SCCS 698

A Framework for Representing Data Parallel Programs and its Application in Program Reordering

by

Rajesh Bordawekar and Alok Choudhary

4 May 1995

Northeast Parallel Architectures Center
at Syracuse University
Science and Technology Center
111 College Place
Syracuse, NY 13244-4100

Contents

SCCS 698

A Framework for Representing Data Parallel Programs and its Application in Program Reordering

Rajesh Bordawekar and Alok Choudhary

4 May 1995

Northeast Parallel Architectures Center
at Syracuse University
Science and Technology Center
111 College Place
Syracuse, NY 13244-4100

`rajesh@npac.syr.edu`
`http://www.npac.syr.edu/users/rajesh/homepage/rajesh.html`

Abstract

In this paper, we present a framework for describing the dataflow and dependence information of a data parallel program. Our framework initially represents the program using a directed Intermediate Program Locality Graph (IPLG). Using Dependence Access Relations (DARs), the IPLG is reduced to a Program Locality Graph (PLG).

The information provided by PLG is used by the compiler to reorder the program to improve program locality. We view the program reordering problem as an optimization problem. To solve this problem, we present a polynomial time heuristic, called the *Range Reduction Heuristic*. The best case time complexity of the heuristic is $O(m^2n^2)$, where m is the number of statements and n is the number of arrays used in the program. The average case running time of the heuristic approaches the best case performance.

This framework will be implemented as a part of the PASSION (**P**arallel **A**nd **S**calable **S**oftware for **I/O**) compiler to compile out-of-core HPF programs. The PASSION compiler will take advantage of the improved program locality and use the dependence/dataflow information to perform further optimizations like (1) Providing Hints to the file system, (2) Data Retaining Policies, (3) Dead Code Elimination and (4) Local File Reorganization.

1 Introduction

The performance of a program often depends upon the time required to access data from the memory, either primary or secondory memory. Large scale multiprocessors use hierarchical memories to reduce large access times. In multi-level memories, the access cost depends on the memory level. Generally registers have the smallest access cost while the secondary storage mediums like disks have a very large access cost.
The advent of high-performance computing has allowed researchers to solve large computational applications. These applications, in addition to requiring a great deal of computational power, also deal with large quantities of data. Due to a relatively small size of main memory, data often needs of be stored on disks. Hence, the overall performance of these applications depends on I/O performance of the program.

The I/O performance of a program can be improved by reordering the program so that variables are accessed while they reside in lower levels of the memory hierarchy, as far as possible. In order to perform *legal* program ordering, it is necessary to understand the dataflow and dependence characteristics of the given program. In this paper, we provide an unified framework for representing dataflow and dependence information of data parallel programs and use it for reordering program statements.

1.1 Contributions of the paper

In this paper, we focus on out-of-core HPF programs. We describe a framework for representing dataflow and dependence information of HPF programs. The dataflow and dependence information is used to reorder array assignment statements so that two statements which access the common array variables are closer in program space. Though this work is targeted towards out-of-core HPF programs, it can be easily applied to in-core HPF, F90 or any other data parallel programs. Our basic approach is similar to that taken by [GS90, GV91] but we view the locality improvement problem as an optimization problem. We show that the global cost can be minimized (corresponding to the overall program optimization) if each step of the problem optimizes the local I/O cost (i.e. improving the locality of a pair of statements). Our framework represents an HPF program using a Program Locality Graph (PLG). In order to have a concise representation of dataflow and dependence properties of the program, we use *Dependence Access Relations* (DAR). Based on our framework, we present a locality optimization heuristic called the *Range Reduction Heuristic.* The running time of this heuristic is bounded by the number of arrays and the number of statements used in the program. The worst case running time of the range reduction heuristic is $O(m^3n^2)$, where m is the number of statements and n is the number of arrays used in the program. The best case running time of the heuristic is $O(m^2n^2)$. In case of programs involving significant data dependencies, the running time approaches the best case running time.
The dataflow information provided by the framework will be used by the HPF compiler to perform various optimizations. These optimizations include

1. Providing caching hints to file systems.
2. Data Retaining Policies.
3. Dead Code Elimination.
4. Reorganizing data in local array files [TBC94] according to the statement ordering in the source program.

1.2 Related Work

Dataflow analysis has been used for by several researchers for program analysis and restructuring: especially for dead-code elimination and code placement. Morel and Renvoise first developed a dataflow framework for performing partial redundancy elimination [MR79] and since then has been refined and used by several researchers for a wide variety of problems [JD82a, JD82b, Dha88a, Dha88b]. For example, Dhamdhere used partial redundancy elimination for placing register load and stores [Dha90] while Carr and Kennedy have used it with dependence analysis to perform scalar replacement [CK94]. Dataflow analysis has also been used to optimizing communication. Amarasinghe and Lam used *Last Write Trees* to optimize communication for regular array accesses [AL93]. Hanxleden *et al.* presented a dataflow framework, called GIVE-N-TAKE, to generate communication statements for the Rice FORTRAN-D compiler [vKK+92, vK93]. The GIVE-N-TAKE framwork has been since extended to perform advanced communication optimizations like amalgamation and vectorization [KN94]. Gupta *et al.* applied partial redundancy elimination techniques on *available section descriptors* to determine availability of data on virtual processors [GSS94]. Ferrante *et al.* described how to compute array section data called, *Communication Sets*, using dataflow analysis [FGS94]. Other problems where dataflow analysis has been used include analysis of array sections by Gross and Steenkiste [GS90] and detection of array accesses in parallel programs by Granston and Veidenbaum [GV91].

1.3 Organization of the paper

The rest of the paper is organized as follows. Section 2 describes the out-of-core compilation strategy used by the PASSION compiler and presents an example of out-of-core HPF program. Section 3 explains in detail how an HPF program is represented as a Program Locality Graph. Section 4 introduces the concept of *range* and proposes two locality cost models. The *Range Reduction* Algorithm is presented and analyzed in section 5. Section 6 illustrates various compiler optimizations in which the information obtained from dataflow analysis is used. Finally, we conclude in Section 7.

2 Motivation

2.1 Out-of-core Compilation

In out-of-core programs primary data structures reside on disks. We call this data out-of-core or (OOC) data. Computations on OOC data, therefore, require staging data in smaller granules that can fit in the main memory of a system (*in-core data*). That is, the computation is carried out in several phases, where, in each phase part of the data is brought into memory, processed, and stored back onto secondary storage (if necessary). The staging of data can be done either by user-controlled explicit I/O or by system-controlled paging. In both cases, computation on *in-core* data requires secondary memory accesses resulting in I/O.
In this work, we focus on out-of-core computations performed on distributed memory machines. In distributed memory computations, work distribution is often obtained by distributing data over processors. For example, High Performance Fortran (HPF) provides explicit directives (`TEMPLATE, ALIGN` and `DISTRIBUTE`) which describe how the arrays should be partitioned over processors [For93, KLS+94]. Arrays are first aligned to a template (provided by the `TEMPLATE` directive). The `DISTRIBUTE` directive specifies how the template should be distributed among the processors. In HPF, an array can be distributed as either `BLOCK`(m) or `CYCLIC`(m). In a `BLOCK`(m) distribution, contiguous blocks of size m are distributed among the processors. In a `CYCLIC`(m) distribution, blocks of size m are distributed cyclically. The `DISTRIBUTE` directive specifies which elements of the global array should be mapped to each processor. This results in each processor having a *local array* associated with it. Our main assumption is that local arrays are stored in files from which the data is staged into main memory. When the global array is an out-of-core array, the corresponding local array is also stored in files. The out-of-core local array can be stored in files using two distinct data placement models. The first model, called the *Global Placement Model* (**GPM**) maintains the global view of the array by storing the global array into a common file. The second model, called the *Local Placement Model* (**LPM**) distributes the global array into one or more files (or distinct sections of one file) according to the distribution pattern. In this paper, we only consider the local placement model [CBH+94].

2.1.1 The PASSION Compiler

As a part of the PASSION (**P**arallel **A**nd **S**calable **S**oftware for **I/O**) project, we are developing a compiler to compile out-of-core programs. The PASSION compiler is targeted for out-of-core programs written using data-parallel languages like Fortran 90D [BCF+93] and High Performance Fortran [For93]. The PASSION compiler performs the following tasks

- Sequentialize data-parallel constructs (e.g. `FORALL` statements) and generate node code.
- Generate runtime calls for communication and I/O.
- Perform automatic program transformations to improve the I/O performance.

The PASSION compiler compiles an out-of-core HPF program in two phases. The first phase performs preprocessing in the source HPF program in the global name space. In the second phase, optimizations on the corresponding node program are carried out. While the first phase is independent of the underlying data placement model, optimizations in the second phase differ according to the data placement model.

- Phase I: *Global Program Preprocessing*

  In this phase, the source HPF program is analyzed in global name space to obtain flow and dependence information about the scalars and array variables. The main aim of the dataflow analysis is to examine the access and dependence patterns of the program variables. Using these data, the compiler can check whether the array statements could be reorganized so that the statements that access the common program variables are closer in the program space. The global program preprocessing also provides dependence information about the `FORALL` and array assignment statements. This information is used to detect and eliminate redundant computations.

- Phase II: *Local Program Optimizations*

  In the second phase, the PASSION compiler operates in the local name space. The second phase involves three steps, namely (1) Work Distribution, (2) Communication and I/O Placement and (3) Inter and Intra-file reorganization. A detailed description of these steps is given in [CBH+94].

In this paper, we focus on the global program reorganization and illustrate how the information obtained from the global program analysis can be used for local program optimizations.

2.2 An HPF Example

Figure 1 presents a fragment of an HPF program. Let us assume that `x, y` and `z` are square arrays of size `(1000,1000)`. Each array is distributed over p processors in a certain distribution using the HPF distribution directives. Assume that the available memory is too small to hold the local arrays.

```
x(1,1:1000)=y(1:1000,1)
z(1:1000:2,2:1000:4)=2
y(1:1000,1)=4
x(1:1000,1)=z(1,1:1000)
```

Figure 1: Motivating Example

Consider the first statement from Figure 1. It requires first row of array `x` and first column of array `y`. Assume that the data is stored in a column-major order (natural order for fortran), fetching a row of an array requires several separate I/O requests. Also, since the entire local array cannot fit in the memory, reading/writing of the arrays has be carried out several times. In the second statement array `z` is used. In order to make space for array `z`, arrays `x` and `y` need to be stored back to disks. In the third statement array `y` is accessed again and needs to be fetched back. Similar situation exists for arrays `z` and `x`, which are used again in statement 4. Consider statements 1 and 3. The same section of array `y` is used both these statements. If we can reorder the program so that statements 1 and 3 next to each other then data of array `y` can be reused, thus saving extra I/O accesses. Another advantage of such reorganization is knowing that a certain array section is to be accessed next, one can prefetch this array section. Such code reorganization requires knowledge about the dependence and dataflow relations in the program.

3 A Framework for Representing Dataflow and Locality Properties of HPF Programs

In this section we present a framework for representing an HPF program using a Program Locality Graph (PLG). Initially the source HPF program is analyzed and an Intermediate Program Locality Graph (IPLG) is generated. Using the dataflow and dependence information, IPLG is simplified to obtain the Program Locality Graph (PLG).

3.1 Intermediate Program Locality Graph

The *Intermediate Program Locality Graph* is a variation of a Program Flow Graph. A *Program Flow Graph* for a program segment P is a directed graph (V, E, v_0), where $v_i \in V$ is a weighted node which represents a basic block of P, E denotes the edges between the nodes and v_0 is the node representing the first basic block of the program.

We focus our attention on HPF [Hig93] programs. An HPF program provides a global view to the programmer. In other words, programmer can assume a single *thread* of execution when programming in HPF. Hence, we can represent an HPF program using a single program flow graph.

We restrict our analysis to the programs which use data-parallel constructs like `FORALL` and array assignment statements. Since array assignment can also be represented as a `FORALL` statement, we focus on `FORALL` statements in the rest of the paper[1]. By definition, a `FORALL` statement does not guarantee a fixed order of execution [Hig93]. Moreover, `FORALL` statements exhibit *copy-in-copy-out* semantics. Following the interval analysis of array sections presented in [GS90], it can be observed that each `FORALL` statement can be represented as an extended basic block (EBB) [AC76]. Each EBB represents a sequence of program instructions corresponding to the iterations of the `FORALL` statement.

A node v_i of the program flow graph represents an EBB. Considering a single thread of execution, we can assume a sequential order of execution. Therefor, each node can be numbered according to the position of the represented statement in the source program ($num(v_i)$). Each node is associated with a weight function. The weight function is a set of k two-element tuples $< ae_i, s_i >$, where k is the number of arrays used in the `FORALL` statement, ae_i is the number of *active*[2] elements of the array i and s_i is the access stride of the array i.

There is an edge e_i^j from node v_i to node v_j *iff* there is at least one array which is used in the corresponding `FORALL` statements. In general, two nodes v_i and v_j can have m edges, where m is the total number of out-of-core arrays used in the program. Each edge e_i^j represents the dataflow characteristics of the associated array between the corresponding `FORALL` statements. Node v_i is said to be a *predecessor* of node v_j *iff* $num(v_i) < num(v_j)$ and there is an edge between nodes v_i and v_j. Similarly, v_i is said to be a *successor* of node v_j *iff* $num(v_i) > num(v_j)$ and there is an edge between nodes v_i and v_j.

Since the program flow graph describes locality of the arrays in the source program, it will be called the *Intermediate Program Locality Graph* or IPLG[3].

3.2 Definitions

In order to introduce various operations on the IPLG, several definitions will be now defined. These definitions will be used throughout the rest of the paper.

3.2.1 Array Section Representation

During the scalar dataflow analysis, only the names of the variables are propagated. But for array dataflow analysis, in addition to the array name, array section information should also be transferred between the EBBs. Since we are focusing on out-of-core problems, listing every element of the array section will be cumbersome. We need a compact representation of array section which will simplify implementation of union and intersection of the dataflow definitions.

We represent the array section using an array section descriptor (ASD). Similar structures have been proposed by other researchers [Bal90, HK91, GSS94, GS93]. ASD represents the array section using the triplet notation $(l : u : s)$, where l is the lower bound, u is the upper bound and s is the stride. For example, a two dimensional array section is represented using two triplets $< (l_1 : u_1 : s_1), (l_2 : u_2 : s_2) >$. When the stride is 1, then the array section can be represented as a rectangular region (also described in [GS90]). All other types of accesses can be easily represented using the triplet notation. We can define the following operations on the ASD.

[1]For simplicity reasons, we will use examples having assignments statements.

[2]Active elements denote the elements of the array used in the current statement.

[3]Note IPLG is different than PLG defined by [FOW87]

- Intersection: An intersection of two ASDs, $ASD(i)$ and $ASD(j)$ is an ASD representing the *common* elements of these ASDs.
- Union: An union of two ASDs, $ASD(i)$ and $ASD(j)$ is an ASD which represents elements of both the ASDs.

3.2.2 Dataflow Definitions

Traditional dataflow approaches deal with analyzing dataflow properties of scalar variables. In such cases dataflow definitions need to know only the node (or the basic block in an interval) where a particular scalar is defined or killed. For array dataflow analysis, different sections exhibit different data flow properties. It is, therefore, necessary to study dataflow properties of individual array sections rather than that of the entire array. We use the subscript i for arrays, j for the array sections, k and l for PLG nodes. We define the following terms:

Node Definitions: Return node values

- $FIRACS(i)$ returns the index of the node where the array i is first accessed (defined or referred).
- $LASACS(i)$ returns the index of the node where the array i is last accessed.

Array Section Definitions: Return boolean values

- $GEN_i^j(k)$: An array section j of array i is said to be generated at node v_k if it is defined for first time in that node.
- $TRPS_i^j(k)$: An array section j of array i is said to be transparent at node v_k if it is not used in that node.
- $KILL_i^j(k)$: An array section j of array i is said to be killed at node v_k if there exists at least one element of the section which is killed by definition in node v_k.
- $COMP_i^j(k)$: An array section j of array i is said to be locally available at node v_k if there is at least one computation of the section in the node and the section is not killed at this node.

3.2.3 Data Dependence Relations

Using dataflow analysis we can obtain a clear picture of the access patterns of the data variables. However, as Maydan *et.al* have noted ([MAL93]), traditional dataflow analysis is not sufficient to analyze array computations. Maydan *et.al* point out two important deficiencies of dataflow analysis; (1) Dataflow analysis models accesses to array elements as accesses to the entire array; and (2) Dataflow analysis fails to *distinguish* between different instances of array accesses. These two deficiencies can be overcome by performing exact dependence analysis of array variables[4].
As described earlier, `FORALL` statements by definition, do not have dependence across iterations. However, dependence may exist between two `FORALL` statements. Specifically, if we describe the dependence problem as memory disambiguation, data dependence between two `FORALL` statements checks if any iterations of the two `FORALL` statements refer to the same location. In our case, we are concerned with a larger problem; i.e., (1) to find if there is a dependence between two `FORALL` statements, (2) if such dependence exists (between two array sections of the `FORALL` statements), to find the *active sizes*[5] of the array sections, and (3) compute the common *active* elements between the arrays sections in two `FORALL` statements. To represent these characteristics, we use *Dependence Access Relations*, $R_a^d(i) : v_j \rightarrow v_k$.
A Dependence Access Relation (DAR), $R_a^d(i) : v_j \rightarrow v_k$, captures the dependence as well as access information of the sections of array i between nodes v_j and v_k. The superscript d denotes the dependence pattern, i.e. true (R/W), anti (W/R), output (W/W) and access (R/R). The subscript a denotes the access information, i.e. whether two array sections share any elements and which array section has larger active elements. The parameter a can take the following values

[4]However only data dependence analysis is also not enough to capture the access information of the program. See [MAL93].
[5]Total number of active elements of an array section

- $<$: The array sections share common elements but the array section accessed in node v_j has more active elements.
- $>$: The array sections share common elements but the array section accessed in node v_k has more active elements.
- $=$: The same array section is used in nodes v_j and v_k.
- $\subset$: The array section accessed in node v_j is a subset of the array section accessed in node v_k.
- $\supset$: The array section accessed in node v_k is a subset of the array section accessed in the node v_j.
- ϕ: The array sections accessed in nodes v_j and v_k are distinct. In addition symbols $<, >, =$ are used to compare the number of active elements.

The dependence access relation is assigned boolean values depending on whether the dependency allows the motion of node v_j wrt node v_k[6]. Table 1 presents the boolean values of different DARs. A DAR has value 1 *iff* it allows code motion else it has value 0. Note that output dependency allows code motion *iff* the corresponding array sections are distinct. The DAR also satisfies the following properties

- **Associative Property** : DAR is associative between any pair of nodes of IPLG. We can replace a set of DARs between a pair of nodes by a resultant DAR obtained using the product of individual DARs.
- **Transitive Property** : DAR is not transitive over the nodes of IPLG.

To illustrate these properties, let us consider a simple example from Figure 2. The example shows three statements in which arrays `a`, `b` and `c` are used. Consider statements 1 and 2. In the IPLG, the corresponding nodes will be connected using two edges. The DAR for array `a`, $R^{W/R}_{>}(\texttt{a})$, has the boolean value 0. Similarly the DAR for array `b`, $R^{R/R}_{<}(\texttt{b})$, has the boolean value 1. Using the associative property of the DARs, we can replace these DARs with a new DAR $R_1^2 : v_1 \rightarrow v_2$ such that $R_1^2 = R^{W/R}_{>}(\texttt{a}) \circ R^{R/R}_{<}(\texttt{b})$, where $\circ$ denotes a boolean product. Therefore, R_1^2 will be 0. Similarly DAR between statements 2 and 3, R_2^3, has the value 0 and DAR between statements 1 and 3, R_1^3, has the value 1. It is easy to observe that $R_1^3 \neq R_1^2 \circ R_2^3$.

```
a(1:10)=b(1:10)
c(1:20)=a(1:20)+b(8)
b(20:30)=c(10:20)+a(23)
```

Figure 2: Properties of the DAR

It should be noted that according to the DAR R_1^3, statements 1 and 3 can be legally exchanged. However, this is not the case because statement 3 cannot be placed before statement 2 (since R_2^3 is 0). This example brings forward one serious problem associated with dependence information: *Data dependency describes relationship between any two pair of statements, however, it but clearly fails to predict relationship between more than two statements. In order to "view" the overall picture, we need global information (as provided by dataflow analysis).*

3.3 Program Locality Graph

The *Program Locality Graph* (PLG) is obtained from the Intermediate Program Locality Graph (IPLG). The Program Locality Graph is an *undirected complete* graph (V, E, v_0), where $v_i \in V$ is a weighted node which represents a basic block of P, E denotes the *weighted* edges representing the dataflow and dependence relations between the basic blocks and v_0 is the node representing the first basic block of the program.
In order to understand the properties of PLG, let us examine a sample HPF program fragment from Figure 3(a). It consists of four array assignment statements which use three arrays `a`, `b` and `c`. Figure 3(b)

[6]This code motion refers to motion that would modify the predecessor-successor relationship between the nodes.

Table 1: Boolean values of Access Relations

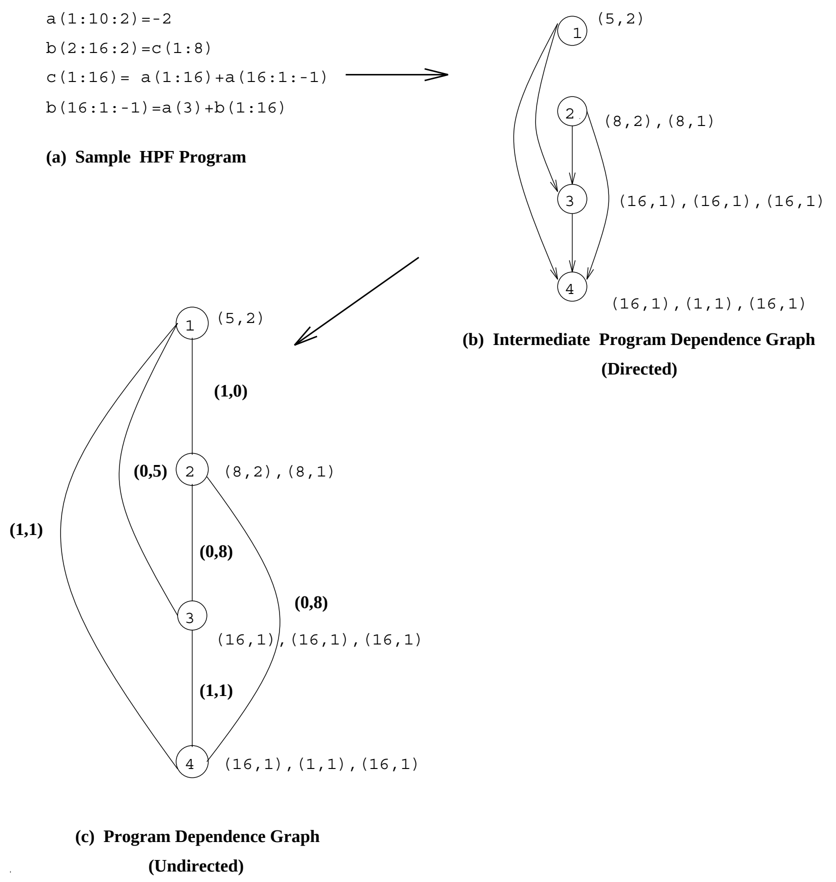
Dependency Pattern	Access Pattern	Boolean Value
R/R	$<, >, =, \subset, \supset, \phi$	1
R/W	$<, >, =, \subset, \supset$	0
R/W	ϕ	1
W/R	$<, >, =, \subset, \supset$	0
W/R	ϕ	1
W/W	$<, >, =, \subset, \supset$	0
W/W	ϕ	1

represents the corresponding IPLG. The statements are represented using four nodes which are ordered according to the statement ordering in the source. Each node is given a weight representing the number of active elements of each array used that statement. Each node is connected with another node if the corresponding statements access the same array. For example, statements 1 and 3 use the array a. Hence these nodes are connected using one edges. Statements 1 and 2 do not use any common array, and thus, these two nodes are not connected. The dependence between the nodes can be obtained considering the DAR relations. Using Table 1 and the transitive property of DARs, we can compute the resultant DAR between two statements and it's corresponding boolean value. Table 2 presents the DARs for the given program fragment. The second column presents the DARs for each pair of nodes. The third column represents the boolean value of the resultant DAR obtained using the transitive property of the DAR. Since nodes 3 and 4 do not share any array, there is no edge between v_3 and v_4. Hence, the boolean value of DAR between nodes v_3 and v_4 is 1.

Table 2: Access Relations for the sample HPF program

Statements	DAR	Boolean Value
$v_1 \longrightarrow v_2$	-	1
$v_1 \longrightarrow v_3$	$R(a)^{W/R}_{>}$	0
$v_1 \longrightarrow v_4$	$R(a)^{W/R}_{<}$	0
$v_2 \longrightarrow v_3$	$R(c)^{W/R}_{>}$	0
$v_2 \longrightarrow v_4$	$R(b)^{W/W}_{>}$	0
$v_3 \longrightarrow v_4$	$R(a)^{R/R}_{<}$	1

Using the resultant DARs, we simplify the IPLG considerably. We represent multiple edges between the nodes using the a single edge which represents the boolean value of the resultant DAR. If any two nodes were not connected in IPLG (i.e. the corresponding statements did not have any common array) then these nodes are connected using an edge representing a DAR of value 1. Therefore, in the Program Locality Graph each node is connected with the remaining ones. Hence the Program Locality Graph becomes a complete graph. We can now introduce the concept of *edge weight*, EW_j^k. We define a new variable $EDGE_j^k$ which represents the boolean value of the resultant DAR between the nodes v_j and v_k. Let $S_j(i)$ denote set of active elements of an array i in statement j (given by the node weights of the corresponding PLG). Then $S_j^k(i) = S_j(i) \cap S_k(i)$ computes the set of active elements of array i which are used in statements j and k. Let $Smax_j^k = max_i(S_j^k(i))$ denote the maximum number of active elements shared by an array between statements j and k. We can compute the edge weight EW_j^k as a two-element tuple $(EDGE_j^k, Smax_j^k)$. For example, statements 2 and 3 share 8 elements of array c and there is a R/W dependence between statements 2 and 3. Therefore, the edge e_2^3 has the weight EW_2^3 as (0,8). Figure 3(c) represents the resultant PLG. The edges of the PLG are associated with weights which represent dependence and access relations of the arrays between statements while the nodes of the PLG represent the access pattern of the arrays within a statement. Hence, we can represent the PLG as an undirected graph.



```
a(1:10:2)=-2
b(2:16:2)=c(1:8)
c(1:16)= a(1:16)+a(16:1:-1)
b(16:1:-1)=a(3)+b(1:16)
```

Figure 3: The Program Representation

3.4 Computing Array Access Variables

Using the PLG, dataflow equations and DARs we analyze the access pattern of the array variables. We compute the following variables:

- *Distance*: $d_i(j,k)$ is the distance between any two nodes v_j and v_k which are connected using an edge which represents the DAR $R_a^d(i)$. $d_i(j,k)$ can be computed as $|num(v_j) - num(v_k)|$. Distance array can be stored as an upper triangular matrix with 0s on it's diagonal.
- *Range*: r_i of an array i is the maximum distance $d_i(j,k)$. Range can also be computed using the dataflow variables $FIRACS$ and $LASACS$ as

  $$r_i = LASACS(i) - FIRACS(i)$$

  For example in Figure 3, $r_a = 3$, $r_b = 2$ and $r_c = 1$.
- *Cost*: C_i of an array i is the sum of the distances over all statements in which the array i is accessed. C_i can be computed as

  $$C_i = \sum_j \sum_{k>j} d_i(j,k)$$

  Cost C_i thus represents (1) how many times an array i is accessed in a program and, (2) the range r_i of an array i. For the example in Figure 3, $C(a) = 6$, $C(b) = 2$ and $C(c) = 1$.
- *Overall Range Cost*: θ is the sum of the costs of all arrays accessed in the program.

  $$\theta = \sum_i C_i$$

  For our example, the ORC is 9.

4 Range Reduction

We want to perform code motion in order to minimize the overall I/O cost of an out-of-core HPF program. The overall I/O cost depends on the following parameters: (1) Number of I/O calls, (2) Cost of each I/O call. Number of I/O calls are decided by the data access pattern in the program and the cost of an I/O call depends on the type of the I/O call and the amount of data accessed in each I/O call. The overall I/O cost can be reduced by reducing both parameters.
To reduce the number of I/O calls, we want to keep array data in memory as much as possible. To achieve this, we would like to bring statements accessing same array data closer to each other. Significant performance improvement can be achieved if the statement accessing the most used array are brought together. Cost of the I/O calls is reduced by aggregating I/O calls[7]. Specifically, we coalesce the I/O calls of the arrays having large I/O costs.
These specifications lead us to defining two cost functions, (1) Global Cost Function and (2) Local Cost Function.

- Global Cost Function: Global cost function tries to bring together the statements accessing same arrays by *reducing the Overall Range Cost* θ. In order to minimize θ, cost C_i (and in turn range r_i) of each array i has to be minimized. Hence this procedure is called the *Range Reduction*.
- Local Cost Function: Local cost function tries to optimize the I/O cost associated with each statement. Local cost function *chooses the array from a statement j, with respect to which, the statement(s) should be moved.* This array is called the *Dominant* array.

[7]Similar techniques have been used in optimizing interprocessor communication [Pon94].

4.1 Dominant Array Computation

The dominant array is chosen using the combination of following criteria

1. *Number of common elements with the other statements*: Each array is compared using the number of common elements it has with an another statement j, ($S_k^j(i)$). The array having the largest number common elements is chosen.
2. *Amount of stride in access*: The I/O cost, in addition to the total number of active elements, also depends on *how these elements are accessed.* Since each I/O call fetches consecutive block of data, accesses requiring strides require multiple I/O calls to complete the I/O operation. An array having strided accesses is chosen as a candidate. If there are several arrays with strided accesses, the array having the largest stride is chosen as a candidate. The information about the strides can be computed using the node weights of the PLG.
3. *Size of the active set*: The cost of an I/O call is directly proportional to the amount of data being read or written during the call. Hence an array having the maximum number of active elements is selected as a candidate.

Choice of the dominant array depends on all the three factors. We use a simple heuristic to choose the dominant array. An array which is selected in more than one category is chosen as the dominant array. However, if different array is chosen in each category then the array i having the maximum $S_j^k(i)$ is chosen as the dominant array (i.e. condition 1 is given the highest priority). These conditions guarantee that in most cases, the dominant array will have the maximum $S_j^k(i)$ and either the maximum number of active elements or access stride. In only one case, the dominant array will not have maximum $S_j^k(i)$. But in this case, the dominant array will have both the largest number of active elements and the largest access stride which will result in the largest I/O cost. Note that since the condition (1) chooses a dominant array of a statement i with respect to a statement j, *the dominant array is chosen for a pair of statements i and j.* Therefore, two pairs of statements may have different dominant arrays. For simplicity, we denote a dominant array between statements i and j as DA_i^j. Recall that edge weight between two nodes represents the largest number of elements shared between two statements ($Smax_j^k$). In other terms, weight of edge e_j^k represents the weight of the corresponding dominant array (i.e. DA_j^k).
Using the criteria 2 and 3, we can compute the *local I/O weight* of each array in a statement. Let $LW_j^k(i)$ denote the local I/O weight of a section k of array j in statement i. $LW_j^k(i)$ can be easily computed using the node weights of the PLG.

```
a(1:100:2)=b(1:50)+c(4)
c(10:30)=a(1:20)
b(1:100)=c(1:100)
```

Figure 4: Computing Dominating Array

Figure 4 presents an HPF program fragment consisting of three arrays `a`, `b` and `c`. Consider statements 1 and 2. In the first statement, arrays `a` and `b` both have 50 elements. However array `a` uses a stride of 2 and $S_1^2(a) = 10$. Hence, array `a` is the dominating array of statement 1 with respect to statement 2 (DA_1^2). Now consider statements 1 and 3. These two statements use arrays `b` and `c`. Since $S_1^3(b) = 50$, array `b` is chosen as a dominant array for statement 1 with respect to statement 3 (DA_1^3). Consider statements 2 and 3. In statement 2, arrays `c` and `a` both have 20 elements but $S_2^3(c) = 20$ which is greater than $S_2^3(b)$ (which is 0). Hence, array `c` is chosen as the dominant array DA_2^3.
The local I/O cost of a given node is said to be optimized if the node connected to it by the edge having the maximum weight is assigned either as a *successor* or *predecessor* of the given node (*In other words, if a statement accessing the dominant array is made a successor or predecessor of a given statement*). By optimizing the local I/O cost of a statement, we improve the program locality of the dominant array of that statement.

We can now frame the *Range Reduction Problem* as following
To reorganize the code so that the Overall Range Cost (ORC) is reduced. Each statement should be moved so that the local I/O cost is optimized and the dependence constraints are not violated.
Note that the problem does not aim to *minimize* the ORC. This is because there may be dependencies in the program which will prevent the program ordering corresponding to the *minimum* ORC. The problem will try to reduce the ORC as much as possible *without violating the dependence constraints.*

5 Range Reduction Heuristic

5.1 Description of the Range Reduction Heuristic

Figure 5 represents the range reduction heuristic. This heuristic takes the PLG as an input and returns a modified PLG. The HPF program is then modified according to the new PLG. This heuristic uses three costs for program reordering, (1) Global Access Cost $C(i)$, (2) Cost of the `DominantArray` in a statement $DA_j^k(i)$ and (3) Local I/O weight of the arrays in a statement $LW_j^k(i)$.
The heuristic uses *initialize* routine to read the PLG and initialize various data structures. An array `Element` is used to store names of the arrays used in the HPF program and each array is initially marked `MOVABLE`. Cost array `C` is initialized to store costs $C(i)$ of the arrays. Variable `NumArray` is assigned the value of total number of arrays used in the program. *update_ORC* takes array `C` as an input and returns the initial value of ORC. The *sort* routine sorts array `C` in a decreasing order of the cost. The `Element` array is also updated. The heuristic iterates over each array in the HPF program. In every iteration of the loop, the `MOVABLE` array having the largest access cost is chosen from `Element` and initialized to `CurrentArray`. `NumStat` is assigned the value of total number of statements (nodes in PLG) in which `CurrentArray` is accessed (*active nodes*). `Stat` is initialized with the list of active nodes and then sorted according the corresponding local I/O weights of the `CurrentArray`[8].
The innermost loop iterates over the active nodes of the `CurrentArray`. First the node accessing the `CurrentArray` having the largest local I/O weight is chosen and assigned to `CurrentStatement`. The `CurrentStatement` is passed to *find_new_position* routine. *find_new_position* routine returns the node to which the `CurrentStatement` could be moved so that the ORC is reduced. *find_new_position* routine takes as input the `CurrentStatement` and the PLG. The routine first analyzes the edges of the `CurrentStatement` and chooses the node connected by the edge having the maximum weight. Then the routine *find_new_position* computes the `DominantArray` using the heuristic presented in the previous section. If the `DominantArray` is marked `FIXED` then the `CurrentStatement` is updated to the statement having the next highest weight. If the `DominantArray` is `MOVABLE` then the the `CurrentStatement` can be moved along the edge (also termed as `CurrentEdge`) to reduce the local I/O cost of the `DominantArray`. The code motion is allowed *iff* dataflow and dependence constraints are not violated. Note that if two nodes are connected with an edge having $EDGE$ parameter 0, then only the code motion that modifies the existing predecessor-successor relationship is not allowed. The code motion that maintains the existing predecessor-successor relationship is permitted. Formally, we can classify the code motion into (1) Code Hoisting and (2) Code Pushing.

- **Code Pushing**

  A code is said to be pushed *iff* the corresponding node k in the PLG is moved to a position n such that $n > num(node(k))$. A code statement can be pushed from a position k to a position n *iff* following conditions are satisfied (Figure 7).

  Condition (a) stipulates that if a node k is to be moved before or after node n then $EDGE_k^l = 1, k < l < n$. According to condition (b), in order to have the code motion legal, *each* section j of the *every* array i used in node k may be read or not used in the nodes between k and n but may not be killed or generated in any of these nodes [9].

  Figure 6 illustrates two code pushing strategies. Consider the program dependence graph from Figure 6(b). Assume that node 2 is the `CurrentStatement` and edge e_2^8 is the `CurrentEdge`. Since

[8] The local I/O weights are compared using the (1) Access Strides and (2) Number of Active Elements
[9] In Figure 7, for simplicity, array and section notations are neglected. This condition is also termed as *availability* ($AVAIL$) [MR79]

```
range_reduction (PLG)
begin{
   initialize (PLG)
   update_ORC(C)
   sort (C) /* Sorting the cost array C */
   sort (Element)
   while (i < NumArray) do {
      CurrentArray= Element(i)
      NumStat =  Number of Statements in which CurrentArray is accessed
      Stat = Statements in which CurrentArray is accessed. /* Initialize Stat */
      sort(Stat) /* Sorting the statements according to local I/O weights */
      j=0
          while (j < NumStat) do {
             CurrentStatement=Stat(j)
             Node= find_new_position (Stat(j),PLG)
             if (Node !=CurrentStatement) then
                move(Node)
                update_ORC(C)
             endif
             j=j+1
             CurrentStatment=Stat(j)
          } endwhile
       status(Element(i))=FIXED
       i=i+1
    } endwhile
    return PLG
 } end
```

Figure 5: The Range Reduction Heuristic

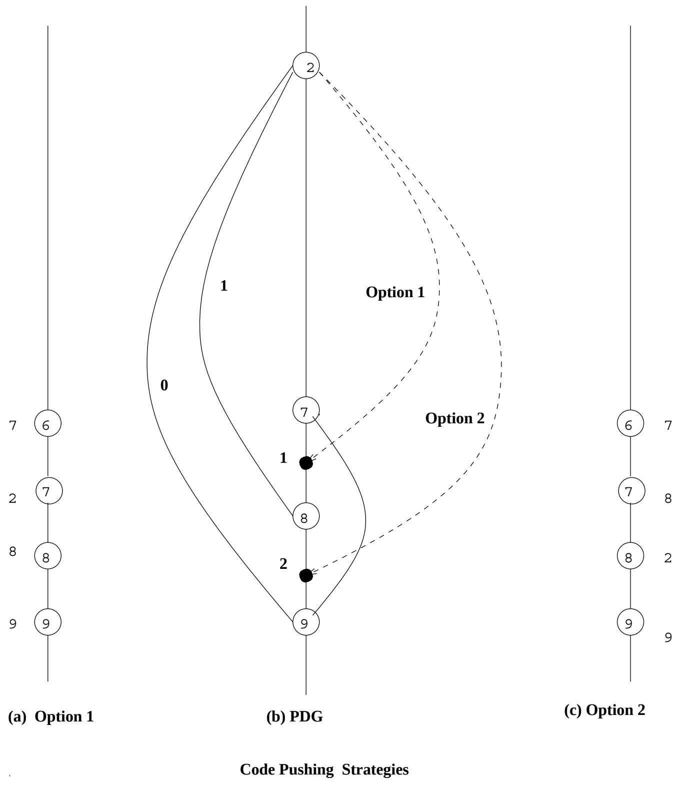


Figure 6: Code Pushing Strategies

$$\text{(a) } \prod_{k<m\leq n} EDGE_k^m = 1 \text{ (Dependence Condition)}$$
$$\text{(b) } \prod_{k<m<n} (\overline{KILL(m)} \cdot \overline{GEN(m)} \cdot (TRPS(m) + COMP(m))) = 1 \text{ (Dataflow Condition)}$$

Figure 7: Conditions for code pushing

$EDGE_2^8 = 1$, if the dataflow and dependence conditions are satisfied, node 2 can be moved either above (point 1) or below (point 2) node 8. These options are shown in Figures 6(a) and (c) respectively. Choice between option (a) and (b) can be found by using the $Smax$ value of the corresponding edge. Specifically, node k will be moved before node n (i.e. after node $n-1$) *iff* $Smax_k^{n-1} > Smax_k^{n+1}$ (Option 1) otherwise node k will be moved before node $n+1$ (Option 2). In Figures 6(a) and (b), original node numbers are shown outside each node. Figure 6 illustrates another important characteristics of code motion. Notice that $EDGE$ value of E_2^9 is 0 but the motion of node 2 with respect to node 9 is allowed since the existing predecessor-successor relationship is not modified.

- **Code Hoisting**

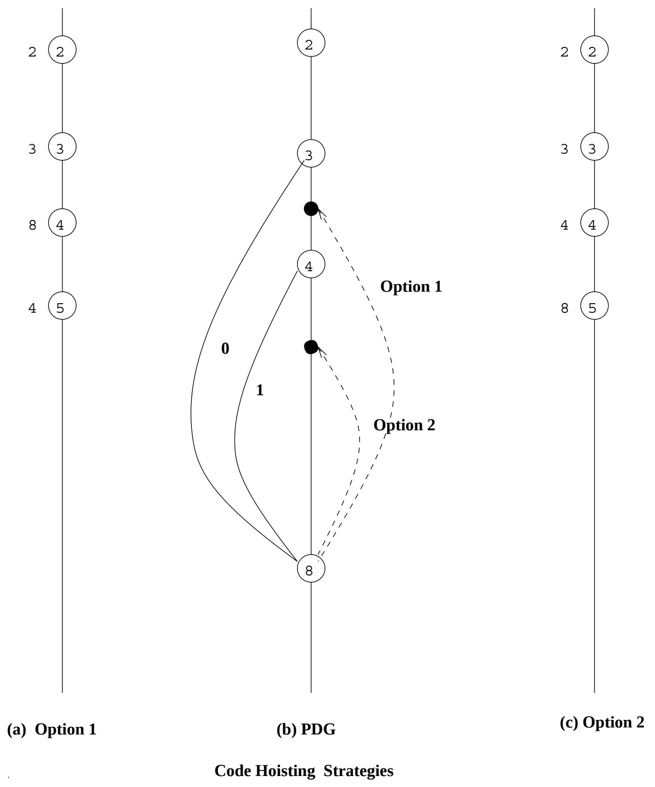


Figure 8: Code Hoisting Strategies

A code is said to be hoisted *iff* the corresponding node k in the PLG is moved to a position n such that $n < num(node(k))$. A code statement can be pushed from a position k to a position n *iff* following conditions are satisfied

$$\text{(a) } \prod_{k>m\geq n} EDGE_k^m = 1 \text{ (Dependence Condition)}$$
$$\text{(b) } \prod_{k>m>n} (\overline{KILL(m)} \cdot \overline{GEN(m)} \cdot (TRPS(m) + COMP(m))) = 1 \text{ (Dataflow Condition)}$$

Figure 9: Conditions for code hoisting

Condition (a) stipulates that if a node k is to be moved before or after node n then for each l $(k < l < n)$, $EDGE_k^l$ should be 1. According to condition (b), *each* section j of *every* array i may be read or not used in the nodes between k and n but may not be killed or generated in any of these nodes [10].

Figure 8 illustrates two code hoisting strategies. Consider the program dependence graph from Figure 8(b). Assume that node 8 is the `CurrentStatement` and edge e_8^4 is the `CurrentEdge`. Since

[10]For simplicity, array and section notations are neglected. This condition is also termed *anticipatibility ANTILOC* [MR79].

$EDGE_8^4 = 1$, if the dataflow and dependence conditions are satisfied, node 2 can be moved either above (point 1) or below (point 2) node 8. Choice between option (a) and (b) can be made by the same procedure as followed for code pushing. Figures 8(a) and (c) illustrate the two options.

If the dependence and dataflow constraints are not violated then new node is marked as a `POSSIBLE` position. The ORC is then updated according to the `POSSIBLE` position. If the new ORC is less than the original ORC then new position returned as the *new_position*. This condition ensures that if a statement is moved then both the global access cost of the `CurrentArray` and the local I/O cost of the `DominantArray` are reduced simultaneosly.
If no `POSSIBLE` position is obtained then the routine returns the value of the `CurrentStatement`. If the suggested node is a new statement then the `CurrentStatement` is moved to the required position and the ORC is updated. The `CurrentStatment` is then assigned the value of the next node in the `Stat` array. The process is repeated until all the statements using the `CurrentArray` are analyzed. Then the status of the `CurrentArray` is marked `FIXED` and the `CurrentArray` is assigned the next array from `Element`. This process is repeated until all the arrays are analyzed.

5.2 Analysis of the Range Reduction heuristic

Let us first verify if the heuristic presented in Figure 5 gives the correct *possible* result. Note that by definition, the range reduction problem tries to *reduce* the ORC *as much as possible* (not *minimize* the ORC). This condition is necessary because due to the dependence constraints inherent in a program, an heuristic may not be able to reorder the code to obtain the *least* ORC. Therefore, any range reduction heuristic tries to reorder the code so that the ORC is reduced and the dependence constraints are not violated.
The heuristic presented in Figure 5 can be viewed as an optimization heuristic which tries to minimize two independent costs. The global I/O cost measures the frequency and the range of access of each array in the entire program. On the other hand, the local I/O cost measures the I/O cost connected with each individual statement. Each step of our heuristic tries to optimize the local I/O cost of a statement such that the global I/O is also reduced. Our heuristic partitions the problem into n subproblems, where n is the number of arrays used in the program. The i^{th} subproblem tries to reduce the access cost of the i^{th} array by bringing together the statements that access the array. Since an assignment statement can have more than one participating array, statement reordering performed to reduce the access cost of an array, *may increase the access cost of another array*. To prevent this, two different arrays are used as targets for the two cost functions. The global I/O cost is minimized for the `CurrentArray` and the local I/O cost is minimized for a `DominantArray` of a program statement. In addition, two array status flags are defined; `FIXED` and `MOVABLE`. Any statement motion is said to be legal if it satisfies the following condition: *any statement can be moved* iff *it's* `DominantArray` *is* `MOVABLE`. This condition prevents extra computation. The proof is as follows: By definition, `DominantArray` is defined as an array having the largest I/O cost for a pair of statements. If the chosen `DominantArray` is `FIXED` then the particular array was previously both the `CurrentArray` and `DominantArray`. This means that a previous iteration has reordered the statement(s) accessing the chosen array and existing statement ordering is the *best possible* ordering for it. Hence, any new statement motion is not possible.
Due to the statement ordering condition, the time complexity of the heuristic will depend on the order in which the subproblems are solved. Ideally we would like to solve the subproblems with large number of statements first. As a result, when a `CurrentArray` of the i^{th} subproblem becomes `FIXED`, the statements in which this array was the `DominantArray` will be also `FIXED`. As a result, computations in the succeeding subproblems will be reduced. Therefore, the subproblems are ordered according to the access costs of the corresponding arrays. The first subproblem reorders the statements so as to reduce the access cost of the array having the largest cost. The second problem targets the array with the second largest cost and so on. At the end of the last subproblem, the access cost of each array will be reduced as much as possible. Each statement may be reordered in more than one subproblem. After a few iterations, the heuristic will reach a global minima *under dependence constraints* and the further statement reordering will not be possible.
Let us now analyze the time complexity of the heuristic. Let n denote the total number of arrays in the program and m denote the number of statements in the program. The heuristic consists of two main phases, an Initialization phase and an Iteration phase. In the initialization phase, the arrays `Element` and `C` are

sorted according to the access costs of the corresponding arrays. The sorting requires $O(nlgn)$ time. The iteration phase consists of two main loops, the outer loop iterates over the arrays and the inner loop iterates over the statements. Each iteration of the outer loop selects the `CurrentArray` and computes the number of statements in which this array is used. The inner loop of the heuristic sweeps over the statements which access the `CurrentArray`. These statements are ordered according to the local I/O weight of the `CurrentArray`. In general, this sorting can be performed in $O(mlgm)$ time, where m is the number of statements used in the program.

Each iteration of the inner loop operates on the `CurrentStatement` and tries to reduce the local I/O cost of its `DominantArray`. The computations to find `DominantArray` take constant time. But the execution time of *find_new_position* is bounded by m^2n. The routine needs to check the ORC for each statement. Using the algorithm presented in Appendix A, an ORC can be computed in $O(mn)$ time. Since each array may be used in m statements, *find_new_position* can be performed in m^2n time. If a statement is moved to the `POSSIBLE` node, the ORC is updated. The updating can be performed in $O(mn)$ time. Hence the time required for one iteration of the inner loop is $C + m^2n + mn$. This is a very loose bound because, as the heuristic progresses, the dependence constraints and the statement ordering condition increases the number of `FIXED` arrays and statements. Hence most of the expensive computations (like updating ORC) are not performed. Using the above result, the overall complexity of the heuristic can now be computed. Each outer iteration will require $O(mlogm + m(m^2n + mn))$ time. The number of outer iterations are n. Therefore, the worst case complexity of the Range Reduction heuristic is $O(nlgn + n(mlgm + m^3n + m^2n))$. Usually m is much larger than n. Hence, as compared to m^3n, we can neglect $nlgn, mlgm$ and m^2n to give the worst case complexity of the Range Reduction heuristic as $O(m^3n^2)$.

The best case complexity of the heuristic is obtained when the inner loop is executed only once. This happens when the array having the largest access cost is the `DominantArray` in all the statements. Therefore, when this iteration is over, all the statements in the program are `FIXED`. The best case complexity of the heuristic is $O(m^2n^2)$. In practice, most programs have sufficiently large number of dependencies. For such problems, the running time of the heuristic approaches it's best case value.

However, the running time of the heuristic becomes extremely large when there are no dependence constraints. This situation happens when there are no dependencies within the program statements or there are only R/R dependencies. Appendix B analyzes two such cases, namely, Code Motion with No Dependence (CMND) and Code Motion with Read Dependence (CMRD).

5.3 An Example of Program Reordering using Range Reduction

Consider the HPF program fragment illustrated in Figure 10 (a). The program uses three arrays `a`, `b` and `c`. In this program $r_a = 5$, $r_b = 4$ and $r_c = 6$. The costs of the arrays can be computed as

$$\text{C}(\texttt{a})=(1+2+4+5)+(1+3+4)+(2+3)+1=26$$
$$\text{C}(\texttt{b})=(1+2+4+6)+(1+3+5)+(2+4)+2=30$$
$$\text{C}(\texttt{c})=(1+3+6)+(2+5)+3=20$$

Though the range of array `C` is the largest, array `b` has the largest cost. The initial value of ORC is (26+30+20)=76. Table 5.3 presents the edge weights in the corresponding PLG. Note that edge weight shows the dependence and the local I/O weight of the `DominantArray` between a given pair of statements.

Let us analyze how this code is reorganized to obtain minimum ORC. In the first phase the arrays are sorted out according to their costs and the array having the largest cost, i.e. `b`, is chosen. Then the statements in which `b` is accessed are computed and the corresponding node numbers are stored in the array `Stat`. `Stat` is then sorted according to the local I/O cost of array `b`. From Figure 10, it can be observed that `b` is accessed in statements 1,2,3,5 and 7. Array `b` has the largest local I/O weight in statement 3, followed by statements 1,7,5 and 2. Therefore, `Stat` is sorted as 3,1,7,5 and 2.

First statement 3 is selected for motion. In statement 3, array `b` is the `DominantArray` (wrt to the statements 1, 2, 5 and 7). From the weight tableau, it is easy to observe that the edge e_3^7 has the largest weight. However, since $EDGE_3^7$ is 0 statement 7 can be hoisted to a point after statement 3. Since, $EDGE_4^7$ is 0 and $EDGE_5^7 = EDGE_6^7 = 1$, hence statement 7 can be brought up between statements 4 and 5. The *find_new_position* chooses node 5 as a `POSSIBLE` position. Then cost of arrays are updated and the new ORC isf found. The new access costs are

```
a(1:10:2)=b(1:10:2)+c(4)
c(5:10)=a(11:16)+b(1)
b(2:16:2)=b(5)+a(16:2:-2)
c(10:16)=-2
a(8:16)=a(1:8)+b(8:16)
a(1:8)=-4
c(1:16)=b(1:16)
```

(a)

```
a(1:10:2)=b(1:10:2)+c(4)
c(5:10)=a(11:16)+b(1)
b(2:16:2)=b(5)+a(16:2:-2)
c(10:16)=-2
c(1:16)=b(1:16)
a(8:16)=a(1:8)+b(8:16)
a(1:8)=-4
```

(b)

```
c(5:10)=a(11:16)+b(1)
a(1:10:2)=b(1:10:2)+c(4)
b(2:16:2)=b(5)+a(16:2:-2)
c(10:16)=-2
c(1:16)=b(1:16)
a(8:16)=a(1:8)+b(8:16)
a(1:8)=-4
```

(c)

```
a(1:10:2)=b(1:10:2)+c(4)
b(2:16:2)=b(5)+a(16:2:-2)
c(5:10)=a(11:16)+b(1)
c(10:16)=-2
c(1:16)=b(1:16)
a(8:16)=a(1:8)+b(8:16)
a(1:8)=-4
```

(d)

Figure 10: An Example Program

$$C(\texttt{a})=(1+2+5+6)+(1+4+5)+(3+4)+1=32$$
$$C(\texttt{b})=(1+2+4+5)+(1+3+4)+(2+3)+1=26$$
$$C(\texttt{c})=(1+3+4)+(2+3)+1=14$$

Now C(`b`)=26, C(`a`)=32 and C(`c`)=14. Hence the ORC is 72. Since the ORC is reduced, the statement 7 is moved to statement 5 and the PLG is updated. Figure 10(b) shows the reordered program.
After this statement movement, statements 1, 5 and 2 are examined for statement motion. Consider statement 1. The array `b` is the `DominantArray` of this statement. From the weight tablue, it can be observed that the edge e_1^3 has the largest weight. Since $EDGE_1^2$ is 1, statement 1 can be placed after statement 2. Hence the *find_new_position* routine chooses node 2 as a `POSSIBLE` position. The access cost of arrays are

$$C(\texttt{a})=(1+2+5+6)+(1+4+5)+(3+4)+1=32$$
$$C(\texttt{b})=(1+2+4+5)+(1+3+4)+(2+3)+1=26$$
$$C(\texttt{c})=(1+3+4)+(2+3)+1=14$$

The ORC is still 72. Though the ORC is not reduced, the local I/O cost of array `b` in statement 1 is decreased. Hence the statement 1 is moved to position 2 and the PLG is updated. Figure 10(c) shows the updated program. The remaining statements in the `Stat` array are checked, but they are found not suitable for motion. When all the statements in the `Stat` array are analyzed, array `b` is made `FIXED`.
The array `a` is chosen as the next `CurrentArray`. Statements 1, 2, 3, 5 and 6 are assigned to `Stat` (Figure 10(c)) and then `Stat` is reordered according to the local I/O weights of array `a`. The statements are ordered as 3,2,5,6 and 1. After repeating the earlier procedure, it is observed that the present statement orderings are optimal for array `a` and therefore, array `a` is made `FIXED`.
The array `c` is chosen as the next `CurrentArray`. Statements 1, 4 and 5 are assigned to `Stat`. Since `c` has the maximum I/O cost in statement 5, it is chosen to be moved first. Array `c` is the `DominantArray` wrt statements 1 and 4. Since statement 4 is the next statement in the program, statement 1 is checked for possible motion. From the dependence information, one can observe that statement 1 can be moved to any position before statement 5. Considering the locality of arrays `b` and `a`, the *find_new_position* routine chooses position 3 as the `POSSIBLE` position. The access cost of arrays is

$$C(\texttt{a})=(1+2+5+6)+(1+4+5)+(3+4)+1=32$$
$$C(\texttt{b})=(1+2+4+5)+(1+3+4)+(2+3)+1=26$$
$$C(\texttt{c})=(2+3+4)+(1+2)+1=13$$

The ORC is 71. Hence the statement 1 is moved to position 3 and the PLG is updated. Figure 10(d) shows the final program. Note that in the reordered program, any two consecutive statements access a common array.

Table 3: Tableau of the edge weights for the program in Figure 10

	1	2	3	4	5	6	7
1		(1,1)	(1,1)	(1,0)	(0,5)	(0,4)	(1,0)
2			(1,3)	(0,1)	(0,5)	(1,0)	(0,6)
3				(1,0)	(0,8)	(0,4)	(0,9)
4					(1,0)	(1,0)	(0,7)
5						(0,9)	(1,9)
6							(1,0)
7							

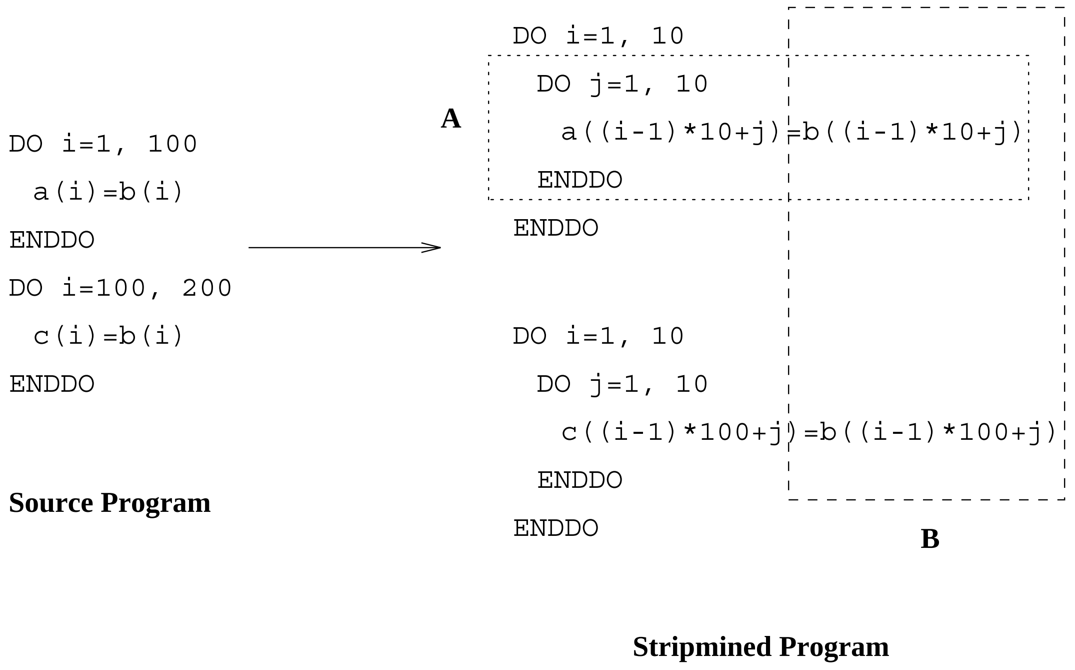
This example also illustrates that in presence of sufficient dependencies, the range reduction heuristic requires very few iterations.

6 Applications

Using the framework presented in Section 3, the compiler can obtain dataflow and dependence information about the program. Using this information and the Range Reduction Algorithm, the program will be reordered to improve the array locality. The compiler can take advantage of the improved array locality and use the knowledge about array access patterns to perform further optimizations. Some of these optimizations are briefly described below.

6.1 Providing Prefching Hints to file systems

In a reordered program, two adjacent statements can share at least one array. This information can be used by the compiler to prefetch the array data. In out-of-core problems, in-processor memory is at premium. Therefore, prefeched data can not be stored in an user buffer. To solve this problem, the compiler can provide *hints* to the filesystem so that necessary data can be prefeched into the system buffer.
In an out-of-core program, the compiler can give two types of hints, i.e. *Inter-statement* and *Intra-statement* hints. Intra-statement hints provide information about the access pattern of the arrays which are being used in the current statement (*active* arrays) and inter-statement hints provide access information about the arrays which are also used in the other statements. To illustrate these *hints*, let us consider the following node program fragment (Figure 11).



```
DO i=1, 100
  a(i)=b(i)
ENDDO
DO i=100, 200
  c(i)=b(i)
ENDDO
```

```
DO i=1, 10
  DO j=1, 10
    a((i-1)*10+j)=b((i-1)*10+j)
  ENDDO
ENDDO

DO i=1, 10
  DO j=1, 10
    c((i-1)*100+j)=b((i-1)*100+j)
  ENDDO
ENDDO
```

Figure 11: Compiler Directed Prefetching Hints

Figure 11 illustrates two `DO` loops. In the both the loops array `b` is used. In the first loop, first 100 elements of `b` are used while in the second loop next 100 elements are used. Figure 11(b) presents the corresponding stripmined code (The available memory is assumed to be 10). Regions **A** and **B** describe the code regions from where prefetching information could be obtained. Consider region **A**. In this region, consecutive *slabs* of array **B** are referred. Since compiler performs this stripmining, it can *ask the file system to prefetch the next slab into the system buffer.* This hint is called the Intra-statement hint since the data to be prefetched is required during the execution of the same statement. Consider the region **B**. In this region, next 100 elements of array `b` are referred. Using the framework described in the previous section, the compiler can find out what elements of array `b` are also read in the next `DO` loop. Using this information, the compiler can *advice the filesystem to fetch the next 100 elements* in it's system buffer. Knowing that both statements read two consecutive sections of an array, the compiler can further suggest to the file system to read the data in one chunk and store it in the system buffer. This hint is called the Inter-statement hint. The compiler (or the runtime system) then can read the data from the system buffer, thus avoiding extra disk accesses.

6.2 Data Retaining Policies

As described earlier, out-of-core computation involve computations on *in-core* array slabs. Each out-of-core computation is performed in several phases, each phase reads data into memory, performs computations and writes the data back to disks (if necessary). Therefore, performance of an out-of-core program depends on how many times an array is read or written back onto the disk. In order to reduce this I/O cost, efficient memory management strategies are required. Specifically, a memory management strategy would (1) allocate memory to a array section depending on it's access pattern in a statement or in the entire program and (2) decide if a particular array section should be replaced from the memory and written to the disks. These problems can be collectively called *Data Retaining Problems*.
The Data Retaining Problem is a magnified (by several orders of magnitude!) version of the Register Allocation Problem. The available memory can be considered as a large set of registers which needs to be allocated to more than one array sections. When the computation is over either the entire or part of the array section can be replaced from the memory. Thus during the course of the program, number of registers allocated to an array change dynamically. Thus, the Data Retaining Problem can be viewed as the *Dynamic Register Allocation Problem*. Indeed this problem is a difficult one and we will describe it in detail separately. For purposes of this paper, we will describe how the information provided by our framework can be used for solving this problem.
Our framework provides the following information:

- **Program Information**
  - Number of arrays are used in the program.
  - Global access and dependence information about the arrays.
- **Array Information**
  - Access cost and range of the arrays.
  - Number of statements accessing the arrays and local cost of each statement (as a function of the cost of participating arrays).

Any memory management strategy will allocate memory according to the access cost of the arrays. The choice of an array can be made using either its global access cost or the local I/O cost (node weights of the PLG)[11]. The global access cost of an array i can be obtained using the access cost variable $C(i)$. Similarly, in a statement, the array with the largest local cost can be easily obtained. Another important parameter is the access pattern of the array. The distance matrix provides a very compact way of representing the access pattern of an array. The data replacement policies can use this metric very effectively. For example, the replacing policy may want to replace the array which is not used in near future; however, at the same time retain the array which has highest local cost. The array which is used the furthest can be found by checking the distance matrix and the array with the largest local cost can be found using the local I/O weights.

6.3 Dead Code Elimination

A computation (specifically *a definition*) of an array section is said be *dead* (redudant) *iff* the array section is killed before being referred again. Formally, let the variable $REDUN_i^j(k,l)$ denote the redundancy of a section j of array i between nodes k and l. Then $REDUN_i^j(k,l) = TRUE$ if the following equation is $TRUE$
If an array definition is redundant, then it can be replaced by the definition has killed it. This replacement involves statement movement. Suppose definition at node l kills the definition at node k. If node l is not

[11]Notice the similarity to Global and Local Register Allocation Policies

$$GEN_i^j(k) \cdot (\prod_{k<m<l} \overline{KILL_i^j}(m)) \cdot (\prod_{k<m<l} TRPS_i^j(m)) \cdot KILL_i^j(l)$$

Figure 12: Dataflow Conditions for Dead Code Elimination

a predecessor or successor of node k then the node l can either be hoisted or pushed down. The statement movement has to be carried according to the conditions in Figures 9 and 7. Figure 10(d) illustrates redundant computation elimination. Consider statements 4 and 5. Statement 4 defines a section of array `c` which is immediately killed by statement 5. Since the array section defined in section 4 is not used in statement 5, the definition of the section in statement 4 can be considered redundant and can be eliminated.

6.4 Local File Reordering

Our out-of-core HPF compiler uses the local placement model as an underlying execution model. As described earlier, in this model, each processor stores it's out-of-core local array (OCLA) in a separate *logical* file called the local array file (LAF). Each local array file is stored on a *logical* disk associated with each processor. The local array files are normally required during computation, therefore, they are generated as scratch files. Since data belonging to a processor is stored together in a separate file, locality in processor space is translated into locality in file space[12]. If each local array file is stored on a separate disk, the locality in file space gets translated into locality in disk space.

The range reduction algorithm improves the locality of the arrays in an HPF program. Local file reordering tries to reorder the data in a local array file so that it matches the access pattern of the corresponding array in the source program. Local array reordering distributes the array elements into three different data-sets, (1) elements which are accessed only once, (2) elements which are accessed more than once and (3) elements which are not accessed at all. These groups are stored separately (and consecutively) in the local array file. Moreover, the elements which are accessed only once, are stored in the order in which they are accessed. As a result, the locality in the program space gets translated into locality in file space.

The information required by the file reordering algorithm (access patterns, array dataflow information) can be easily computed from the our framework. The compiler can use the local file reordering to translate the locality in program space to the locality in file space. The reorder local file improves the I/O performance by allowing contiguous data transfers and data prefetching [BC95].

7 Conclusion

We have presented a framework to describe the dataflow and dependence information of an HPF program. Our framework represents an HPF program as an undirected weighted Program Locality Graph (PLG), where an edge represents the dependence between two statements sharing an array section and a node weight represents the access pattern of the arrays used in a statement. We have defined a new boolen descriptor, called *Dependence Access Relation*, to provide a concise representation of the dependence and access pattern of the array sections.

The information provided by the PLG is used by the compiler to reorder the program so that any pair of statements using a common array is brought close to each other. We approach the program reordering problem as an optimization problem which tries to reduce two independent costs simultaneosly. To solve this problem, we have presented a polynomial time algorithm, called the *Range Reduction Algorithm*. The best case complexity of the algorithm is $O(m^2n^2)$, where m is the number of statements and n is the number of arrays used in the program. In presence of sufficient dependence constraints, the running time of range reduction algorithm approaches the best case performance. The worst case running time of this algorithm is $O(m^3n^2)$.

The framework is being implemented in the PASSION compiler to compile out-of-core data parallel programs. Taking advantage of the restructured program and the information provided by the PLG, the PASSION

[12]Elements belonging to one processor are said to have locality in processor space. Similarly elements lying in one file are said to have locality in file space.

compiler performs further optimizations like (1) providing cache hints to the file system, (2) choosing a good data management strategy, (3) eliminating dead code and (4) reorganizing local array files.

Acknowledgments

Authors thank Rakesh Krishnaiyer for carefully reading the paper and giving constructive hints.

References

[AC76] F. E. Allen and J. Cocke. A Program Data Flow Analysis Procedure. *Communications of the ACM*, 19(3):137–146, March 1976.

[AL93] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. *SIGPLAN Notices*, 6(28):126–138, 1993.

[Bal90] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: the data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, 1990.

[BC95] Rajesh Bordawekar and Alok Choudhary. Compiler directed file reorganization strategies. Technical Report 696, Northeast Parallel Architectures Center, Syracuse University, February 1995.

[BCF$^+$93] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. In *Proceedings of Supercomputing '93*, pages 351–360, November 1993.

[CBH$^+$94] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: Parallel and Scalable Software for Input-Output. Technical Report SCCS–636, NPAC, Syracuse University, September 1994.

[CK94] Steve Carr and Ken Kennedy. Scalar Replacement in the Presence of Conditional Control Flow. *Software-Practice and Experience*, 24(1):51–77, January 1994.

[Dha88a] D.M. Dhamdhere. A fast algorithm for code movement optimization. *SIGPLAN Notices*, 23(10): 172–180, 1988.

[Dha88b] D.M. Dhamdhere. Register assignment using code placement techniques. *Computer Languages*, 13(2):75–93, 1988.

[Dha90] D.M. Dhamdhere. A usually linear algorithm for register assignment using edge placement of load and store instructions. *Computer Languages*, 15(2):83–94, 1990.

[FGS94] Jeanne Ferrante, Dirk Grunwald, and Harini Shrivivasan. Computing communication sets for control parallel programs. In *Seventh Workshop on Languages and Compilers for Parallel Computers*, pages 21.1–21.16, 1994.

[For93] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0. Technical Report CRPC-TR92225, Center for Research in Parallel Computing,Rice University, January 1993.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependnce Graph and its use in Optimization. *ACM. Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[GS90] Thomas Gross and Peter Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software-Practise and Experience*, 20(2):133–155, February 1990.

[GS93] Manish Gupta and Edith Schonberg. A framework for exploiting data availability to optimize communication. *Proc. of Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[GSS94] Manish Gupta, Edith Schonberg, and Harini Srinivasan. A unified data-flow framework for optimizing communication. *Proc. of Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, pages 18.1–18.15, August 1994.

[GV91] E. Granston and A. Veidenbaum. Detecting redudant accesses to array data. In *Proceedings of Supercomputing'91*, November 1991.

[Hig93] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[JD82a] S.M. Joshi and D.M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization part i. *International Journal of Computer Mathematics*, 11:21–41, 1982.

[JD82b] S.M. Joshi and D.M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization part ii. *International Journal of Computer Mathematics*, 11:111–126, 1982.

[KLS$^+$94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *High Performance Fortran Handbook*. The MIT Press, 1994.

[KN94] Ken Kennedy and Nenad Nedeljkovic. Combining dependence and data-flow analyses to optimize communication. Technical Report CRPC-TR94484-S, CRPC, Rice University, September 1994. URL= gopher://softlib.rice.edu/99/softlib/CRPC-TRs/reports/CRPC-TR94485-S.ps.

[MAL93] Dror E. Maydan, Saman Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *ACM Symposium on Principles of Programming Languages*, 1993.

[MR79] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[Pon94] R. Ponnusamy. *Runtime Support and Compilation Methods for Irregular Computations on Distributed Memory Parallel Machines*. PhD thesis, Department of Computer Science, Syracuse University, Syracuse, NY, May 1994. Available as NPAC Technical Report SCCS–633.

[TBC94] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 382–391, July 1994.

[vK93] R. von Hanxleden and K. Kennedy. A Code Placement Framework and its Application to Communication Generation. Technical Report CRPC-TR93337-S, Center for Research in Parallel Computing,Rice University, October 1993.

[vKK$^+$92] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler Analysis for Irregular Problems in Fortran D. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, August 1992.

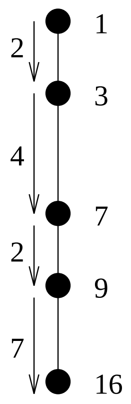
A An $O(m^2n)$ Algorithm for computing the ORC

The Overall Range Cost (ORC) is sum of the access costs of all arrays in a program (Section 4). The access cost of an array can be computed as a sum of the elements of the *strictly* upper triangular distance matrix. This computation requires $O(m^2)$ operations. Since there are n arrays, the ORC can be computed in $O(m^2n)$ time.
In this section, we describe a $O(m)$ algorithm to compute the access cost of an array. Let us consider an array i which is used in statements 1, 3, 7, 9 and 16. Let us define an index array which stores the index of the statements. Figure 13(A) illustrates the corresponding distance matrix. The access cost $C(i)$ can be computed as a sum of the elements of the distance matrix. Figure 13(B) illustrates the access cost computation. The access cost of the array i is 72. Figure 13(B:2) presents a *per-statement* way of computing the access cost. It can be observed that the access cost of array i can be computed using only four values; 2, 4, 2 and 7; which are nothing but the offsets between the consecutive statements (Figure 13(C)). We can define an offset array and initialize it as $offset(i) = index(i) - index(i-1)$. Therefor, offset(0)=0, offset(1)=2, offset(2)=4, offset(3)=2 and offset(4)=7.

$$\boxed{\begin{array}{cccc} 2 & 6 & 8 & 15 \\ & 4 & 6 & 13 \\ & & 2 & 9 \\ & & & 7 \end{array}}$$

```
2+6+8+15=31          (2)+(2+4)+(2+4+2)+(2+4+2+7)=31
  4+6+13=23                    (4)+(4+2)+(4+2+7)=23
     2+9=11                            (2)+(2+7)=11
       7=7                                   (7)=7
       ---                               ---------
        72                                      72
```

(A) Distance Matrix **(B) Access Cost Computations**



$$\boxed{\begin{array}{cccc} 2 & 4 & 2 & 7 \\ 2 & 4 & 2 & \\ 2 & 4 & & \\ 2 & & & \end{array}}$$

```
          7=7
  (2*2)+(7)=11
 (4*3)+(11)=23
 (2*4)+(23)=31
 -------------
           72
```

(C) Computing Offsets **(D) Offset Matrix** **(E) Computations using the offsets**

Figure 13: Computing Array Access Costs

Let us consider the offset matrix Figure 13(D). The offset matrix is a square triangular matrix of size $(m-1, m-1)$, where m is the number of statements. For our example, offset matrix is a (4,4) matrix. The offset matrix is initialized as follows: $m-1+j$ rows of the j^{th} the column are assigned the value of $offset(j+1)$, $0 \le j < m-1$. This condition allows us to represent the *entire upper triangular matrix as a* $(0 : m-2)$ *vector whose* k^{th} *elements should be used* $(m-1-k)$ *times* $(0 \le k \le m-2)$.
Using the offset vector (computed using the index vector), we can compute the access cost of an array in $O(m)$ time. Figure 14 shows the $O(m)$ algorithm. Each step of the algorithm computes an intermediate value of the ORC (`sum`) using the current offset and the value of the accumalated offset from the previous iteration. Figure 13(E) shows the computations according to the new algorithm. In the first step, current

```
sum=0
temp=0
do i=0, m-2
  sum=sum+(temp+(i+1)*offset(m-1-i))
  temp=temp+offset(m-1-i)*(i+1)
enddo
```

Figure 14: The Array Access Cost Algorithm

offset is 7 and accumalated offset is 0. After the execution of the first step, `sum` is 7 and the accumalated offset (denoted by `temp`) is also 7. In the second step, using the current offset is 2 and the accumalated offset 7, `sum` is computed to be 11 and so on. The overall computation can be performed in $O(m)$ time. Since there are n arrays, the ORC can be computed in $O(mn)$ time.

B Analysis of the Restricted Code Motion Problem

This section analyzes the complexity of the Range Reduction Algorithm for two types of codes, (1) Code with no dependencies (CMND) and (2) Code with read dependencies (CMRD).
Figure 15(A) presents an HPF program fragment with no data dependencies. As a result, our framework would present this program as a PLG with edges having edge weights (0,0). Since this code has no dependence constraints, one can place statements in any order. In order to find the statement ordering which corresponds to the minimum ORC, one has to perform an exhaustive search of the problem space.
Suppose the number of statements in a program segment are m and the number of arrays are n. Therefor, the number of elements in the problem space is $m!$. For each case, the algorithm needs to check the value of ORC, which requires $O(mn)$ time. Hence, the overall complexity of the Range Reduction Algorithm will be $O(m!mn)$, which is loosely bounded by $O(2^m)$. Thus for the code having no data dependencies, the Range Reduction Algorithm runs in exponential time.
One can view this problem as an cluster ordering problem. We can organize m statements into i distinct clusters. The i^{th} cluster consists of all the statements accessing the i^{th} array. Thus, two distinct clusters may share statements. In this case, the statement ordering will depend on *how the clusters are ordered and how the statements within a cluster is ordered.* If m' denotes the number of statements in the biggest cluster, then the complexity of the Range Reduction Algorithm will be loosely bounded by $O(n!m'!mn)$, which is also exponential.

```
a(1:12:1)=b(12:1:-1)
c(10:20:2)=a(20:30:2)+b(40:45)
b(50)=b(64)
c(30:60:3)=a(40:60)
```

(A) Code with No Dependencies

```
a(2:12:1)=b(1:11:1)
c(10:20:2)=b(1:6)+b(40:45)
b(50)=2
c(30:60:3)=a(30:41)+b(42)
```

(B) Code with Read Dependencies

Figure 15: Restricted Code Motion Problem

Consider the program fragment from Figure 15(B). This program is an example of a code with read-read dependencies (CMRD). This code will be represented as a PLG with non-zero edge weights. Since the read-read dependence does not prevent statement motion, one still needs to perform an exhaustive search of the problem space. This problem is therefor, a variant of the CMND problem described before. Hence, the complexity of the Range Reduction algorithm will be exponential.

This problem can be viewed as a cluster growing problem. Initially a pair of statements having the largest edge weight (i.e. sharing the largest number of elements) is chosen as a *seed* for the cluster. Then the cluster is grown by choosing a connected statement having the largest edge weight. This process is continued until all the statements are used and the ORC is computed. The process is repeated for different arrangements of statements for the given *seed.* In order to do an exhaustive search, one needs to use each statement as a seed. The complexity of this problem is also exponential, as a result, the Range Reduction Algorithm runs in exponential time.