BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-95-M15

# "Identifying Failure Modes in Compiler Algorithms applied to Distributed Memory Data Parallel Computation"

by

Peter A. Walker

# Identifying Failure Modes in Compiler Algorithms applied to Distributed Memory Data Parallel Computation

Peter A. Walker

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University.

May 1994

# Identifying Failure Modes in Compiler Algorithms applied to Distributed Memory Data Parallel Computation

Peter Walker

This document describes the results of research examining a representative set of published algorithms used in automatic compilation of code for data parallel computation on distributed systems; we determine the data dependence conditions, as expressed in the structure of the input program, that result in these algorithms producing output code which generate incorrect result upon execution. Methods for instrumenting these compilation environments to enable run-time detection of errors in executed code are also suggested.

## 1.0 Introduction

This work examines four representative algorithms that have been developed and used in compilers for the automatic generation of code that is executed in parallel on a distributed memory architecture; in particular, all data transfer between processors is managed solely by the compiler, thereby releasing the user from the complexity associated with programming inter-processor communication in this environment. The model of parallel computation addressed by these algorithms is the data-parallel model [11], which is defined as a method of parallel computation where the data used in a given computation is decomposed into smaller subsets that may be independently operated on by computing nodes executing the same program. This form of parallel program is described as SPMD (single-program, multiple-data [24]). The algorithms are examined for identifying stated and unstated assumptions on how data dependencies in source code are addressed during compilation and, to ascertain how these assumptions are applied to realize parallelism in an execution; the identification of the constraints embedded in such assumptions (which defines the scope of application of the algorithms) is discussed. Also, the static and/or dynamic data dependence types in source code that would trigger a violation of these constraints are explored. Further, it is determined whether the parallel execution from code generated by compilers using these algorithms will exhibit data races as, the presence of data races

undermines confidence in the results produced from such executions. Additionally, analysis is done to determine how the behavior of executions is affected with source code having different data dependence types -- whether the computation will consistently generate an incorrect output, exhibit data races or deadlock. Finally, for each algorithm that has the potential to generate incorrect code, a run-time method for detecting these conditions during an execution of the compiled code is developed.

## 1.1 Terms and Definitions

In the context of this paper, a failure in the discussed compiler algorithms is defined as follows:

**Definition:** A parallelizing compiler algorithm is defined to have failed if the results of computation generated from the execution of code produced by the application of that algorithm differs consistently, or intermittently, from the results of code generated by a sequential algorithm supplied with the same input and which is known to produce correct (or desired) results.

The types of failures anticipated are

(1) Deadlock [18] arising from incorrect communication patterns between processors.

(2) Erroneous results due to the persistent violation of data dependencies [9] between shared variables.

(3) Non-determinacy arising from the presence of data races [19]. The Bernstein conditions [23] define the types of unsynchronized accesses by processors to a variable that create data races. However, a common feature of many of the distributed data parallel methods referenced, is the *owner's compute rule* [3]; shared data structures are partitioned such that processor nodes participating in the computation are exclusively responsible for writing to a subset of the original structure [1, 2, 4, 6, 12] for which they have been assigned ownership. Unordered write-write accesses to the local subset of the node is therefore avoided. However, without a mechanism to enforce the order between writes (by the owning processor) and reads (by

remote nodes), data races will occur. Of the four methods discussed in this paper, only one does not use the owner's compute rule.

### 1.1.1 Data Parallel Computing Environment

The data parallel paradigm [11] of computation in a distributed memory system, can be described as a computing environment that emphasizes the partitioning of an iteration space and associated data (over which the iterative operation is performed), among the compute nodes in the system for concurrently operating on the data placed at those nodes. For example, consider the code shown below.

***Example 1***

```
do i=1 to N
        A(i)= B(i) + C(i)
 end do;
```

Using data parallel method to perform this computation, each of the arrays A, B and C would be partitioned among the nodes such that each node would compute for a subset of the values of the range 1..N, the iteration space covered by the iteration index ***i***. Ideally, the data elements of the each array reference in an iteration ***i***, would be placed at the computing nodes such that only a local reference is required to access it.

The intent is to maximize on the parallelism in a given computation for minimizing total computation time, while maintaining correctness. As such, source code with iterative structures that have little or no loop-carried data dependence [9] is most suited for execution in this environment. This approach differs from *functional* (or task) parallelism in which independent functions in a segment of code are assigned to separate compute nodes (multi-threaded) for execution. An example of functional parallelism is a system making remote procedure calls to multiple nodes for them to perform specific operations at those nodes, e.g., for one node to execute the command 'ls,' another to start a database process, etc.

### 1.1.2 Data dependence Considerations

The definition given for correctness in the code generated for parallel execution centers on the requirement that the result of the parallel execution is the same as that generated from a uni-processor execution with the same input data. Correctness can further be traced to the non-violation of data dependence between statements in the parallel execution. From [ 9] three types of data dependencies can be identified:

1. If statement S1 uses the result of another statement S2, then S1 is ***flow dependent*** on S2.
2. If S1 can store only after S2 fetches the data stored in that location, then S1 is ***anti-dependent*** on S2.
3. If S1 overwrites the result of S2 then S1 is ***output dependent*** on S2.

Thus, data dependencies dictate the execution precedence among statements that must be honored for consistent correct results to be produced. Consider the following code fragment:

***Example 2***

```
  DO I = 2, N-1
S1:     A(I) = ....
S2:       . = A(I+1)
S3:       A(I-1) = ...
S4:     . = A(I - 1)
ENDO
```

In the above example, S3 is output dependent on S1, since it overwrites the results of S1 and the execution of S3 in iteration *I* must follow the execution of S1 in iteration *I-1*. Also, statement S1 is anti-dependent on statement S2 in that it overwrites the value read at S2. Thus the execution of statement S1 in iteration *I* must follow the execution of S2 in iteration *I+1*. S4 is flow dependent on S3 because it reads results of S3; the execution of S3 must precede the execution of S4.

When the data dependence between statements extends across iterations, it is called ***loop-carried*** data dependence. The presence of control directives such as if-then-else statements can modify the execution precedence between two statements. Where the order of execution is modified by the presence of control statements, a ***control dependence*** is said to exist between the statements.

It is also possible for the data dependence between statements to change at runtime; e.g., Consider the assignment y(b(i)) = y(b(i+1)); if all b(i) < b(i+1) then iterations have loop carried anti-dependence. However, if all b(i) > b(i+1), the iterations have a loop-carried flow dependence. Where the data dependence between statements can change at runtime, a ***dynamic*** data dependence is said to exist; if this is not possible the data dependence is described as ***static***.

If data dependence occurs across several iterations of a loop, the distance is called the ***dependence distance*** (with respect to the loop). In example 2, the output data dependence between S3 and S1 is 1. In example 2, all data dependencies have constant distance. For statements nested in multiple loops, the dependence distance of each statement with respect to the each loop may be different. Dependence distance provides an index of the degree to which a loop may be parallelized. A loop in which all statements have zero dependence distance for assigned variables is fully parallelizable. Where non-zero dependence distances occur, loop-skewing or strip-mining [4] is required to correctly parallelized the execution of the code. Thus a knowledge of dependence distance is important for extracting parallelism in a fragment of code. For simple loop indices (linear) [9] this is may be determined at compile time. However, for non-linear subscripts, run-time evaluation may be needed to assess the dependence distance between statements, for evaluating the degree to which the execution may be parallelized.

The ***dependence direction vector*** [9] specifies the direction of dependence distance of a loop index in an assignment statement with respect to the loops nesting the statement. Each element of the dependence distance vector is called a ***dependence direction*** . For example, consider a nest of two loops, where the outer loop has index *I*, and the inner loop has index variable *J*. Assume there is data dependence between

two statements S1 and S2 such that the execution of S1($i_1 j_1$) must precede that of S2($i_2 j_2$). The dependence direction for the $J$ loop is "<","=" or ">" depending on if $j_1 < j_2$, $j_1 = j_2$ , or $j_1 > j_2$ respectively. For the $I$ loop the dependence direction is similarly determined. The vector formed from the consideration of the dependence direction is the *direction vector*. Again, the dependence vector is a crucial indicator of the degree of parallelism that exist in a given program module.

### 1.1.3 Data Races

Data races [19] exist when the order of read/write or write/write operations to a variable occurs without some mechanism for enforcing a fixed order of access to the variable. In a SPMD execution governed by the owner's compute rule [3], the execution of each processor is such that a single processor is responsible for updating a subset of data elements to which it is assigned write ownership. This ensures the process of updating a variable, is race free in the write/write sense. However, this does not eliminate races or non-determinism in the parallel execution; this follows as the relative temporal ordering of reads of a variable and writes by the owner node may vary from one execution to another in the absence correct inter-processor synchronization.

In the Multiple-Data-Single-Compute (MDSC) model, multiple nodes may write to a single variable. In this situation, write/write data races can occur, thereby leading to unpredictable outcome if the values written are different.

On message passing systems, as examined in this work, data races are manifested as races in messages between communicating processors. In [22] a message race between two messages is defined as a situation in which either message could have been accepted first by some receive due to variations in message latencies or process scheduling.

### 1.1.4 Data Dependence violation and Data Races

The data dependence between two statements specifies the order in which those statements can be executed (one after the other or in parallel). A data race however, by definition, is an observation of the order in which concurrent processes without explicit synchronization, access a variable used in those processes and in which the

purpose of access is contrary -- say one reads the variable and the other writes it. Data race can exist even if an execution correctly satisfies the data dependence between two statements. For example, one processor, say p1, may write a variable and another, say p2, reads it and p1 writes the variable again without any synchronization. The value read by p2 may correctly satisfy the data dependence between the executed statements but the execution would be classified as one that involved a race, as synchronization was absent between the processors to enforce a definite order in access to the variable. An alternative execution could have been that p1 updates the variable twice before p2 reads it. Thus, when an execution correctly satisfies the data dependence among statements, if a data race exists this should be reported as it implies instability in the structure of the code leading to non-determinacy in the global state of program from one execution to another.

## 1.2 Models Examined

This work examines representative algorithms that attempt different methods of extracting data parallelism from code at compilation. The methods share the common paradigm of partitioning large data sets among processors but vary on how parallelism is realized in operating on the partitioned data; the variations include how the computation time is minimized by reducing the effects of communication latency from sending data between nodes and, how data dependence that requires access to off-processor data elements is handled. The four techniques discussed individually emphasize one of the following strategies:

1. Inter-processor transactions to communicate to a node all data used by the node in computations on the assigned subset of the iteration space; execution of the iterations at the local node starts only after receiving all data referenced in those iterations.
2. Communication by a node to request data used in the execution of a statement at the point of execution the statement.
3. Overlapping the communication for off-processor data referenced in an execution with the execution of iterations that reference only local data.

4. Using run-time supplied information to determine iterations that can be executed concurrently and providing synchronization for scheduling loop-carried dependent iterations.

These four algorithms are representative in that they cover the four basic strategies used to realize efficient parallel computation on message passing systems using the data-parallel paradigm. Indeed, there are only a finite number of ways that any given piece code may be executed in parallel and produce correct results. The data dependence between statements dictates the approach possible although different strategies for placing and communicating data between nodes can affect the underlying communication complexity. Our intent is to demonstrate through the examined models an approach that can be extended to analyze the behavior of any closely related technique.

# 2.0 Case I: Communication Before Execution (Iteration level)

## 2.1 Algorithm Description

In [20] Saltz et. al, discusses a method for attaining parallelization that involves the use of distribution and alignment directives which specify the mapping of global data onto processors in a MIMD computational environment named Parti. The mapping directive includes the use of arrays for specifying irregular distribution such as may occur in computations involving sparse arrays; these mapping arrays are also used when run-time methods of determining the parallelization paths (index set) of processors are intended. Their parallelization efforts center on computations over *do-loops*; do-loops that are to be parallelized are indicated to the compiler by a *distribute* clause at the start of the loop. An outline of the algorithm is shown in figure 1.

The algorithm has two primary phases, namely a preprocessing or ***inspector phase*** and an ***execution phase***. In the inspector phase the algorithm performs the following operations:

1. Use the distribute and partition directives to place data elements on processors. A *distributed-translation* table is built that uses static directives, such as block, cyclic, etc., to determine how data elements should be placed on the processor nodes. Run-time regular and irregular data placement is supported by using arrays that specify data mapping onto processors; runtime scheduling of loop iterations for parallel execution is specified via a similar array structure. For example consider the code shown below.
   In this code, S0 specify that array *partition* is to be distributed by

```
....
S0    distribute regular using block integer partition(n)
...
S2 distribute do i=1,n on partition
        .... code for do loop
```

Example 2

block [12] on the available processors. In S2, the loop iteration to be executed by a processor is determined by the mapping information in the array *partition*. Thus, every processor is capable of resolving the iterations that belong to its *iteration or index set* using the run-time supplied information.

2. Once data mapping is resolved, the system then determines which data elements in the distributed arrays are required at a node to carry out the defined computation. The location of a data element is resolved using the *distributed-translation* table. Subsequently, a set of inter processor communication is carried out; each processor is able to anticipate exactly which send and receive call it will need to execute for all inter-processor data communication to be correctly carried out. A hash cache associated with the distributed translation-table is used to record the off-processor fetches and stores; this allows for the recognition and single fetch of variables that may be

**Preprocessing phase** *(inspector )*:
Call procedure *build-translation-table* using the mapping
defined by array *partition* (This generates distributed translation table $T_{partition}$.)
Call procedure *dereference* to find processor assignments, PA, and local indices, LA,
for consecutive references to array elements. (Procedure *dereference* uses the
distributed translation table to find processor and memory locations of distributed
array data that have been mapped to processors using the array partition.)
Use setup hash table H to record off-processor elements

**Execution phase***(executor)*:
communication:
Use procedure *gather-exchanger* to find distributed data elements to
be transmitted and send/receive of elements. Write data to hash table

computation:
Use locally stored and off-processor data elements of distributed array
register in hash table to do computation.
Store: l
identify distributed elements to be stored off-processor ;
fetch value from hash cash and send to off-processor location

**fig. 1 : Algorithm outline**

referenced multiple times in the current computation.

The second phase of the algorithm is the ***executor***, where the established communication plan generated from the inspector phase is carried out, followed by the computation on the data. The communication process in the executor reads all off-processor distributed array data elements that are used in the execution of the loop iterations scheduled for the node and place those data elements in local storage (a hash table) for subsequent use. At the completion of the execution of the loop set at a node, data elements of arrays that have been written to, but owned by another node, are retrieved from the hash cache and sent to the appropriate off-processor locations. To illustrate, the code generated from this algorithm by an example (extracted from [20])

```
distribute regular using block integer partition(n)
distribute irregular using partition real *8x(4,n), y(4,n), f(4,4,maxcols,n)
distribute irregular using partition integer cols(9,n), nclos(n)
 ..... initialization of local variables ...
distribute do i=1,n on partition
        do j=1,cols(i)
          do k=1,4
                    sum = 0
                     do m=1,4
                              sum = sum + f(m.k.j.i)*x(m.cols(j,i))
                     end do
                      y(k,i) = y(k,i) + sum    .... 1
                   end do
            end do
         end do
```

**figure 2: Sparse Block Matrix Vector Multiply**

is shown in figure 2.

With compilation the code is decomposed into a SPMD type code (i.e., executed by each processor) shown in figure 3.

```
I. call gather-exchanger using schedule S to obtain off-processor elements of x
   gather-exchanger places gathered data in hash table H
     count = 1

II.  for all rows i assigned to processor P
        do j=1,ncols(i)
          do k=1,4
                sum = 0
   IIa. If  PA(count)==P then                  //PA is processor assignment array
                vx(1:4)=x(1:4,LA(count))      //LA  is the local  indices of each processor
          else
                Use PA(count), LA(count) to get vx(1:4) from hash table
       endif
         do m=1,4
                sum = sum + f(m,k,j,i,)*vx(m)
          end do
         IIb. y(k,i) = y(k,i) + sum
       end do
         count = count +1
  end do
```

**figure 3: Executor generated from ARF for sparse Block Matrix Vector Multiply**

## 2.2 Failure Modes

Step I of figure 3, identifies that all processors call routine gather-exchanger to obtain off-processor data elements. These data elements are placed in the local hash cache to be accessed when the *executor* references those data elements. Thus, there is a distinct data gathering stage followed by execution that uses the collected data. Further, the algorithm does not consider the data dependence that may exist across iteration boundaries of a processor's assigned index set; this omission may result in incorrect computation as the following discussion illustrates.

### 2.2.1 Flow dependence

Correct computation in the presence of flow (or true) loop-carried data dependence (e.g., y[k] = y[k-1] +...) requires that the processors actively collaborate during computation by waiting until variables on which a true dependence exists have been updated by the node responsible for doing so; after updating, the processor must then send the computed values to the dependent processor. An effective implementation of

strategy can result at best in a *skewing* or *tiling* [14] of the computation time among processors and a serialization of the total computation at the worse.

For example, assuming a loop-carried data dependence then, if statement IIb is changed to

```
y(k,i)= y[k,i-1] + y[k,i] + sum
```

then, the processor that assigns y[k,i] should proceed only after the processor that assigns y[k,i-1] has done so and sent the computed value to the dependent processor. No provision is made to guarantee this synchronization in the algorithm.

### 2.2.2 Output dependence

An output data dependency exists if a variable is updated multiple times in an iteration or on different iterations (loop carried) and read in iterations other than those in which it was updated. For example, consider a statement from an iterative structure such as,

```
y(b(i)) = ...
```

then a loop-carried output dependence exists if there exists some b(i) = b(j) for i not equal j.

If the indexing array, i.e., array b, is partitioned among processors without assuring that all repeated values of b (say b(i)= b(j) = ... =k ) are given to the same processor then, the final value of the array element with that index (y(k) = ... ) will be indeterminate; this is so as different processors will write to the same location in y with possibly differing values. This scenario is a possible as the algorithm allows for multiple nodes to compute new values for in a given data element and then send those values to the owner node for that element. This condition defines the presence of a *data race* if the order in which messages are received at the storage node is non-deterministic, i.e., there is no mechanism for enforcing that data from remote nodes arrive at the owner node in a defined order. The algorithm does not present such a mechanism. The presence of data races leads to non-determinism in an execution and may consequently lead to the generation of incorrect results.

### 2.2.3 Anti-dependence

In the presence of loop-carried anti-dependencies only, this algorithm will compute correctly, since by definition, a loop-carried anti-dependence on a variable x, requires that the dependent processor obtain the value of x before it is updated in the current loop.

**Observation I**: The strategy as used by this algorithm involves *preceding the computation over a given set of iterations at a node with the collecting of data used in those iterations without consideration to loop-carried computational dependence; this results in the algorithm failing to compute the results correctly on any code that involves a true or output dependence that extends across the iteration boundaries of a processor.*

**Observation II:** Without *a mechanism to enforce an order in the messages among processors, data races will occur when a value that has been updated at multiple nodes is stored at the designated off-processor location.*

## 2.3 Run-time Error Detection

We have shown that the algorithm generates incorrect code with inputs that have loop-carried flow or output data dependence. A method for detecting such violation at run-time by instrumenting the system is now discussed.

Each processor is assigned a subset of the iteration space; an execution is carried out using the given subset, in conjunction with the distribution specified, to place and locate data. A read and write access history is associated with each data element in the distributed arrays. As off-processor data is retrieved before the execution of the loop iterations, it is necessary that the read access history of those elements read by the *gather-exchanger* be updated during this phase of the execution. Further, as off-processor data updated by a node is written back to the parent node the write access history of such data elements much be checked by the parent to see whether the values sent by other nodes are inconsistent; if the values differ, then an error is reported as the messages are racing in that the outcome of the updating process is determined only by the order in which update-messages are received at the parent node. If the update values are consistent, then one could choose not to report this; however, it

could be significant to the user that a data race exists in the updating process. This follows as another execution of the program with a different set of input data may result in different update-values being sent, hence leading to a random final value in that location. Thus a user should always be made aware of data race and whether the updating process leads to a random state in the current execution.

```
check( var, pid, access,new_value){
 if (access == READ) {
        for each entry in var->write_history {
                                        if (var->write.pid > pid) then "report error"
                                          }
                      insert pid in read_history
                     }
if(access == WRITE) {
        for each entry in var->read_history {
                    if (var->read.pid > pid) then "report error"
                      }
        for each entry in var->write_history {
                    if(var->write.pid !=pid){
                                if(var->value != new_value) report "data race with inconsistent value"
                                      else "report data race"
                                  }
                               }
                insert pid  in write_history
              }
}
```

**figure: 4**

Without loss of generality, assume the index variable of iterations is increasing and that the iterations of the loop are partitioned among processors such that processor id (pid) increases with increase in the range of the index set associated with the processors; i.e., a processor with pid of 1 would be assigned iterations in a lower range of the iteration space than that assigned to a processor with pid of 2. This implies that, if a processor with a higher pid reads a variable and a processor with lower pid writes it subsequently, then a loop-carried flow/output data dependence would have occurred. Also, when a variable is read by a processor with lower pid after it has been written to by a processor with higher pid, then an incorrect execution has occurred.

As the access history of a variable can be reset at the end of a given parallel loop, we consider only the management of the access history within a loop. The pseudo

algorithm, ***check***, shown in figure 4 shows how the access history of a variable is managed to assess when an error has occurred.

# 3.0 CASE II: Communication Before Execution (Statement level)

## 3.1 Algorithm Description

In [1] Kennedy et al, describe at system for compiling programs for execution in a distributed memory multi-processor environment. They identify two areas that have hindered the utilization of the distributed-memory environment that they seek to address. The two areas of concern identified are:

1. The communication complexity associated with using distributed-memory computers for solving certain problems will exceed the time complexity, thereby making it not efficient to solve such problems in a distributed-memory environment. A programmer may be reluctant to invest the time to code for this environment in the face of such uncertainty.

2. The degree of difficulty associated with programming distributed-memory environment is higher that of programming the tightly coupled shared-memory systems. This has further discouraged the use of this environment for solving computationally intensive problems.

Concern 2 is identified as due largely to the absence of language support tools that make it easier for programmers to use the particular environment, a factor that the paper seeks to address directly through language augmentation and directives that transfer responsibility from the programmer to the compiler for generating SPMD (Single Program Multiple Data) code for distributed execution. They seek to address concern 1, by providing a paradigm that attempts to minimize the communication required to solve a computation problem; they provide distribution directives that a user may utilize to pass hints to the compiler for maximizing data locality with computation, i.e., the data is placed at the node that will reference it the most in the computation. The approach is similar to that used in [16] with variations on how ownership and distribution of variables is specified but similar in that inter processor communication

for off-processor variables occur only at the point that those variables are referenced during a computation

## 3.2 Implementation Structures

The algorithm incorporates the use of distribution directives to specify the mappings of data from input arrays onto processors. A *distribute* statement is defined, and provides a method of specifying a local function that identifies the processor at which elements of a shared array are stored. A *decompose* statement is also provided, and defines a virtual array, the members of which are elements of an underlying real array. These two functions are used by the compiler to decide the memory allocation needed at nodes for storing the data elements of shared arrays that will reside at those nodes. The execution algorithm makes no assumption on the location of data at any given instant but rather resolves each reference at the time of use. Through the *distribute/decompose* statements the compiler generates code that allows each processor to resolve the variables it needs for performing a computation and to determine where those variables are located. Once a processor acquires an array element that belongs to it, the array element is stored in the allocated space; if there are variables used in the current computation that are owned by another processor, that data is requested from the owning processor.

In this distributed computation model, every data element is assigned an *address* that is an ordered pair. The first component identifies a processor and the second component identifies an address in the local memory space of the processor. The functions $\delta$ and $\alpha$ are used to refer to these components individually; $\delta$ returns the processor identifier of the processor that contains the selected data element and $\alpha$ returns the location of a particular data element in the local memory of the processor that holds it. Some data elements may also be defined as floating or replicated as is done for arrays for which no distribute or decompose specification is provided.

The central task of the compiler is to separate the movement of data from the computation of new results. Two statements are formulated that explicitly handle this concern, namely **LOAD** and **STORE**. LOAD moves the values stored in a data element into a specified memory location and processor. STORE assigns the value of a local computation to a distributed variable. The algorithm for LOAD and STORE are shown in

```
.LOAD( MI,t,pid){
   INPUTS:        pid = processor on which variable should reside
                        MI  = reference to original variable
                        t   =  compiler generated local variable

if δ(MI) = thisproc then                    // if variable belongs to this proc
      if pid = thisproc
               then t ← δ(MI)                          //put  it in a local var. 't'
        else if pid ≠ 0
               then SEND(DEST = pid) δ(MI)      // send it to owner
         else do
                     t ← δ(MI)                                  // if owned by all, copy it and
              GSEND t                          // broadcast element

else if δ(MI) ≠ 0 then
         if pid = thisproc or pid =0                    //but pid is mine then
          then RECV(ONLYSRC = δ(MI))          // expect from source
else if pid = thisproc or pid =0                    //else element is floating
       then  t ← δ(MI)
        }

STORE(MO, f(t1, ..... tk){
       INPUTS:      MO =  reference to original variable
                      t1,...tk = compiler generated variables with loaded values
                           f()  = a function operating on      t1,...tk

if  δ(MO) = thisproc or δ(MO) = 0                   //if var is mine or floating
     then  α(MO) ← f(t1, ..... tk)                  //do computation and write to it
}
```

**Figure 5. Load /Store algorithm used in [1]**

figure 5. The LOAD statement reads a value using the $\delta$ and $\alpha$ functions. If the variable is located off-processor, it is read and placed in a temporary local variable. The computation is then carried out using the local version of the variable. The STORE statement is used to place the results of a computation into distributed memory locations.

Program execution consists of two phases. A data collection-transmission phase, where variables used in the pending computation are loaded into local variables followed by an execution phase where the specified computation is carried out using the locally stored variables.

For Example, the statement

B(I) = A(PI(I))

is compiled to

```
δ(T) ← 0                    //assume variable is replicated
LOAD PI(I), t1, δ(T)        //use load statement to locate and load PI(I) into t1
 STORE T = t1               // store t1 in T
 LOAD A(T), t1, δ(B(I))     //use load statement to locate and load PI(T) into t1
STORE B(I) = t1             //store t1 into B(I)
```

assuming the index I is already replicated to the processors.

## 3.3 FAILURE MODES

This algorithm is similar to that discussed case I: communication precedes computation without consideration to the dependence that may exist in the problem structure. It differs from the former algorithm in the technique used for identifying variable location and the absence of the strategy to exclusively precede computation for a given set of iterations with the necessary communication; here, communication is demand driven occurs at the statement level within an iteration.

### 3.3.1 Flow Data Dependence

Applied to an iterative computation with loop carried dependencies this model will fail to compute the correct result if there exists any true dependence that requires cross-processor references. The algorithm does not vectorize the process of gathering data for computation, as in case I discussed above does. Instead, communication via the LOAD routine, occurs for each data element that is off-processor and needed to be used in the pending computation. This can be viewed as a localized (about the computation point) communication before computation without consideration to dependence and, as such, the algorithm could generate erroneous results in the presence of true data dependencies that extend across processor boundaries.

Consider the example below:

B(I)= B(I-1) + ....

where B is an array partitioned on a distribute/decompose directive. At the boundary case where B(I-1) is owned by another processor, the LOAD statement should retrieve the new B(I-1) as assigned by the owning processor. However, the algorithm indicates no mechanism to ensure that the processor that owns the variable sent it after it has written to it or, that the node that reads it sees the updated variable.

It is possible for the algorithm to *dead-lock* here. The distributed memory information stored on a variable specifies (a) the processor that owns the variable (this is 0 for variables with unspecified distribution in which case, the variable is replicated) and (b) the address of the variable in the local memory of the processor that owns it. If a distribution is specified for array B, then all elements of B will be owned by a particular processor. An examination of the load algorithm shows that at the boundary case, where (I-1) or (I+1) references to an element that is off processor, the load algorithm request the element from the owner processor. The owning processor however, have no way of knowing that it should anticipate this request and consequently the processor will not respond. The algorithm allows processors to assume ownership of sections of shared data structure without facility to anticipate redistribution of the data elements during computation. Consequently, processors that reference an element owned by another processor will block at that point and the computation deadlocks. *Thus this algorithm will manifest deadlock in the presence of iterations that have loop-carried dependence that requires inter-processor communication during execution.*

If we assume however, that there is a mechanism to read a memory location at another processor, then dead-lock will not occur. However, the execution will have data races as there will be no enforced order between the reading of off-processor data and the writing of those elements at the owner locations while the owner is executing its iteration set.

### 3.3.2 Output and Anti-dependence

Again, assuming there is a method to read data located at another processor, then output and anti-dependence will have data races associated the execution fulfilling

those dependencies. This is so owing to the demand driven communication structure used in the computation, i.e., to request a variable only when it is need in a computation; thus the variable may be read before or after it has been updated at the parent node. Consequently, the results of the execution will be in-determinate for situations that cross-processor loop-carried data dependencies exist.

## 3.4 Run-time Error Detection

The spontaneous and unpredicted generation of messages between processors without a mechanism for enforcing message ordering, maps the execution into the general category of message passing programs that generic debuggers for message passing systems may be used.

Non-determinacy in an execution is indicated when messages between programs race. By messages racing, we mean that the observed order between message sends/receives could have occurred in another manner and thus affect the outcome of the computation. For example, consider the following possible executions in this environment:

```
doall i=2,4
      a(i) = a(i-1) ;
end do
```

Possible trace of the messages among 3 processors in two executions, with each processor executing an iteration, is as follows:

Execution Trace 1:

| execution at p1 | execution at p2 | execution at p3 |
|---|---|---|
| LOAD a(1) - read a(1)<br>read mess to read a(2)<br>send a(2)<br>execute a(2) = a(1); | LOAD a(2) - send message to read a(2)<br>receive mess to read a(3)<br>receive mess with a(2)<br>send a(3)<br>execute a(3) = a(2) | LOAD a(3) - send message to read a(3)<br>receive mess with a(3)<br>execute a(4)=a(3) |

(a)

Execution Trace 2

| execution at p1 | execution at p2 | execution at p3 |
|---|---|---|
| read a(1)<br>read mess to read a(2)<br>send a(2)<br>execute a(2) = a(1); | send message to read a(2)<br>receive mess with a(2)<br>execute a(3) = a(2)<br>receive mess to read a(3)<br>send a(3) | send message to read a(3)<br>receive mess with a(3)<br>execute a(4)=a(3) |

(b)

figure 6: Possible Traces of Executions

These executions show a race in the messages at p2; p2's reply message to the request to read a[3] could have occurred before or after the arrival of the response message to p2's message to read a[2] followed by the execution of code to change a[3]. Consequently, the final value of the elements of array **a** varies with the message ordering during the execution.

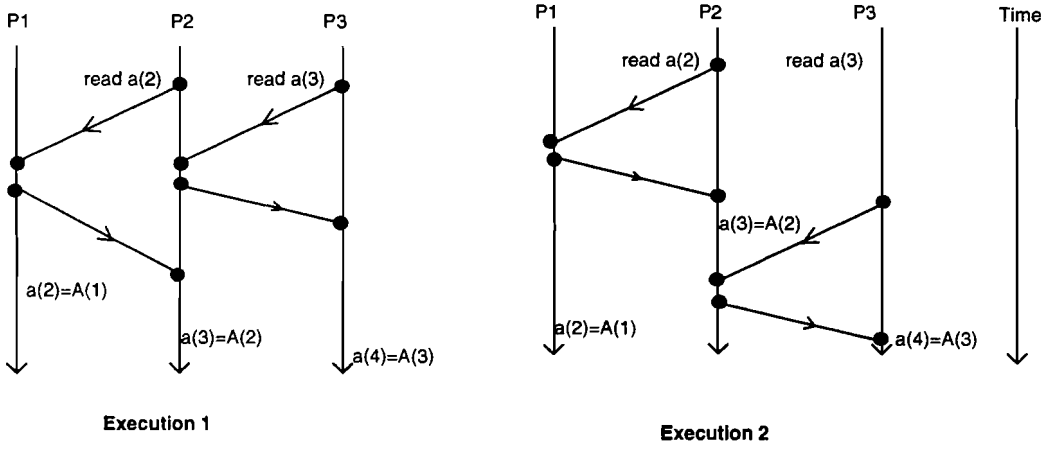


Figure 7: Partial Order Execution Graphs for traces

In [22] Netzer and Miller discuss a run-time method for detecting the presence of such races in message passing parallel programs; the technique applies to an execution in the parallelizing environment as discussed and, thus can be used to determine when races occur within programs generated for this system. A summary of the technique is presented below.

### 3.4.1 On-the-fly Race Detection and Tracing

An on-the-fly race detection algorithm is applied after each receive. The algorithm assumes that the receiving ends of communication channels are each associated with a single process and that messages can race only if they are received by the same process. After a message is received, this algorithm determines whether the message could have instead been received by a pervious operation in the same process. To identify these situations, an earlier receive is located that accepted a message from the same channel over which the current message was sent. Both the earlier message and the current message are treated as race candidates. If the previous receive did not happen before the current message, then a race exists: both the previous and current message could have been simultaneously in transit and either could have arrived first at the previous receive. If instead the previous receive happened before, then no race exists: the two messages could not have been simultaneously in transit, and no race exists. Implementation to of the algorithm involves the use of *vector time stamps* in each process that serve to encoded the happened before relations during an execution. A vector time stamp is a vector of length $p$ (the number of processes) containing event serial numbers[25] and the happened before relation [22] describes the temporal ordering between events. The user may refer to [22] for a detail description of the algorithm and the associated theory.

# 4.0 Case III: Overlapped Communication and Execution

## 4.1 Algorithm Description

In [4] Koelbel et al describe Kali, a FORTRAN augmented compiler that provides code for a software layer emulating a global name space on distributed memory architecture. In Kali, computation is specified via a set of parallel loops using the global name space as done on shared memory architecture; that is, the user is abstracted from the underlying model and presented with a view of a shared memory environment; It is the job of the Kali compiler to handle the generation of code to emulate the shared memory architecture via messages among the nodes of the distributed memory system.

The decomposition and distribution pattern for shared data structures are specified by Kali data mapping primitives similar to those used in High Performance FORTRAN (HPF) [12, 13] specifications. Once data distribution has occurred, computation is carried out in parallel with each processor executing the same code but operating on different index sets, a SPMD-style execution; *forall* program constructs are the main program structures for specifying concurrent execution. The index set given to each processor is derived using the data distribution specified by the user. Mapping functions or distribution directives are used determine how data should be partitioned to optimize on computation time through overlapping data transfer with computation. This reduces the effects of communication latency on computation time. An outline of the algorithm is shown in figure 6.

The algorithm is designed such that every processor has a knowledge of the data required by another node for it to do the required computation. To take advantage of communication latency between processors, the algorithm identifies iterations in the index set assigned to a processor that includes a reference to a non-local variable. It then identifies all iterations that reference local data.

At of the start of execution of a given *forall* loop, the processors use their knowledge of the variables needed at other nodes to initiate the transmission of that data to those

nodes. They then execute those iterations that require access only to local variables and subsequently use the data received from off-processor locations to compute those iterations that are dependent on those data elements. Thus the effect of communication latency on the computation time is minimized. Further, the generation of the index sets of a processor and the determination of loops that may be executed with local references only, can be determined at compile time if the compiler has adequate knowledge to do so. Otherwise, run-time techniques are used when the compiler needs added information, such as when the loop index variable has one or more levels of indirection.

The semantics of the *forall* loop used are "copy-in-copy-out," in the sense that values on the right hand side of the assignment are the old values in the array being updated in the loop. Thus the array assigned in a computation is effectively "copied into" each invocation of the forall loop and then the changes are "copied out." This may be expressed as having {**A**(i) = ...} transformed to { **A_new**(i) = ... } and the new values of **A_new** copied to array **A** at the completion of the loop.

*generate index set for local processor*
*generate index of remote variables referenced by this processor*
*generate list of variables local variables used by remote processors*
*generate list of variables to be expected from remote processors*

*send local variables needed by remote processors*

*do local iterations*

*receive messages from other processors*

*do non-local iterations*

**figure 8: Kali model for distributed memory computation**

## 4.2 Failure Modes.

### 4.2.1 Anti dependence

The algorithm will generate correct results in a program that uses only anti-dependence. For example, consider the following taken from paper [4].

```
forall  i in 1 .. N on A[i].loc do
   A(i):= A(i+1)
end;
```

where program fragment **on A(i).loc** causes the *i*th loop invocation to be executed on the processor owning the *i*th element of array **A**. With the loop shown, the algorithm will generate correct results as the off-processor elements of **A** needed by a node will be sent to the node before **A** is updated and, as the assignment is changed from **A**(i) = **A**(i+1) to **A_new**(i) = **A**(i+1). Thus, loop iterations can execute in any order without data dependence violation occurring.

### 4.2.2 Output and Flow dependency

A fundamental assumption used by the partitioning algorithm in generating the index set is, there exists no loop-carried output and flow data dependency between the variables assigned in the forall construct. Index sets that determine the work done by a processor are generated strictly with the intent of maximizing computation while minimizing the effects of communication time. Thus, lower index iterations in the index set of a processor could be computed after iterations with higher index values if the iterations of lower index values depended on off-processor data elements. Consequently, whether the index sets of the processors are such that only local loop-carried dependencies exist, the algorithm does not guarantee correctness as the execution of the iterations can be re-ordered.

## 4.3 Run-time Failure Detection

The communication plan established among the processors is well defined. The access pattern to variables used in a computation is also ordered -- all variables are sent before they are updated in the current iteration and the owners compute rule is applied. This well-defined structure in the communication amongst processors suggests that the execution will be race free and consequently computed results are deterministic.

However, if a programmer were to mistakenly enter

```
forall i in 1 .. N on A[i].loc do
  s1:   A(i) := A(i-1)
end;
```

instead of

```
forall i in 1 .. N on A[i].loc do
  s1:   A(i) := A(i+1)
end;
```

then a flow data dependence will occur at *s1* in the execution, but may not be noticed in a debugged session by examining the results of the computation. To detect a loop-carried flow and output data dependence violation at run-time the following approach is suggested.

Assume, without loss of generality, that the iterations within the loop are increasing and that the processor ids increase with the range of iterations assigned to a processor, i.e., processor with id 1 would execute lower iterations than a processor with id 2. As taken from [4], each processor has associated with it the following set of information, generated by the partitioning algorithm.

| | |
|---|---|
| exec(p) | := the list of iterations executed by processor p |
| local(p) | := subset of exec(p) that references only variables local to p |
| exec(p) - local(p) | := set of iterations of p that reference off processor elements |
| ref(p) | := the list of variables referenced by proc p executing iter exec(p) |
| in(p,q) | := set of elements received by p from q, and |
| out(p,q) | := the set of elements sent from p to q. |

Associate with each variable within the iteration space a read and a write access history. As explicit information is available on the iterations that a variable is read and written, the access history contains only the iteration number in which the variable was

```
generate index set for local processor : exec(p)
generate index of remote variables reference by this processor: ref(p)
generate list of variables local variables used by remote processors : out(p,q)
generate list of variables to be expected from remote processors: in(p,q)

send local variables needed by remote processors : out(p,q)
>>  for each variable, i, in out(p,q) do
>>     iter = reference of i in q iteration
>>          check(i, iteration , read)

do local iterations : local(p)
 >>    for each variable read/written ,i, in iteration j, do
 >>          check( i, j, read/write);

receive messages from other processors: in(p,q)

do non-local iterations: exec(p) - local(p)
>>   for each variable read/written ,i, in iteration j, do
>>         check( i, j, read/write);

   ------------------------------------------------------------------------------------------------------

check(var, iter, mode){
  if( mode == write)do
       if var read in higher iteration then report 'violation'
       insert in i write history iter;

else if (mode==read )do
      if i written in higher iteration, then report 'violation'
       insert in i read history, iteration;
}
```

**figure 9: Kali algorithm modified for checking data dependence violation**

accessed along with the mode of access. The iteration number specifies the iteration order that an execution should follow and by using this information a conflicting access to data arising from an out-of-sequence iteration execution is detectable. The code for detecting such data dependence violation is merged with the algorithm as shown in figure 9 below.

The routine ***check*** manages the access history and checks for accesses that indicate data dependence violation. A flow or output data dependence violation is reported at the node that defines the variable as this is where conflicting accesses occur. As the model is data race free, the only perturbation that this analysis incurs is an extension in the computation time; the outcome of the computation is unaffected.

The approach to merge the access history management and checking routines with the algorithm is important for the following reason. If an independent debugger were applied to detect access violation it would see any conflicting access as the Kali environment substitutes assignments to variables with assignments temporary variables; this is a side-effect of the "copy-in-copy-out" rule used by the environment. Consequently, for run-time detection of errors to work, when a write to a variable is performed within an iteration, the write history of the variable represented by that temporary should be updated and the access history checked for conflicts, a procedure that requires support from the Kali environment in identifying the mapping between temporary variables and variables they represent.

# 5.0 Case IV: Loop Scheduling

## 5.1 Algorithm Description

In [5] Saltz et al present an algorithm for performing static and run-time parallelization of do loops on message passing systems. Their optimizations for attaining parallel execution are specifically targeted toward loops having array references made through a level of indirection. This approach is important as it has been shown [9] that array references that involve indirection are a major reason for parallelization failure of loops by compilers. They discuss an implementation of this system in [21] for shared memory systems; variants of this approach are discussed in [ 7, 8,10]. Two basic principles are utilized in the algorithm, namely

1. A schedule of the loops that can be executed in parallel is formulated from an examination of the dependency information obtained from the array that serves as the indirection function. A topological sort on this array is used to identify those iterations that may be scheduled in parallel, i.e., iterations between which no loop-carried data dependence exists.

2. An *inspector* is used to identify variables referenced in the execution of the loop; This information is later used to generate send and receive messages for passing data between nodes on execution of a loop schedule. Control of execution is then passed to an *executor* that performs the computation with the acquired data.

To illustrate the operation of the system, consider the following example shown in figure 10(a) below (taken from [5]). Assume the outer loop S1 has to be executed in a sequential fashion. Sets of iterations of S1 (figure 10(a)) that can be executed concurrently are identified by performing a topological sort on the dependence graph relating the left hand side of S2 to the right hand side. This sort is performed by examining the integer array *column*. In this way, the sequential construct in Figure

10(a) is transformed into a parallel construct consisting of a sequence of parallel do loops. Each parallel do loop represents a concurrently executable set of indices from S1 of figure 10(a).

In figure 10(b), the inspector resolves the list of variables referenced in a schedule of concurrently executed iterations and uses that information to generate messages amongst the nodes to transfer those data elements. It is assumed that data has been distributed on nodes using the distribution directives discussed in previous cases. Thus parallel execution proceeds as a series of waves of concurrent execution.

```
S1 do i = 1, n**2
       do j = low(i), high(i)
S2          x(i) = x(i) + a(j)*x(column(j))
        end do
      end do
```

**figure 10(a): Sparse mesh Jacobi**

```
S1  do phase =1, num_phases

S2  doall pe = 1, num_processors
S3     do j = 1, npoints(phase, pe)
S4           next = schedule(phase, pe, j)
        do k = low(next), high(next)
 S5          x(next) = x(next) + a(k)*x(column(k))
        end do
      end do
    end doall
 end do
```

**figure 10(b): Transformed Sparse mesh**

## 5.2 Failure Modes

### 5.2.1 Flow Dependence

This algorithm correctly handles all loop-carried data dependence types when complete information on the data dependence within the loop has been captured in the indexing array that is used to determine independent iterations. Thus, for the example shown in figure 10(a), in statement S2 the array *column* is adequate to represent the dependency graph of the loop as all other references in the loop have a dependence distance of zero with respect to the inner loop.

However, if any of the variables referenced incurred a flow dependence with non-zero dependence distance with respect to the inner loop, then array *column* is inadequate to specify the data dependence between the iterations. Therefore, if statement S2 were changed to

```
S2        x(j) = x(j-1) + a(j)*x(column(j))
```

then for correct execution to occur, the flow dependence now present in the problem has to be considered with array *column* to determine the possible parallel schedule of loops. Thus *column* may indicate that loops say, *j* equal to 2,3 and 4, are independent and schedule them for concurrent execution, when a flow dependence exists between those iterations.

### 5.2.2 Anti dependence

Similarly, if an anti dependence was introduced outside the knowledge of array *column*, then execution could execute erroneously. Assume for example that statement S2 was changed to,

```
S2        x(j) = x(j+1) + a(j)*x(column(j))
```

As above, if *column* indicated that loops say, *j* equal to 2, 4 and 6, were independent and schedule them for concurrent execution and if, loops with iteration index of say, *j* equal to 3, 5 and 7, were then executed afterwards, the values of *x(j+1)* read in these executions would be incorrect.

## 5.3 Run-time Error Detection:

The parallel execution produced by this system is race free for the following reasons, (1) the inspector carries out all inter processor communication prior to executing a set of loop iterations -- that is all reads occur before writes (2) further, each node executes a single iteration in the set of concurrent do loops and thus variables are updated in a race free manner. The results of an execution are therefore deterministic.

The behavior of the system is similar to that of case III, in that loop iterations may be executed out of sequence. Thus, to detect a data dependence violation in an access, the iteration in which the variable was referenced and the nature of the reference, must

be kept in its access history; checks must be made at each access to ascertain whether an access anomaly has occurred. As in case III, a data dependence violation, or access anomaly is observed to have occurred if any of the following conditions occurs: (1) a variable is written in a higher iteration and then read in a lower iteration and (2) a variable is read in a higher iteration and then updated in a lower iteration.

# 6.0 Summary

The table in figure 11, summarizes the failure modes observed in the algorithms discussed. Some conditions that trigger these failures can be detected by compile-time (static) data dependence analysis. When a parallelizing compiler is unable to assess the data dependence structure within a source code, a common default is to abandon the parallelization effort or to force the process at the request of the user. The discussions in this paper presume that a compiler using the algorithms discussed does generate the intended parallel code as the techniques assume that the user has supplied code that meets the assumptions on which the technique is designed.

| **Case** | **Parallelizing Strategy** | **Failure Cause** (loop carried dependence) | **Exhibit Data Race** |
|---|---|---|---|
| I | Exclusive Communication Before iteration set Computation | Flow and Output data dependence | yes |
| II | Communication at statement execution (Demand driven) | All data dependencies | yes |
| III | Overlapped Communication with Computation | All data dependencies | no |
| IV | Loop scheduling | All data dependencies | no |

**figure 11: Summary of Failure Modes**

The presence of loop-carried data dependence (particularly flow and output) restricts the degree of parallelism realizable in the execution of a program. As such, each technique makes assumptions on the type of data dependence that exists in the problem structure and then focuses effort on the placement and access of data among the compute nodes for producing an execution that requires minimum time. Little attention is given to the need to produce complex synchronization actions between processors for handling problems with flow and output data dependencies. This is understandable as the approaches emphasize the construction of pre-planned communication that allows processors to anticipate messages from other nodes as well as the messages that need to be sent; where this anticipatory scheduling is not used,

there is an assumption that data dependence in the problem is resolved without access to off-processor data locations; this is possible if the distribution and mapping directives placed the data elements at nodes such that references are correctly resolved through local access only.

In [9], Yew et al, have shown that in the incidence of common nest loops with determinable dependence distance, 11% of array references have zero dependence distance, i.e., the dependence does not extend across the iteration and such loops are parallelizable without regard to data dependence. This low incidence of zero loop-carried data dependence in user code, implies a proportionate level of parallelizing success in algorithms that fail due to the presence of loop-carried data dependence. The algorithms also have an underlying assumption that there is enough regularity in the problem structure that allows the cost of communication to be bounded by the cost of computation, thereby leading to speedup in program execution. This assumption requires that the index of arrays referenced in the computation, be largely linear. Again, in [9], it is shown that 53% of all array subscripts are linear; this implies there are sufficient occurrences of regularity in problems for the efficient use of the mapping and alignment directives to partition data as done in the distributed memory data-parallel model, thereby minimizing communication overheads during computation. Further, the primary case of non-linearity is the presence of unknown variables (at compile time) in arrays subscript functions; parallelizing code with this type of data dependency requires run-time methods as discussed in case IV.

The variations in the techniques used requires that different run-time methods be applied to detect data dependence violations. For all cases except case II, it was shown that by augmenting the system with code for managing the access history of shared variables, it is possible to use simple checks on the history to determine incidence of data dependence violation. In case II, it was shown that by using an established technique that check for races in messages between processes, potential conditions for data dependency violation could be detected. Is was also discussed that, the presence of a data race in an execution undermines confidence in the results generated from such a computation.

Finally, we observe that an approach to guarantee correct parallel execution for a larger body of problems may need to use run-time information acquired from uniprocessor execution or information supplied by the user that serve as hints to compiler to

improve the possibility of parallelization; also, a general purpose system used for identifying and extracting larger amounts of data parallelism for execution on a distributed memory system, could utilize combinations of the algorithms as is the case in [3, 12].

## *References*


[1] D. Callahan, K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors", In *'The Journal of Super computing'*, 2:151-169 October 1988.

[2] K. Kennedy ,et al "High Performance FORTRAN Language Specifications" Ver 1.0 DRAFT, Jan. 25, 1990, Rice Univ.

[3] K. Kennedy ,et al, " Compiler Optimizations for FORTRAN D on MIMD Distributed-Memory machines", in *SuperComputing* 1991.

[4] C. Koelbel, et al " Supporting Shared Data Structures on Distributed Memory Architecture" in PPoFP ' 1990

[5] J. Saltz et al, "Run-Time Scheduling and Execution of loops on Message Passing Machines", in *Journal of Parallel and Distributed Computing* 8,303-312 (1990)

[6] C. Koelbel et al, "Compiling Global Name-Space Parallel Loops for Distributed Execution" in *IEEE Trans. on Parallel and Distributed Systems*, Vol. 2 no. 4 Oct,1991.

[7] Joel Saltz, Janet Wu, Seema Hiranandani, and Harry Berryman, " Runtime Compilation Methods for Multicomputers" in *1991 International Conference on Parallel Processing*, Vol. II page 26-30

[8] Peiyi Tang, Pen-Chung Yew, Chun-Qi Zhu, " Impact of Self-scheduling Order on Performance of Multiprocessor Systems", 1988 ACM

[9] Zhiyu Shen, Zhiyuan Li, Pen-Chung Yew, "An Empirical Study on Array Subscripts and Data Dependencies", in 1989 International Conference on Parallel Processing, Vol. II page 145-152

[10] Joel Saltz, Ravi Mirchandy and Kay Crowley, "Run-Time Parallelization and Scheduling of Loops", in IEEE Transactions on Computers, Vol. 40 No. 5, May 1991.

[11] Cherri M. Pancake, "A report from SuperComputing '92; Languages for High-Performance Computing: A Smorgasbord", in IEEE Parallel & Distributed Technology , February 1993

[12] High Performance FORTRAN Forum, "Draft High Performance FORTRAN Language Specification, Version 1.0", January 25 1993

[13] David B. Loveman, " High Performance FORTRAN", in IEEE Parallel & Distributed Technology , February 1993

[14] Saman P. Amarasinghe, Jenifer M. Anderson, Monica S. Lam, and Amy W. Lim " An Overview of a Compiler for Scalable Parallel Machines",

[15] Ravi Mirchandaney, Joel Saltz, Roger M. Smith, David M. Nicol, Kay Crowley, " Principles of Runtime Support for Parallel processors" In *'Proc. 1988 ACM International Conference on super computing'* , St. Malo France, July 1988.

[16] Michael Gerndt, "Updating Distributed Variables in Local Computations" in *CONCURRENCY: PRACTICE AND EXPERIENCE*, Vol. 2(3), 171-193 September 1990.

[17] Charles koelbel, Piyush Mehrotra "Compiler Transformations for Non-Local Memory Machines", in *Proc. 4th Conf. Supercomput.* Vol. 1, May 1989, pp. 390-397.

[18] G.R Andrews and F.B. Schenider. "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys* 15, 1, pp. 3 -43, March 1983

[19] Robert H. B. Netzer and Barton P. Miller, "What are Race Conditions? Some Issues and Formalizations." *ACM Letters on Programming Languages and Systems* 1, 1, March 1992

[20] Janet Wu, Joe Saltz , Seema Hiranandani, Harry Berryman, " Runtime Compilation Methods for Multicomputers", *1991 International Conference on Parallel Processing*, Vol II, pp. 26- 30

[21] Joel Saltz, Ravi Mirchandaney, " The PREPROCESSED DOACROSS Loop", *1991 International Conference on Parallel Processeing*, Vol. II pp. 174-179

[22] Robert H.B. Netzer and Barton P.Miller, "Optimal Tracing and Replay for Message-Passing Parallel Programs, " *Supercomputing* '92, Mineaplois

[23] Anne Dinning and Edith Schonberg, "An Evaluation of Monitoring Algorithms for Access Anomaly Detection," *Ultracomputer* Note #163, July 1989.

[24] A.H. Karp, "Programming for Parallelism.'", *Computer*, 20(5):43-57, 1987.

[25] C. J. Fidge, "Partial Orders for Parallel Debugging," *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 183-194 Madison, WI, January 1989.