Interprocedural Partial Redundancy Elimination and Its Application To Distributed Memory Compilation *

Gagan Agrawal and Joel Saltz and Raja Das
UMIACS and Department of Computer Science
University of Maryland
College Park, MD 20742
(301)-405-2756
{gagan, saltz,raja}@cs.umd.edu

Abstract

Partial Redundancy Elimination (PRE) is a general scheme for suppressing partial redundancies which encompasses traditional optimizations like loop invariant code motion and redundant code elimination. In this paper we address the problem of performing this optimization interprocedurally. We use interprocedural partial redundancy elimination for placement of communication and communication preprocessing statements while compiling for distributed memory parallel machines.

1 Introduction

Partial Redundancy Elimination (PRE) is a well known technique for optimizing code by suppressing partially redundant computations. It encompasses traditional optimizations like invariant code motion and redundant computation elimination. It is widely used in optimizing compilers for performing common subexpression elimination and strength reduction. More recently, it has been used for more complex code placement tasks like placement of communication statements while compiling for parallel machines [12, 15].

A number of schemes for partial redundancy elimination have been proposed in literature [10, 11, 20, 19, 25], but are largely restricted to optimizing code within a single procedure. All these schemes perform data flow analysis on Control Flow Graph (CFG) of the procedure. In this paper, we address the problem of performing partial redundancy elimination interprocedurally. There are several difficulties in extending the existing intraprocedural algorithms for application on a full program, rather than a single procedure. First, a full program representation is required which will allow efficient data flow analysis, while maintaining sufficient precision to allow useful transformations and to ensure safety and correctness of transformations. Renaming of formal parameters across procedures must be correctly done while propagating data flow information. Calling context of procedures must be correctly accounted for, also correctness and safety must be maintained if a procedure is called at multiple sites with different sets of parameters.

We have developed an Interprocedural Partial Redundancy Elimination algorithm (IPRE). Our method is applicable to arbitrary recursive programs and arbitrary control flow within each procedure.

We have used interprocedural partial redundancy elimination for optimizing placement of communication statements and communication preprocessing statements in distributed memory compilation. We have implemented our scheme using the the existing Fortran D compilation system [17] as infrastructure. We have shown significant performance gains by optimizing placement of communication preprocessing statements.

The rest of this paper is organized as follows. In Section 2, we discuss how interprocedural partial redundancy elimination is required for placement of communication statements and communication preprocessing statements while compiling for distributed memory parallel machines. In Section 3, we briefly review an intraprocedural partial redundancy elimination scheme. The program representation used in interprocedural data flow analysis is stated in Section 4. Interprocedural partial redundancy elimination is presented in Section 5. In Section 6, we present experimental results to evaluate the effectiveness and cost of our scheme. In Section 7, we mention related work. We conclude in Section 8. Intraprocedural partial redundancy scheme is presented in the appendix.

2 Distributed Memory Compilation

In compiling programs for execution on distributed memory parallel machines, an important consideration is optimizing communication between processors. Since the existing machines have relatively large communication latencies, communication overhead can be reduced if message aggregation is done, i.e. each processor sends a small number of large messages. There are several cases in which the set of data elements to be communicated between the processors cannot be determined at compile-time. This can happen because data

*This work was supported by NSF under grant No. ASC 9213821 and by ONR under contract No. N00014-93-1-0158. The authors assume all responsibility for the contents of the paper.

```
Real    X(m), Y(m)      ! data arrays
Integer IA(n)           ! indirection array



forall  i  =  1, n
  X(i)   =   X(i)  +  Y(IA(i))
```

```
C       Build the required schedule
        DS    =    Sched(.. parameters ..)
C       Communicate data using the schedule build above
        Call Data_Move(Y,DS)

        do  10  i  =    1, n_local
        X(i)  =  X(i)  +  Y(IA_local(i))
```

Figure 1: Compiling an parallel loop which accesses data through an indirection array. Sequential code is shown in the left and SPMD code is shown in the right.

is accessed using indirection arrays, data distribution may not be known at compile-time, number of processors on which the program is to be run is not known till runtime or due to the presence of symbolic loop bounds and strides in a parallel loop. In these cases, communication can be optimized by placing a preprocessing statement, which determines the set of data elements to be communicated between the processors at runtime. The preprocessing statement stores this information in a data-structure called *communication schedule* [24]. A collective communication routine then performs the data movement, using the information in the communication schedule. This ensures that for a parallel loop, each processor packages the set of data elements it wants to send to any other processor in a single message.

In Figure 1, we show how SPMD code can be generated for the given loop in which data is accessed through an indirection array (IA). A communication schedule is generated by a call to *Sched*, which analyzes the contents of array IA to determine the exact communication required. The required data elements are sent or received by the *Data_Move* primitive.

The reason for separation of these two phases of communication is that the result of preprocessing can be used for communicating several times. Compiler analysis has been developed for analyzing the data access patterns associated with a given parallel loop and inserting calls to appropriate communication preprocessing routines and collective communication routines [2, 3, 9]. After such an initial analysis at a single parallel loop or a single procedure level, placement of these statements must be optimized interprocedurally. Large scientific and engineering applications often present opportunities for reusing communication schedule several times and it is important to do this optimization to obtain reasonable performance. We, therefore, identify two optimization problems, *communication schedule generation placement* and *communication placement*. Partial redundancy elimination can be applied interprocedurally for solving both these optimization problems. Partial redundancy elimination encompasses loop invariant code motion and redundant code elimination and has been widely used intraprocedurally to improve the runtime performance of codes. We believe that it can be applied interprocedurally to optimize placement of communication preprocessing statements and communication statements.

3 Intraprocedural Redundancy Elimination

The details of interprocedural redundancy elimination we present are derived from the intraprocedural node based method of Dhamdhere [10], also referred to as Modified Morel Renvoise Algorithm (MMRA). Detailed data flow equations and the meaning of the terms used are given in the appendix. All terms used are for a particular computation, e.g $\text{AVIN}_{\mathcal{C}}(i)$ is the availability of the computation $\mathcal{C}$ at the beginning of node i. Whenever there is no scope for ambiguity, the subscript is dropped (as in the equations given in this paper).

PRE considers subexpressions or *computations* as candidates for placement. *Transparency* of a basic block means that the none of variables involved in the computation are modified in the basic block. Based upon transparency, two properties, *availability* and *partial availability* are computed for beginning and end of each basic block (denoted respectively as AVIN, PAVIN, AVOUT and PAVOUT for each basic block). Availability of a computation at a point p in a procedure means that this computation is currently placed at all the paths leading to p and if this computation is placed at p, it will have the same result as the last computation on any of the paths. Partial availability is a weaker property, which means that the computation is currently placed on at least one control flow path leading to p and if it is placed at p, it will have the same result as the last computation on at least one of the paths. A computation placed at p is partially redundant if it is partially available at p.

Next, for each basic block in the program, properties PPIN (possible placement at the beginning) and PPOUT(possible placement at the end) are computed. PPIN reflects the fact that it is feasible and profitable to hoist the computation occuring in this node (or a computation which has been hoisted into this node). PPOUT indicates that it is safe to place the computation at the exit of this node. INSERT determines if a computation is to be inserted at the end of the a block as a result of the optimization and DEL determines if the computation in this node has become redundant and can be deleted.

4 Program Representation

4.1 Definition

We assume that a variable is either global to the entire program or is local to a single procedure. We further assume that all parameters are passed by reference. We do not consider the possibility of aliasing in our discussion.

Each procedure has one or more return statements, which end the invocation of this procedure. We define a basic block to consist of consecutive statements in the program text without any procedure calls or return statements, and no branching except at the beginning and end. A procedure can then be partitioned into a set of basic blocks, a set of call statements and a set of return statements. Each call statement is a *call site* of the procedure invoked there. In general, a procedure can be invoked at several call sites in the program.

For the purpose of performing interprocedural PRE on the full program, we have defined the following representation. Intuitively, the idea is to construct *blocks of code* within each procedure. A block of code comprises of basic blocks which do not have any call statement between them. In the directed graph we define below, each edge e corresponds to a block of code $B(e)$. The nodes of the graph help clarify the control flow relationships between the blocks of code.

Full Program Representation: (*FPR*) is a directed multigraph $G = (V, E)$, where the set of nodes V consists of an entry node and a return node for each procedure in the program. For procedure i, the entry node is denoted by s_i and the return node is denoted by r_i. Edges are inserted in the following cases:

1. Procedures i and j are invoked by procedure k at call sites cs_1 and cs_2 respectively and there is a path in CFG of k from cs_1 to cs_2 which does not include any other call statements. Edge (r_i, s_j) exists in this case. This edge is said to be associated with call site cs_1 at its start and with call site cs_2 at its end. The block of code $B(e)$ consists of basic blocks of procedure k which may be visited in any control flow path p from cs_1 to cs_2, such that the path p does not include any other call statements.

2. Procedure i invokes procedure j at call site cs and there is a path in CFG of i from the *start* node of procedure i to cs which does not include any other call statements. In this case, edge (s_i, s_j) exists. This edge is said to be associated with call site cs at its end. The block of code $B(e)$ consists of basic blocks of procedure i which may be visited in any control flow path p from start of i to cs, such that the path p does not include any other call statement.

3. Procedure j invokes procedure i at call site cs and there is a path in CFG of j from call site cs to a return statement within procedure j which does not include any other call statements. In this case, edge (r_i, r_j) exists. This edge is said to be associated with call site cs at its start. The block of code $B(e)$ consists of basic blocks of procedure j which may be visited in any control flow path p from cs to a return statement of j, such that the path p does not include any call statements.

4. In a procedure i, there is a possible flow of control from start node to a return statement, without any call statements. In this case, edge (s_i, r_i) exists. The block of code $B(e)$ consists of basic blocks of procedure i which may be visited in any control flow path p from start of i to a return statement in i, such that the path p does not include any call statements.

```
Program Foo
    a = 1
    Do i = 1, 100
        Call P(a,b)          ...cs1
        Call Q(c)            ...cs2
    Enddo
    Call Q(a)                ...cs3
    Call P(a,c)              ...cs4
    if cond then
        Call Q(a)            ...cs5
    Endif
    Call R(a,c)              ...cs6
End

Procedure P(x,y)
    Sched(x,y)
    ..other computations ..
End

Procedure Q(z)
    z = ...z...
End

Procedure R(y,z)
    Sched(y,z)
    ..other computations ..
End
```

Figure 2: An Example Program. A call site number is marked for each call site

An example program and its *FPR* are shown in Figures 2 and 3 respectively. In Figure 3, the blocks of codes $B(4)$, $B(9)$ and $B(11)$ comprise of all basic blocks in procedures P, Q and R respectively. Block of code corresponding to all other edges comprise of basic blocks from the main procedure. e.g. B(1) comprises of statement "a = 1" and the loop header, B(2) comprises of the end of the do loop and the loop header.

A block of code is a unit of placement in our analysis, i.e. we initially consider placement only at the beginning and end of a block of code[1]. Note that a basic block in a block of code may or may not be visited along a given control flow path from source to sink of the edge, and similarly, a basic block may belong to several blocks of code. This is taken into account during intraprocedural analysis done for determining final local placement, which we discuss in Section 5.5.

The availability of the following information is assumed during our interprocedural analysis phase. For each edge in *FPR*, we compute all the variables which are modified in the block of code corresponding to this edge. This information is used by the TRANS_e func-

[1] This is different from intraprocedural PRE in which placement is considered at beginning and end of node (basic block) of the graph.

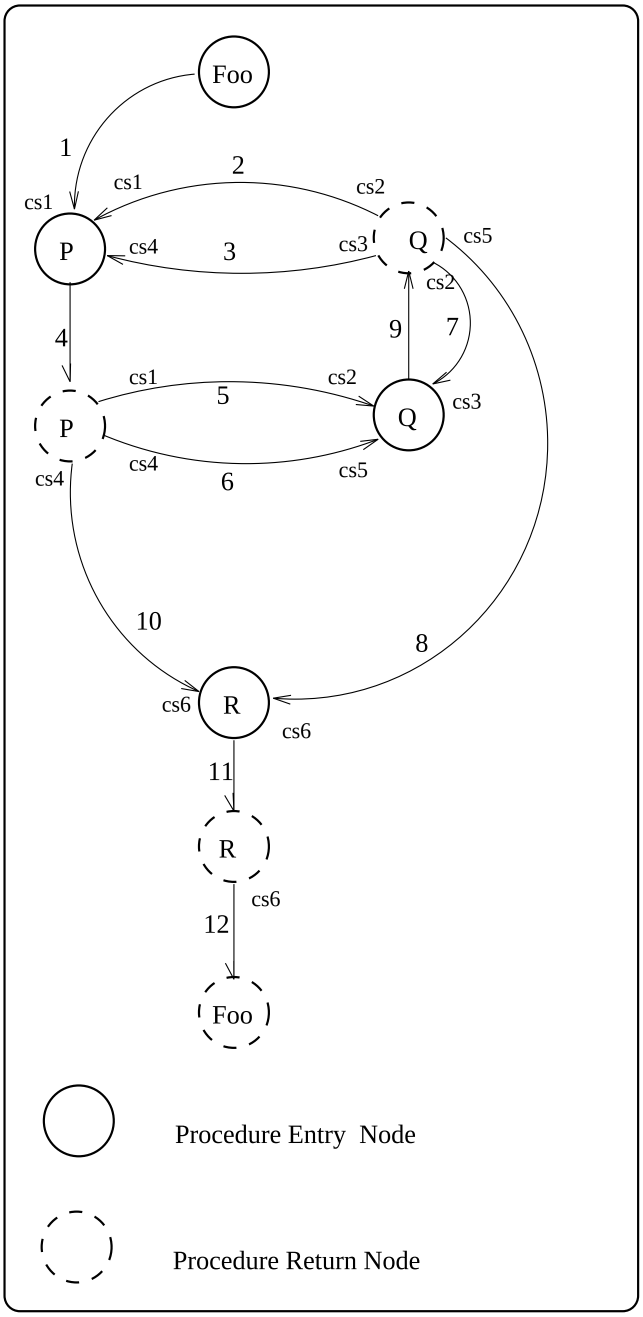

Figure 3: *FPR* for Program in Left. Edge numbers and call sites at which edges start/end (whenever applicable) are marked in the Figure.

tion defined later. For each procedure in the program, we also compute the list of variables modified by the procedure or any of the procedures invoked by this procedure. In absence of aliasing, this information can easily be computed by flow-insensitive interprocedural analysis in time linear to the size of call graph of the program [7]. This information is used by the CMOD_{cs} function defined later.

4.2 Candidates for Placement

We consider only the placement of pure functions. A pure function takes a number of parameters as input and produces a single result or output, without any side-effects or change in the value of inputs. In general, any subexpression can also be viewed as a pure function. In practice, one may want to focus on placement of only certain high cost functions, like communication statements and communication preprocessing statements in the case of distributed memory compilation.

A particular invocation of a pure function is considered for hoisting out of the procedure only if none of the parameters of the pure function is modified along any path from the start of the procedure to this invocation of the pure function and the invocation of pure function is not enclosed by any conditional or loop. (This can be generalized by considering slice of the pure function, but we do not discuss this possibility here). A particular invocation of a pure function is referred to as *candidate* if it is considered for interprocedural placement. We refer to the list of parameters of this pure function as the *list of influencers* of the candidate.

5 Interprocedural Partial Redundancy Elimination

We now present the IPRE scheme we have developed. We use the terms edge and the block of code corresponding to it interchangeably in this section.

Given the full program representation we described in Section 4, the major difficulties in applying data flow analysis for PRE are:

1. A procedure having a candidate for placement (or a procedure invoking such a procedure) may be invoked at multiple call sites with different sets of actual parameters, leading to different sets of influencers. (e.g. in the code shown in Figure 2, procedure P is invokes at two call sites with different parameters). While considering placement of the candidate outside the procedure it is originally placed, it must be ensured that only the computation of the candidate with correct set of influencers is visible during each invocation of the procedure.

2. For placement of a candidate at a certain point in a certain procedure, besides safety and profitability of the placement, it is also required that all influencers of the candidate are visible inside that procedure, i.e. each of them is either a global variable, a formal parameter or a local variable. (e.g. in the code shown in Figure 2, no placement will be possible inside procedure Q).

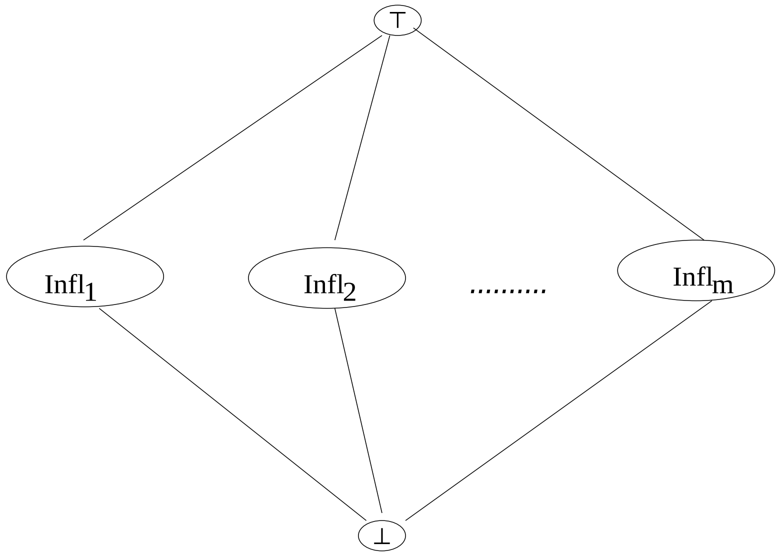

Figure 4: Lattice used in the data flow problems

3. If a procedure is invoked at several call sites in the program, our program representation shows paths from edges ending at a call site calling this procedure to the edges starting at other call sites for this procedure. (e.g. in Figure 3 there is a path from edge 6 to edge 9 to edge 2. Edge 6 ends at call site cs5 whereas edge 2 starts at call site cs2). These paths are never taken and the data flow analysis must not lose accuracy because of such paths in the graph.

4. Transparency of blocks of code cannot be determined before starting the solution of data flow equations, since it is not known what are the local variables which need to be unmodified for the propagation of data flow information.

5.1 Lattice for Data Flow Problems

We assume that the result of the computation of a candidate is always placed in a global store, i.e. it is not passed along as an actual parameter at the call sites. Consider a procedure p which has a candidate $\mathcal{C}$ for placement and is invoked at call sites cs_1 and cs_2 with different sets of parameters. Our scheme cannot place this candidate at a point from which there are paths leading to cs_1 and cs_2 and these paths do not have any further computation of $\mathcal{C}$. This restriction must be incorporated while propagating availability and while considering locations for possible placement (PPIN and PPOUT). If a candidate is available or if its placement is possible, it is always with a list of influencers, which will be used in placing the computation (i.e. if it is decided that the candidate is to be placed at this location).

For this purpose, we use a three-level lattice for the data flow problems. The lattice is shown in Figure 4. Each middle element in the lattice refers to a list of influencers, i.e. $\mathrm{Infl}_i = < v_1, v_2, \ldots, v_n >$. We define the following functions on this lattice: $\vee$ and $\wedge$ are standard binary join and meet operators. For ease in presenting our data flow equations, we use $\bigvee$ and $\bigwedge$ as confluence join and meet operators i.e. for computing join and meet respectively over a set of elements. $\neg$ is a unary operator which returns $\top$ when applied to a list of influencers and $\perp$ when applied to $\top$ or $\perp$. $\uplus$ is a binary non-commutative operators whose definition is as follows:

$$
\begin{aligned}
\top \uplus x &= \top \\
\mathrm{Infl}_i \uplus x &= \mathrm{Infl}_i \\
\perp \uplus x &= x
\end{aligned}
$$

5.2 Terminology

We further use the following terms to describe the data flow equations in this paper. We had defined our program represenation earlier in Section 4. In our Full Program Representation (FPR), the entry node corresponding to the main procedure is referred to as BEGIN node in the graph and similarly, the return node corresponding to the main is referred to as the END node in the graph.

The set of procedure return nodes is represented by $\mathcal{R}$ and the set of procedure entry nodes is represented by $\mathcal{E}$. Consider an edge $e = (v, w)$. The source node of e (i.e. the node v) is also referred to as $So(e)$ and the sink node of e (i.e. the node w) is also referred to as $Si(e)$. We use $pred(e)$ to refer to the set of edges whose sink node is v. We denote by $succ(e)$ the set of edges whose source node is w.

If the sink of the edge e is a procedure entry node, then the call site with which the edge e is associated at its end is denoted by $Si'(e)$. We use $succ'(e)$ to refer to the set of edges which are associated with the call site $Si'(e)$ at their start. Alternatively, if the source of the edge e is a procedure return node, then the call site with which the edge e is associated at the start is denoted by $So'(e)$. We refer by $pred'(e)$ the set of edges which are associated with the call site $So'(e)$ at their end.

Consider any edge e whose source is a procedure entry node. The set $cobeg(e)$ comprises of edges whose source is the same as the source of edge e. If an edge e has a procedure return node as the source and if cs is the call site with which the edge e is associated at its start, then the set $cobeg(e)$ comprises of the edges which are associated with the call site cs at their start.

Next, consider any edge e whose sink is a procedure return node. The set $coend(e)$ comprises of the edges whose sink is the same as the sink of the edge e. If an edge e has a procedure entry node as the source and if cs is the call site with which the edge e is associated at its end, then the set $coend(e)$ comprises of edges which are associated with the call site cs at their end.

The sets $pred(e)$, $pred'(e)$, $succ(e)$, $succ'(e)$, $cobeg(e)$ and $coend(e)$ for edges in the Graph shown in Figure 3 are shown in Figure 5.

At any call site cs, the set of actual parameters passed is ap_{cs} and the j^{th} actual parameter is $ap_{cs}(j)$. The set of formal parameters of the procedure invoked at the call site cs is fp_{cs}. (Clearly, this set is the same for all call sites which invoke this procedure). The j^{th} formal parameter is denoted by $fp_{cs}(j)$. The set of global variables in the program is gv.

5.3 Availability and Partial Availability

The equations for computing availability and partial availability are given in Figure 7. In computing availability, all unknowns are initialized with $\top$. This state means that the candidate may be available, but we do not yet know what will be the list of influencers if it is available. Bottom element in the lattice means that the candidate is not available.

Initially, the local data flow property ANTLOC(i) of the edges in the graph is determined. (For a block of code, ANTLOC means that there is a definition of this candidate inside the block.) In Section 4.2, we had discussed how procedures are marked with candidates for placement. Consider an edge i whose source is a procedure entry node s_p. If a candidate $\mathcal{C}$ is marked for placement from the procedure p with the list of influencers Infl_c, we set

$$\text{ANTLOC}_{\mathcal{C}}(i) \;=\; \text{Infl}_c$$

In all other cases, ANTLOC(i) is set to $\perp$.

The following functions are used in our data flow equations. $\text{TRANS}_e[\text{Infl}_i]$ of an edge e in the graph returns the list Infl_i if none of the influencers in the list Infl_i is modified in the block of code associated with this edge. If any of these influencers is modified, this function returns $\perp$. $\text{TRANS}_e[\top]$ and $\text{TRANS}_e[\perp]$ are defined to be $\top$ and $\perp$ respectively. For a call site cs which invokes procedure p, $\text{CMOD}_{cs}[\text{Infl}_i]$ returns the list Infl_i if none of the influencers in the list Infl_i is modified by the procedure p (or by a procedure invoked by p). Otherwise $\text{CMOD}_{cs}[\text{Infl}_i]$ returns $\perp$. $\text{CMOD}_{cs}[\top]$ always returns $\top$ and $\text{CMOD}_{cs}[\perp]$ always returns $\perp$. $\text{OCR}_{\mathcal{C}}(cs)$ determine if the procedure p (or any procedure invoked by p) includes any occurrence of the candidate $\mathcal{C}$. (Clearly, this will be the same for all call sites which call procedure p). Whenever there is no scope for ambiguity, we drop the subscript $\mathcal{C}$. OCR(cs) returns $\top$ or true when there is an occurrence of the candidate in the procedure p and $\perp$ (or false) when there is no occurrence of the candidate at procedure p.

For renaming of formal parameters at call sites, we define two functions RNM1_{cs} and RNM2_{cs} (see Figure 6). Suppose a candidate is available at a call site cs with a list of influencers Infl_i. The function RNM1_{cs} determines if this candidate can be available inside the procedure invoked at cs, and if so, with what list of influencers. If any of influencers is neither a global variable nor an actual parameters at cs, RNM1_{cs} returns $\perp$, otherwise, each actual parameter in the list is replaced by corresponding formal parameter. $\text{RNM1}_{cs}[\top]$ and $\text{RNM1}_{cs}[\perp]$ are defined to be $\top$ and $\perp$ respectively. Suppose a candidate is available at the return of a procedure and let cs be one of the call sites which invoke this procedure. RNM2_{cs} determines if this candidate will be available after the entry of the edges which start at call site cs. If any of the influencers of the candidate inside the procedure is neither a global variable, nor a formal parameter, then RNM2_{cs} returns $\perp$. Otherwise, each formal parameter is replaced by the actual parameter at call site cs.

The equations for propagation of availability can be explained as follows (see Figure 7). Consider an edge e whose source is a procedure entry node. A candidate will be available at the entry of this edge e if the following holds: This candidate should be available at the exit of any edge p which ends at this procedure entry node (i.e. $p \in pred(e)$), and furthermore, after renaming (i.e. applying $\text{RNM1}_{Si'(p)}$), the list of influencers with which the candidate is available should be the same for all such edges.

If an edge e has a procedure return node $So(e)$ as source, e is associated with call site $So'(e)$ at its start. The set $pred(e)$ comprises of edges whose sink is node $So(e)$ and the set $pred'(e)$ comprises of edges which are associated with the call site $So'(e)$ at their end. Note that even if the candidate is available at the end of all the edges p' ($p' \in pred'(e)$) and none of the influencers is modified inside the procedure, the candidate may not be available inside the procedure. This can happen for two reasons, all influencers of the candidate may not be visible inside the procedure, or the procedure may be invoked at multiple call sites and the candidate may not be available at other call sites.

If there is no definition of the candidate in the procedure ($\text{CMOD}_{So'(e)}$ does not return $\perp$), then AVIN(e) is determined by AVOUT at the edges belonging to $pred'(e)$. If there is any definition of the candidate in the procedure, then AVIN(e) is determined by AVOUT at the edges belonging to $pred(e)$. Note that this step preserves calling context of the procedure, i.e. accuracy in data flow analysis is not lost if a procedure is invoked at multiple call sites.

Equation 4 determines availability of a candidate at the end of an edge or block of code. If there is a computation of the candidate in the edge with list of influencers Infl_i, then AVOUT is Infl_i if none of the influencers is modified along this edge. If there is no computation inside the edge (i.e. ANTLOC is $\perp$), then the computation is available at the exit of the edge only if it is available at the entry of the edge and if none of the influencers is modified along the edge.

In computing partial availability, all unknowns are

e	$pred(e)$	$pred'(e)$	$succ(e)$	$succ'(e)$	$cobeg(e)$	$coend(e)$
1	-	-	4	5	1	1,2
2	9	5	4	5	2,7	1,2
3	9	7	4	6,10	3	3
4	1,2,3	-	5,6,10	-	4	4
5	4	1,2	9	2,7	5	5
6	4	3	9	8	6,10	6
7	9	5	9	3	2,7	7
8	9	6	11	12	8	8,10
9	5,6,7	-	2,3,7	-	9	9
10	4	3	11	12	6,10	8,10
11	8,10	-	12	-	11	11
12	11	8,10	-	-	12	12

Figure 5: $pred(e)$, $pred'(e)$, $succ(e)$ and $succ'(e)$ sets for Graph in Figure 3

$$\mathrm{T1}(v_i) = \begin{cases} v_i & \text{if } v_i \in gv \\ fp_{cs}(j) & \text{if } v_i = ap_{cs}(j) \end{cases}$$

$$\mathrm{RNM1}_{cs}[< v_1, \ldots, v_n >] = \begin{cases} \perp & \text{if } \exists i,\ (v_i \notin gv) \wedge (\forall j\ , v_i \neq ap_{cs}(j)) \\ < \mathrm{T1}(v_1), \ldots, \mathrm{T1}(v_n) > & \text{otherwise} \end{cases} \quad (1)$$

$$\mathrm{T2}(v_i) = \begin{cases} v_i & \text{if } v_i \in gv \\ ap_{cs}(j) & \text{if } v_i = fp_{cs}(j) \end{cases}$$

$$\mathrm{RNM2}_{cs}[< v_1, \ldots, v_n >] = \begin{cases} \perp & \text{if } \exists i,\ (v_i \notin gv) \wedge (\forall j\ , v_i \neq fp_{cs}(j)) \\ < \mathrm{T2}(v_1), \ldots, \mathrm{T2}(v_n) > & \text{otherwise} \end{cases} \quad (2)$$

Figure 6: Renaming functions

initialized with $\perp$. The equations for computing partial availability (Equations 5 and 6) are very similar to corresponding equations for computing availability, except that join operator is used instead of meet operator.

The role of partial availability is to suggest profitability of transformations, it does not effect correctness and safety of transformations. Partial availability does not always guarantee that redundant code motion will not occur. We have therefore, used a simple method for determining partial availability, which may not always be accurate. Inaccuracy comes in for two reasons. CMOD and TRANS_e functions return $\perp$ whenever one of the influencers is modified in one of the basic blocks, this basic block may not occur in all the control flow paths taken. Secondly, calling context is not always preserved in propagating partial availability information. Precise computation of partial availability can be expensive, it will require a detailed representation like Myer's SuperGraph [22] and use of stacks and/or graph reachability for preserving calling context [23]. Our computation of partial availability still allows loop invariant code motion and redundant computation elimination. Some other optimizations which can be obtained by suppression of partially redundant computations may not be achieved because of this simple solution.

5.4 Data Flow Analysis for Placement

The data flow equations for determining placement of computations are shown in Figure 8. We briefly explain some of the key terms in these equations.

In computing PPIN in the intraprocedural case, the product term PPOUT + AVOUT ensures availability of the candidate at the entry of the node in the optimized program. PPOUT means that the candidate will be available as a result of the placements determined by the scheme. AVOUT means that the candidate is available in the original program. In the interprocedural case, the same candidate can be available with more than one list of influencers. In computing PPIN in the interprocedural scheme, we use the term PPOUT $\uplus$ AVOUT (Equation 8). If PPOUT is set to a list of influencers Infl_i then, after the placement determined by the scheme, the candidate will be available with set of influencers Infl_i, even if it is available with a different list of influencers before the optimized placements. If PPOUT is $\perp$ and AVOUT is Infl_j, then the candidate will be available with the same set of influencers Infl_j even after the placement.

The rational behind the equation for determining INSERT is as follows. We decide to insert a candidate with the set of influencers Infl_i at the end of a block of code e, if $\mathrm{PPOUT}(e)$ is Infl_i, $\mathrm{AVOUT}(e)$ is not Infl_i and either $\mathrm{PPIN}(e)$ is not Infl_i or one of the influencers in the list Infl_i is modified in this block of code. The term

$$\mathrm{AVIN}(e) = \begin{cases} \perp & \text{if } So(e) \text{ is BEGIN node} \\ \bigwedge_{p \in pred(e)} (\,\mathrm{RNM1}_{Si'(p)}[\,\mathrm{AVOUT}(p)\,]\,) & \text{if } So(e) \in \mathcal{E} \\ \mathrm{CMOD}_{So'(e)}[\bigwedge_{p' \in pred'(e)} \mathrm{AVOUT}(p')\,] & \text{if } (So(e) \in \mathcal{R}) \wedge (\neg\, \mathrm{OCR}(So'(e))) \\ \mathrm{RNM2}_{So'(e)}[\bigwedge_{p \in pred(e)} \mathrm{AVOUT}(p)\,] & \text{if } (So(e) \in \mathcal{R}) \wedge (\,\mathrm{OCR}\,(So'(e))) \end{cases} \tag{3}$$

$$\mathrm{AVOUT}(e) = \bigwedge_{c \in coend(e)} (\,\mathrm{TRANS}_c[\mathrm{ANTLOC}(c) \uplus \mathrm{AVIN}(c)]\,) \tag{4}$$

$$\mathrm{PAVIN}(e) = \begin{cases} \perp & \text{if } So(e) \text{ is BEGIN node} \\ \bigvee_{p \in pred(e)} (\,\mathrm{RNM1}_{Si'(p)}[\,\mathrm{PAVOUT}(p)\,]\,) & \text{if } So(e) \in \mathcal{E} \\ \mathrm{CMOD}_{So'(e)}[\bigvee_{p' \in pred'(e)} \mathrm{PAVOUT}(p')\,] & \text{if } (So(e) \in \mathcal{R}) \wedge (\neg\, \mathrm{OCR}(So'(e))) \\ \mathrm{RNM2}_{So'(e)}[\bigvee_{p \in pred(e)} \mathrm{PAVOUT}(p)\,] & \text{if } (So(e) \in \mathcal{R}) \wedge (\,\mathrm{OCR}\,(So'(e))) \end{cases} \tag{5}$$

$$\mathrm{PAVOUT}(e) = \mathrm{TRANS}_e[\mathrm{ANTLOC}(e) \uplus \mathrm{PAVIN}(e)] \tag{6}$$

Figure 7: Data Flow Equations for Availability and Partial Availability

$$\mathrm{PPOUT}(e) = \begin{cases} \perp & \text{if } Si(e) \text{ is END node} \\ \bigwedge_{s \in succ(e)} (\,\mathrm{RNM2}_{So'(s)}[\,\mathrm{PPIN}(s)\,]\,) & \text{if } Si(e) \in \mathcal{R} \\ \mathrm{CMOD}_{Si'(e)}[\bigwedge_{s' \in succ'(e)} \mathrm{PPIN}(s')\,] & \text{if } (Si(e) \in \mathcal{E}) \wedge (\neg \mathrm{OCR}(Si'(e))) \\ \mathrm{RNM1}_{Si'(e)}[\bigwedge_{s \in succ(e)} \mathrm{PPIN}(s)\,] & \text{if } (Si(e) \in \mathcal{E}) \wedge (\,\mathrm{OCR}(Si'(e))) \end{cases} \tag{7}$$

$$\mathrm{TEMP1}(i) = \bigwedge_{c \in cobeg(e)} (\,\mathrm{ANTLOC}(c) \uplus \mathrm{TRANS}_c[\mathrm{PPOUT}(c)]\,)$$

$$\mathrm{TEMP2}(i) = \mathrm{PPOUT}(i) \uplus \mathrm{AVOUT}(i)$$

$$\mathrm{PPIN}(e) = \begin{cases} \perp & \text{if } So(e) \text{ is BEGIN node} \\ \mathrm{PAVIN}(e) \wedge \mathrm{TEMP1}(e) \wedge & \\ \quad \bigwedge_{p \in pred(e)} (\,\mathrm{RNM1}_{Si'(p)}[\,\mathrm{TEMP2}(p)\,]\,) & \text{if } So(e) \in \mathcal{E} \\ \mathrm{PAVIN}(e) \wedge \mathrm{TEMP1}(e) \wedge & \\ \quad (\,(\mathrm{CMOD}_{So'(e)}[\bigwedge_{p' \in pred'(e)} \mathrm{TEMP2}(p')\,]) & \text{if } (So(e) \in \mathcal{R}) \wedge (\neg\, \mathrm{OCR}(So'(e))) \\ \mathrm{PAVIN}(e) \wedge \mathrm{TEMP1}(e) \wedge & \\ \quad (\,\mathrm{RNM2}_{So'(e)}[\bigwedge_{p \in pred(e)} \mathrm{TEMP2}(p)\,])\,) & \text{if } (So(e) \in \mathcal{R}) \wedge (\,\mathrm{OCR}\,(So'(e))) \end{cases} \tag{8}$$

$$\begin{aligned} \mathrm{INSERT}(e) = {} & \mathrm{PPOUT}(e) \wedge \neg(\mathrm{PPOUT}(e) \wedge \mathrm{AVOUT}(e)) \wedge \\ & (\neg(\mathrm{PPIN}(e) \wedge \mathrm{PPOUT}(e)) \vee \neg \mathrm{TRANS}_e[\mathrm{PPOUT}(e)]) \end{aligned} \tag{9}$$

$$\mathrm{DEL}(e) = \mathrm{ANTLOC}(e) \wedge \mathrm{PPIN}(e) \tag{10}$$

Figure 8: Data Flow Equations for Placement

$\neg\,(\,\mathrm{PPOUT}(e) \wedge \mathrm{AVOUT}(e)\,)$ will return $\top$ whenever $\mathrm{PPOUT}(e)$ and $\mathrm{AVOUT}(e)$ are not set to the same list of influencers Infl_i.

In determining placement (PPIN and PPOUT), we preserve the calling context of the procedures by using a simple method, the same that we used for computing availability. It can be shown that the safety of placement is maintained through this method.

Lemma 1 *Consider any procedure p such that the procedure p or any of the procedures invoked by it do not have any occurrence of the candidate $\mathcal{C}$. Let cs be one of the call sites which call procedure p. Let* PPIN *be* Infl_i *for any edge starting at call site cs and, further, let there be no modification to any of the influencers in the call to procedure p. Then, no placement of the candidate will be done in any block of code inside call to procedure p.*

The initial value of the PPIN and PPOUT are set to $\top$. The desired solution is the largest solution and can be found by iterative method.

5.5 Final Local Placement

We have so far considered the block of code associated with a single edge of *FPR* as the unit of placement. The final placement of the candidates which have to be inserted depends upon further intraprocedural analysis and is not necessarily at the end of blocks of code.

Consider an edge e for which INSERT(e) is true. A number of edges end the same procedure return node or the same call site as the edge e and INSERT may not be true for all of them. Since all these edges have the same $succ(e)$ and $succ'(e)$ sets, they have the same value of PPOUT(e) and AVOUT(e). Therefore, the difference in the value of INSERT(e) comes because of the difference in the value of PPIN(e) or TRANS_e[PPOUT(e)]. For determining the final placement, the control flow graph is traversed backwards from the call site cs or the procedure return statement. Along any such traversal path, we identify the first basic block which belongs to the blocks of code for which INSERT is true but does not belong to the blocks of code for which INSERT is not true. Let b_1 be such a basic block and let b_2 be its successor which belongs to the block of code for which INSERT is true. A new basic block is inserted between the basic blocks b_1 and b_2 and the candidate is inserted in this new basic block. It can be shown that the following property is maintained by this scheme.

Lemma 2 *No insertion is made in any block of code for which* INSERT *is* $\perp$.

Using the above two lemmas, the correctness and safety properties of the interprocedural scheme can be established in the same way as the correctness and safety of the original intraprocedural scheme [20].

Theorem 1 *(Correctness) After insertion of new computations, the computation of the candidate $\mathcal{C}$ becomes redundant in an edge satisfying* $\mathrm{ANTLOC}_{\mathcal{C}} = \mathrm{Infl}_i$ *and* $\mathrm{PPIN}_{\mathcal{C}} = \mathrm{Infl}_i$.

Theorem 2 *(Safety) Consider any edge of FPR in which a new computation $\mathcal{C}$ is inserted. Every path starting from sink of this edge includes a computation which will be deleted, before including any edge in which a new occurrence of this computation will be added.*

The solution of data flow properties for the program shown in Figure 2 is shown in Figure 9. The optimized program is shown in Figure 10.

6 Discussion

6.1 Effectiveness of the Scheme

We have implemented a preliminary version of our scheme using the existing Fortran D compilation system developed at Rice University [17] as the necessary infrastructure. We studied the effectiveness of our scheme in compiling an Euler solver for unstructured grids [8], a code which accesses data through indirection arrays in parallel loops. The existing compiler for irregular applications [9, 14] generated calls to PARTI routines for communication preprocessing and collective communication [24], but did not perform any interprocedural placement of these statements.

The performance achieved by the compiled code (before interprocedural optimizations) and the code after interprocedural optimizations is presented in Figure 11. The experimental results show that interprocedural placement of communication preprocessing statements is a must for obtaining reasonable performance. When the program is run on a small number of processors, the communication cost is small and therefore, the performance difference made by interprocedural placement of communication statements is small. However, when the same data is distributed over a large number of processors, the communication time becomes a significant part of the total execution time of the program. Then, performing interprocedural placement of communication statements also makes a substantial difference on the total execution time of the program.

6.2 Cost of the Scheme

There are two issues in evaluating the cost of our scheme: the number of edges in the graph constructed, and the number of iterations required for data flow equations to be solved. In the worst case, each procedure may contribute edges quadratic in the number of statements in the program. In practice, we expect this to be much smaller than the number of basic blocks in full program, e.g. the Euler code we experimented with had 9 procedures, and a total of 1400 lines of code. The resulting graph had 16 edges, whereas the total number of basic blocks in the program was 117. In future, we plan to do this measurement for a number of different codes.

The number of iterations required for data flow equations to converge is, in the worst case, proportional to the number of edges in the graph. If the number of egdes in the graph is small, the time required for solution will be small.

7 Related Work

We are aware of two efforts on performing interprocedural partial redundancy elimination. Morel and Renvoise briefly discuss how their scheme can be extended interprocedurally [21]. Their solution is hueristic in nature, and no formal details are available for their interprocedural scheme. Their work is restricted to the programs whose call graph is acyclic. They also do not consider the possibility that the procedure having a candidate for placement may be invoked at multiple call sites with different set of parameters and do not maintain accuracy of solutions when procedures are invoked at multiple call sites.

Knoop *et al.* extend a scheme for performing earliest possible code motion interprocedurally [18]. The main limitation of their work is that if any of the influencers of a candidate is a formal parameter, then the candidate is not considered for placement outside procedure boundary (since no renaming of influencers is done). In the example presented in this paper, as well as in the Euler code we used for our experiments, their scheme will not perform any code motion. They do not consider the possibility of using any compact representation for the full program. Also, we believe that our effort is the first one to report an implementation and application of interprocedural partial redundancy elimination. In our earlier work, we had outlined using interprocedural partial redundancy elimination for dis-

Edge	AVIN	AVOUT	PAVIN	PAVOUT	PPOUT	PPIN	DEL	INSERT
1	$\perp$	$\perp$	$\perp$	$\perp$	$< a, b >$	$\perp$	$\perp$	$< a, b >$
2	$< a, b >$	$\perp$	$< a, b >$	$< a, b >$	$< a, b >$	$< a, b >$	$\perp$	$\perp$
3	$\perp$	$\perp$	$\perp$	$\perp$	$< a, c >$	$\perp$	$\perp$	$< a, c >$
4	$\perp$	$< x, y >$	$< x, y >$	$< x, y >$	$< x, y >$	$< x, y >$	$< x, y >$	$\perp$
5	$< a, b >$	$< a, b >$	$< a, b >$	$< a, b >$	$< a, b >$	$< a, b >$	$\perp$	$\perp$
6	$< a, c >$	$< a, c >$	$< a, c >$	$< a, c >$	$\perp$	$< a, c >$	$\perp$	$\perp$
7	$< a, b >$	$< a, b >$	$< a, b >$	$< a, b >$	$< a, b >$	$< a, b >$	$\perp$	$\perp$
8	$\perp$	$\perp$	$\perp$	$\perp$	$< a, c >$	$\perp$	$\perp$	$< a, c >$
9	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
10	$< a, c >$	$\perp$	$< a, c >$	$< a, c >$	$< a, c >$	$< a, c >$	$\perp$	$\perp$
11	$\perp$	$< y, z >$	$< y, z >$	$< y, z >$	$< y, z >$	$< y, z >$	$< y, z >$	$\perp$
12	$< a, c >$	$< a, c >$	$< a, c >$	$< a, c >$	$\perp$	$< a, c >$	$\perp$	$\perp$

Figure 9: Solution of Data Flow Properties for the Graph

```
Program Foo                          Procedure P(x,y)
    a = 1                                ..other computations ..
    Sched(a,b)                       End
    Do i = 1, 100
        Call P(a,b)
        Call Q(c)                    Procedure Q(z)
    Enddo                                z = ...z...
    Call Q(a)                        End
    Sched(a,c)
    Call P(a,c)
    If cond then
        Call Q(c)                    Procedure R(y,z)
        Sched(a,c)                       ..other computations ..
    Endif                            End
    Call R(a,c)
End
```

Figure 10: Optimized Version of Program. Note that further Intraprocedural Analysis is required at call sites cs1 and cs6 to determine final placement

tributed memory compilation [1], but no formal details of the scheme or empirical evaluation was presented.

We compare our work with efforts on other flow-sensitive interprocedural problems. Several different program representations have been used for different flow-sensitive interprocedural problems. Myer has suggested concept of SuperGraph [22] which is constructed by linking control flow graphs of subroutines by inserting edges from call site in the caller to start node in callee. The total number of nodes in SuperGraph can get very large and consequently the solution may take much longer time to converge. Several ideas in the design of our representation are similar to the ideas used in Callahan's Program Summary Graph [6] and Interprocedural Flow Graph used by Soffa et al. [16]. FIAT has been introduced as a general framework for performing interprocedural analysis [13], but is more targeted towards flow-insensitive problems. Interval based approach for solving interprocedural data flow equations has been investigated in [4]. Recompilation in a compiler performing interprocedural analysis has been investigated in [5].

8 Conclusions

In this paper we have addressed the problem of performing partial redundancy elimination interprocedurally. This problem was initially motivated by the problem of placement of communication preprocessing statements in distributed memory compilation. We have developed an interprocedural partial redundancy elimination (IPRE). Our algorithm is applicable on arbitrary recursive programs.

Acknowledgements

We thank Bill Pugh for critically reading earlier versions of this paper and suggesting several improvements to the scheme and its presentation. We have implemented this scheme as a part of the D system being developed under the leadership of Ken Kennedy at Rice university. We gratefully acknowledge our debt to the implementers of the interprocedural infrastructure (FIAT) and the existing Fortran D compiler.

References

[1] Gagan Agrawal and Joel Saltz. Interprocedural com-

Euler Solver on 10K mesh: 20 iterations

No. of Procs.	No IP opt.	IP opt. preproc. stmts.	IP opt. comm. stmt.
2	47.74	30.26	29.75
4	26.77	14.99	14.27
8	17.45	9.85	8.51
16	12.35	7.41	5.56
32	12.72	8.92	5.09

Figure 11: Effectiveness of Interprocedural placement schemes. All numbers are in Seconds, on Intel Paragon

munication optimizations for distributed memory compilation. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, pages 283–299, August 1994. Also available as University of Maryland Technical Report CS-TR-3264.

[2] Gagan Agrawal, Alan Sussman, and Joel Saltz. Compiler and runtime support for structured and block structured applications. In *Proceedings Supercomputing '93*, pages 578–587. IEEE Computer Society Press, November 1993.

[3] Gagan Agrawal, Alan Sussman, and Joel Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 1994. To appear. Also available as University of Maryland Technical Report CS-TR-3143 and UMIACS-TR-93-94.

[4] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[5] Michael Burke and Linda Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[6] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[7] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[8] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives. *AIAA Journal*, 32(3):489–496, March 1994.

[9] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 152–168. Springer-Verlag, August 1993. Also available as University of Maryland Technical Report CS-TR-3076 and UMIACS-TR-93-42.

[10] D.M. Dhamdhere and H. Patil. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM Transactions on Programming Languages and Systems*, 15(2):312–336, April 1993.

[11] K. Drechsler and M. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[12] Manish Gupta, Edith Schonberg, and Harini Srinivasan. A unified data flow framework for optimizing communication. In *Proceedings of Languages and Compilers for Parallel Computing*, August 1994.

[13] Mary Hall, John M Mellor Crummey, Alan Carle, and Rene G Rodriguez. FIAT: A framework for interprocedural analysis and transformations. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 522–545. Springer-Verlag, August 1993.

[14] Reinhard v. Hanxleden. Handling irregular problems with Fortran D - a preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. Also available as CRPC Technical Report CRPC-TR93339-S.

[15] Reinhard von Hanxleden and Ken Kennedy. Give-n-take – a balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 107–120. ACM Press, June 1994. ACM SIGPLAN Notices, Vol. 29, No. 6.

[16] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[17] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[18] J. Knoop and Steffan B. Efficient interprocedural bit-vector data flow analyses: A uniform interprocedural framework. Technical report, Dept. of Computer Science, University of Kiel, September 1993.

[19] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[20] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[21] E. Morel and C. Renvoise. Interprocedural elimination of partial redundancies. In *Program Flow Analysis: Theory and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[22] E. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth ACM Symposium on the Principles of Programming Languages*, pages 219–230, January 1981.

[23] Thomas Reps, Susan Horowitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Fourteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1995.

[24] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[25] A. Sorkin. Some comments on "A solution to a problem with Morel and Renvoise's 'Global optimization by suppression of partial redundancies'". *ACM Transactions on Programming Languages and Systems*, 11(4):666–668, October 1989.

Appendix

A.1 Intraprocedural Partial Redundancy Elimination

The first partial redundancy elimination scheme was presented by Morel and Renvoise [20]. This scheme has been further extended and refined by Dhamdhere [10], Drechsler [11], Knoop et. al. [19] and Sorkin [25]. The details of interprocedural redundancy elimination we present are derived from the intraprocedural node based method of Dhamdhere [10], also referred to as Modified Morel Renvoise Algorithm (MMRA). The data flow equations used in the scheme are presented in Figure 12. The terms used in the data flow equations are explained below.

Local data flow properties:
ANTLOC(i): Node i contains an occurrence of computation $\mathcal{C}$ not preceded by a definition of any of its operands.
COMP(i): Node i contains an occurrence of computation $\mathcal{C}$ not followed by a definition of any of its operands.
TRANS(i): Node i does not contain a definition of any operand of computation $\mathcal{C}$.

Global data flow properties:
AVIN(i)/AVOUT(i): Computation $\mathcal{C}$ is available at the entry/exit of node i.
PAVIN(i)/PAVOUT(i): Computation $\mathcal{C}$ is partially available at the entry/exit of node i.
PPIN(i)/PPOUT(i): Computation of $\mathcal{C}$ may be placed at entry/exit of node i.

$$\mathrm{AVIN}(i) = \begin{cases} \text{false} & \text{if } i \text{ is entry block} \\ \prod_{p \in pred(i)} \mathrm{AVOUT}(p) & \text{otherwise} \end{cases}$$

$$\mathrm{AVOUT}(i) = \mathrm{COMP}(i) + \mathrm{TRANS}(i).\mathrm{AVIN}(i))$$

$$\mathrm{PAVIN}(i) = \begin{cases} \text{false} & \text{if } i \text{ is entry block} \\ \sum_{p \in pred(i)} \mathrm{PAVOUT}(p) & \text{otherwise} \end{cases}$$

$$\mathrm{PAVOUT}(i) = \mathrm{COMP}(i) + \mathrm{TRANS}(i).\mathrm{PAVIN}(i)$$

$$\begin{aligned} \mathrm{PPIN}(i) = {} & \mathrm{PAVIN}(i).\,(\,\mathrm{ANTLOC}(i) + \\ & \mathrm{TRANS}(i)\,.\,\mathrm{PPOUT}(i)) \\ & \textstyle\prod_{p \in pred(i)} (\mathrm{PPOUT}(p) + \mathrm{AVOUT}(p)). \end{aligned}$$

$$\mathrm{PPOUT}(i) = \begin{cases} \text{false} & \text{if } i \text{ is exit block} \\ \prod_{s \in succ(i)} \mathrm{PPIN}(s)) & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathrm{INSERT}(i) = {} & \mathrm{PPOUT}(i)\,.\,\neg\mathrm{AVOUT}(i). \\ & (\neg\mathrm{PPIN}(i) + \neg\mathrm{TRANS}(i)) \end{aligned}$$

$$\mathrm{DEL}(i) = \mathrm{ANTLOC}(i).\mathrm{PPIN}(i)$$

Figure 12: MMRA scheme for Intraprocedural Partial Redundancy Elimination

DEL(i): Occurrence of $\mathcal{C}$ in node i is redundant
INSERT(i): A computation of $\mathcal{C}$ should be placed at the exit of node i.

We now briefly explain the rational behind the key equations. A computation is available at the entry of a basic block if it is available at the exit of all the predecessor basic blocks. A computation is available at the end of a basic block if it is available at the beginning of the basic block and none of the operands are modified in the basic block, or, alternatively, there is an occurrence of this computation in this basic block, not followed by any definition of the operands. A computation is partially available at the entry of a basic block if it is partially available at the exit of at least one predecessor block. The equations for placement can be explained as follows. In computing PPOUT, the $\prod$ term ensures safety in placing an expression at the exit of the node. The $\prod$ term in computing PPIN ensures availability of the expression at the entry of this node in the optimized program. The term PAVIN determines the profitability of hoisting a computation out of this node. This term avoids redundant code hoisting for almost all cases for any real program, however, it does not guarantee. In the original MMRA scheme [10], an additional term is used to further prevent redundant code hoisting, this term still does not guarantee that no redundant code hoisting occurs. For simplicity, we do not include this additional term in our presentation.