠ φ) then
(35)           for all m ∈ pred(n)
(36)             Let n be m's jth child
(37)             request[m] = request[m] +
(38)             {< S − kill_n, m, DOWN, level, j >}
(39)             worklist = worklist + {m}
(40)       else
(41)         if there is a request that can be propagated
(42)           insert m back into the worklist
```

Figure 5: Downward request propagation algorithms

The propagation is initiated by the request $request(<< w, (i,j), < (2*j, i+1), (i,j) >>, 19, UP, 3, -1 >)$ at node 19. By rule 0, the request propagates to node 18 as $request(<< w, (i,j), < (2*j, i+1), (i,j) >>, 18, UP, 3, 1 >)$. Since node 18 is a header node in the lower level than the request ($LEVEL(18) = 2, level = 3$), rule 3 is applied and it generates $request(<< w, (i, 1:200), < (2*j, i+1), (i,j) >>, 17, UP, 2, 1 >)$ at node 17. Note that since there is no kill of w in interval $\{19\}$, no downward propagation is generated. By applying rule 1 at node 17, $request(<< w, (i, 1:200), < (2*j, i+1), (i,j) >>, 16, UP, 2, 1 >)$ will be propagated to node 16. Applying rule 3 at node 16, SCD $outside =< w, \{(1:100, 1:199), (1, 200)\}, < (2*j, i+1), (i,j) >>$ will be propagated out of the loops and SCD $inside =< w, (2:100, 200), < (2*j, i+1), (i,j) >>$ will stay inside the loop. The

requests, $R1 = request(< outside, 9, UP, 1, 1 >)$, $R2 = request(< outside, 14, UP, 1, 1 >)$, and $R3 = request(< inside, 16, DOWN, 1, -1 >)$ are generated.

Request $R1$ will pass through node 9 with rule 1. Request $R2$ will pass through node 14 and node 15 with rule 2. Finally, $request(< outside, 0, UP, 0, 1 >)$ will reach the ROOT node and the upward propagation completes. Request $R3$ triggers a downward propagation from node 16. Rule 4 results in $request(<< w, (2 : i+1, 200), < (2*j, i+1), (i, j) >> , 20, DOWN, 2, 1 >)$ at node 20. Applying rule 5 in node 20 yields $request(<< w, (2 : i, 200), < (2 * j, i + 1), (i, j) >>, 18, DOWN, 2, 1 >)$. Rule 6 will be used in node 18 and yields $request(<< w, (2 : i, 200), < (2 * j, i + 1), (i, j) >>, 17, DOWN, 2, 1 >)$. Since there are no kills in interval 19 and node 17, $request(<< w, (2 : i, 200), < (2 * j, i + 1), (i, j) >> , 16, DOWN, 2, 1 >)$ will reach node 16 and the downward propagation terminates.

5 Demand-driven Optimization

In the previous section, we presented a general demand driven algorithm to propagate SCDs through the interval flow graph. This section presents applications of the general algorithms for specific communication optimizations. Two applications, *message vectorization* and *redundant communication elimination* are discussed. These optimizations can be achieved by propagating the SCD through the flow graph.

5.1 Message vectorization

Message vectorization tries to hoist communication 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 can be generated outside the loop body. This optimization can be done by propagating the SCD for an assignment statement in backward direction in the flow graph. Since in message vectorization, communication is only be hoisted out of a loop, only *upward* propagation is necessary.

For the compiler to perform message vectorization for a statement s, which corresponds to node n in the interval flow graph, the compiler will calculate the SCD set, $SCD(n)$, representing the communication for the statement, and initiate a propagation with $request(< SCD(n), n, UP, LEVEL(n), -1)$. Slightly modified version of rules 0, 1, 2, 3 such that the downward propagation will not be triggered can be used to propagate the SCD. Besides, the local action on loop header node will record which communications should be performed before it. The parts of SCDs that can be propagated out of the loop (not killed within loop body) represent the communications that can be vectorized. The communications of these parts will be placed immediately preceding the loop. The communication of elements that are killed inside loops stay at the points preceding their references.

As an example, consider vectorizing the communication of array w in statement $s23$ in the example program in Fig. 2. The request $request(<< w, (i, j), < (2 * j, i + 1), (i, j) >>, 19, UP, 3, -1 >)$ will be placed by the compiler.

```
(s1)  do i = 1, 100
        C1 : (< a,(i),< (i,1),(i − 1,1) >>, i = 1..100)
(s2)    b(i-1) = a(i)...
(s3)    a(i+2) = ...
        C2 : (< a,(i + 1),< (i + 1,1),(i,1) >>, i = 1..100)
(s4)    b(i) = a(i+1) ...
(s5)    do j = 1, 100
          C3 : (< w,(i, j),< (2 ∗ j, i + 1),(i, j) >>,
          i = 1..100, j = 1..100)
(s6)      x(i, j) = w (i, j)
(s7)    end do
(s8)  end do
(s9)  if (...) then
(s10)   do i = 1, 100
(s11)     do j = 50, 100
            C4 : (< w,(i, j),< (2 ∗ j, i + 1),
            (i + j − 1, 2 ∗ i + 2 ∗ j − 3) >>,
            i = 1..100, j = 50..100)
(s12)       x(i+j-1, 2*i+2*j-3) = w (i, j)
            C5 : (< w,(i, j),< (2 ∗ j, i + 1),(i, j) >>,
            i = 1..100, j = 50..100)
(s13)       y(i, j) = w(i, j)
(s14)     end do
(s15)   end do
(s16)   do i = 1, 100
          C6 : (< x,(i + 1,150),< (i + 1,150),(i,150) >>,
          i = 1..100)
(s17)     y(i, 150) = x(i+1, 150)
(s18)   end do
(s19) end if
(s20) do i = 1, 100
(s21)   b(i) = ...
(s22)   do j = 1, 200
          C7 : (< x,(i + 1, j),< (i + 1, j),(i, j) >>,
          i = 1..100, j = 1..200)
          C8 : (< w,(i, j),< (2 ∗ j, i + 1),(i, j) >>,
          i = 1..100, j = 1..200)
(s23)     z(i, j) = x(i+1, j)* w(i, ,j)
(s24)   end do
(s25)   w(i+1, 200) = ...
(s26) end do
```

Figure 6: Communication before message vectorization

As discussed in the example in section 4.4, communication $< w, \{(1 : 100, 1 : 199), (1, 200)\}, < (2 * j, i + 1), (i, j) >>$ can be propagated out of the outermost loop (s20 – s26) while communication $< w, (2 : 100, 200), < (2 * j, i + 1), (i, j) >>$ can be propagated out of the inner loop (s22 – s24). Hence, the communication for array w in statement $s23$ can be vectorized by placing the communication $< w, \{(1 : 100, 1 : 199), (1, 200)\}, < (2 * j, i + 1), (i, j) >>$ before statement $s20$ and communication $< w, (2 : 100, 200), < (2 * j, i + 1), (i, j) >>$ before statement $s22$. Consider vectorizing the communication for array a in statement $s4$. By applying rule 0 in node 5, rule 1 in nodes 3 and 4, $request(< a, (i+1), < (i+1, 1), (i, 1) > , 2, UP, 2, 1 >$ can reach node 2. Applying rule 3, communication $< a, (2 : 2), < (i + 1, 1), (i, 1) >>$ can be propagated out of the loop and communication $< a, (3 : 101), < (i + 1, 1), (i, 1) >>$ must stay in the

loop. Hence, only the communication for a(2) can be hoisted out of the loop and put before statement $s1$ while all other communication must be placed before $s4$. Fig. 6 shows the communications in the program in Fig. 2. Fig. 7 shows the communications after vectorization. We use SCD with quantifier of the index range to represent the communications. For example, $C_1 : (< a,(i),< (i,1),(i-1,1) >>, i = 1..100)$ denotes the communication $< a,(i),< (i,1),(i-1,1) >>$ will be performed in iteration 1 to 100. $C_5^1 : (< w,(1:100,50:100),< (2*j,i+1),(i,j) >>)$ denotes that communication $< w,(1:100,50:100),< (2*j,i+1),(i,j) >>$ will be performed once at the point.

```
      C_1^1 : (< a,(1:2),< (i,1),(i-1,1) >>)
      C_2^1 : (< a,(2:2),< (i+1,1),(i,1) >>)
      C_3^1 : (< w,(1:100,1:100),< (2*j,i+1),(i,j) >>)
(s1)  do i = 1, 100
        C_1^2 : (< a,(i),< (i,1),(i-1,1) >>, i = 3..100)
(s2)    b(i-1) = a(i)...
(s3)    a(i+2) = ...
        C_2^2 : (< a,(i+1),< (i+1,1),(i,1) >, i = 2..100)
(s4)    b(i) = a(i+1) ...
(s5)    do j = 1, 100
(s6)      x(i, j) = w (i, j)
(s7)    end do
(s8)  end do
(s9)  if (...) then
        C_4^1 : (< w,(1:100,50:100),< (2*j,i+1),
        (i+j-1,2*i+2*j-3) >>)
        C_5^1 : (< w,(1:100,50:100),< (2*j,i+1),(i,j) >>)
(s10)   do i = 1, 100
(s11)     do j = 50, 100
(s12)       x(i+j-1, 2*i+2*j-3) = w (i, j)
(s13)       y(i, j) = w(i, j)
(s14)     end do
(s15)   end do
        C_6^1 : (< x,(2:101,150),< (i+1,150),(i,150) >>)
(s16)   do i = 1, 100
(s17)     y(i, 150) = x(i+1, 150)
(s18)   end do
(s19) end if
        C_7^1 : (< x,(2:101,1:200),< (i+1,j),(i,j) >>)
        C_8^1 : (< w,{(1:100,1:199),(1,200)},
        < (2*j,i+1),(i,j) >>,
(s20) do i = 1, 100
(s21)   b(i) = ...
        C_8^2 : (< w,(i,200),< (2*j,i+1),(i,j) >>,
        i = 2..100)
(s22)   do j = 1, 200
(s23)     z(i, j) = x(i+1, j)* w(i, ,j)
(s24)   end do
(s25)   w(i+1, 200) = ...
(s26) end do
```

Figure 7: Communication after message vectorization

5.2 Redundancy elimination

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

For the compiler trying to find all the communications that can be subsumed by the communication S at node n, the compiler initiates a propagation with $request(< S, n, UP, LEVEL(n), -1)$. During the propagation, when ever 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 node is reached. The propagation follows the rules discussed at section 4. However, downward propagation into the nested loops that is 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 set are propagated to the branch node. For example, let a branch node X has two successors, Y and Z, and Y places $request(< S_1, X, UP, level, 1 >)$ and Z places $request(< S_2, X, UP, level, 2 >)$, the local action at node X will place communication $S_1 - S_1 \cap S_2$ at node Y and communication $S_2 - S_1 \cap S_2$ at node Z. Note that the redundant communication elimination can be carried out together with the message vectorization phase.

For example, propagating communication C_7^1 before $s20$ in Fig. 7 will trigger request into two branchs of the **if** statement ($s9$). In the **then** branch, C_7^1 will be killed in statement $s10$ and will subsume communication C_6^1, while the **else** branch can pass communication C_7^1 to $s9$. The action in node $s9$ will keep C_7^1 at the **else** branch. Note that the node in the interval flow graph that corresponds to the **else** branch is created by the edge splitting transformation. Propagating the communication C_2^2 in Fig. 7 causes the communication after statement $S3$ to be placed one iteration earlier and it completely subsumes the communication C_1^2. The communication C_8^1 before statement $s20$ in Fig. 7 can be propagated to the root node. Along the way, it subsumes the communication C_3^1 before $s1$ and C_5^1 before $s10$. The communications after propagating communication C_2^2, C_7^1 and C_8^1 for redundant communication elimination in the program in Fig. 7 is shown in Fig. 8.

6 Conclusion

Data flow analysis provides precise information for communication optimization. However, it suffers from efficiency problem due to the complexity of the information required by communication optimizations. In this paper, we proposed using demand driven data flow analysis for communication optimization. Our approach has the advantage of no over–analysis and manageable cost. A generic algorithm is presented and applications of the algorithm on communication optimizations are discussed. We are currently implementing these algorithms.

$C_1^1 : (< a, (1:2), < (i,1), (i-1,1) >>)$
$C_2^1 : (< a, (2:2), < (i+1,1), (i,1) >>)$
$C_8^1 : (< w, (1:100, 1:199), (1, 200),$
$< (2*j, i+1), (i,j) >>)$

```
(s1)  do i = 1, 100
(s2)    b(i-1) = a(i)...
(s3)    a(i+2) = ...
        C_2^2 : (< a, (i+2), < (i+1,1), (i,1) >, i = 1..99)
(s4)    b(i) = a(i+1) ...
(s5)    do j = 1, 100
(s6)      x(i, j) = w (i, j)
(s7)    end do
(s8)  end do
(s9)  if (...) then
        C_4^1 : (< w, (1:100, 50:100), < (2*j, i+1),
        (i+j-1, 2*i+2*j-3) >>)
(s10)   do i = 1, 100
(s11)     do j = 50, 100
(s12)       x(i+j-1, 2*i+2*j-3) = w (i, j)
(s13)       y(i, j) = w(i, j)
(s14)     end do
(s15)   end do
        C_7^1 : (< x, (2:101, 1:200), < (i+1, j), (i,j) >>)
(s16)   do i = 1, 100
(s17)     y(i, 150) = x(i+1, 150)
(s18)   end do
(s18) else
        C_7^1 : (< x, (2:101, 1:200), < (i+1, j), (i,j) >>)
(s19) end if
(s20) do i = 1, 100
(s21)   b(i) = ...
(s22)   do j = 1, 200
(s23)     z(i, j) = x(i+1, j)* w(i, ,j)
(s24)   end do
(s25)   w(i+1, 200) = ...
        C_8^2 : (< w, (i+1, 200), < (2*j, i+1), (i,j) >>,
        i = 1..99)
(s26) end do
```

Figure 8: Communication after redundant communication elimination

References

[1] F.E. Allen and J. Cocke "A Program Data Flow Analysis Procedure." *Communication of the ACM*, 19(3):137-147, March 1976.

[2] S. P. Amarasinghe and M. S. Lam "Communication Optimization and Code Generation for Distributed Memory Machine." In Proceedings ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation, June 1993.

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

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

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

[6] J.F. Collard, d. Barthou and P. Feautrier "Fuzzy Array Dataflow analysis." In *5th ACM SIGPLAN Symposium on Principle & Practice of Parallel Programming*, July 1995, Santa Barbara, CA.

[7] E. Duesterwald, R. Gupta and M. L. Soffa "Demand-driven Computation of Interprocedural Data Flow" In *Symposium on Principles of Programming Languages*, Jan. 1995, San Francisco, CA.

[8] C. Gong, R. Gupta and R. Melhem "Compilation Techniques for Optimizing Communication on Distributed-Memory Systems" In *International Conference on Parallel Processing*, Vol II, pagges 39-46, August 1993.

[9] M. Gupta and E. Schonberg "A Framework for Exploiting Data Availability to Optimize Communication." In *6th International Workshop on Languages and Compilers for Parallel Computing*, LNCS 768, pp 216-233, August 1993.

[10] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, K.Y. 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.

[11] 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.

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

[13] High Performance Fortran Forum. "High Performance Fortran Language specification." version 1.0 Technique Report CRPC-TR92225, Rice University, 1993.

[14] K. Kennedy and N. Nedeljkovic "Combining dependence and data-flow analyses to optimize communication." In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[15] I. Kim and M. Wolfe "Communication Analysis for Multicomputer Compilers." In *proceeding of the IFIP WG10.3 Working conference on Parallel Architectures and Compilation Techniques*, PACT'94, pages 101–109, Montreal, Canada, August, 1994.

[16] A. Rogers and K. Pingali "Process decomposition through locality of reference." In *Proc. SIGPLAN'89 conference on Programming Language Design and Implementation*, pages 69-80, June 1989.

[17] R.E. Tarjan "Testing flow graph reducibility." *Journal of Computer and System Sciences*, 9:355-365, 1974.

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