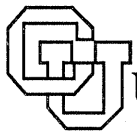
# ARRAY SECTION ANALYSIS FOR CONTROL PARALLEL PROGRAMS

Jeanne Ferrante, Dirk Grunwald,
Harini Srinivasan

CU-CS-684-93



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

# Array Section Analysis for Control Parallel Programs

Jeanne Ferrante
IBM Research Division,
T.J Watson Research Center, P.O Box 704,
Yorktown Heights, NY 10598

Dirk Grunwald and Harini Srinivasan
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430

CU-CS-684-93 November 1993

University of Colorado at Boulder

Technical Report CU-CS-684-93
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Jeanne Ferrante
IBM Research Division,
T.J Watson Research Center, P.O Box 704,
Yorktown Heights, NY 10598

Dirk Grunwald and Harini Srinivasan
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430

# Array Section Analysis for Control Parallel Programs

Jeanne Ferrante
IBM Research Division,
T.J. Watson Reasearch Center, PO Box 704,
Yorktown Heights NY 10598
(Email:ferrant@watson.ibm.com)

Dirk Grunwald and Harini Srinivasan
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430
(Email:{grunwald,harini}@cs.colorado.edu).

November 1993

**Abstract**

Data flow analysis has been used by compilers in diverse contexts, from optimization to register allocation. Traditional analysis of sequential programs has centered on scalar variables. More recently, several researchers have investigated analysis of *array sections* for optimizations on modern architectures. This information has been used to distribute data, optimize data movement and vectorize or parallelize programs. As multiprocessors become more common-place, we believe there will be considerable interest in explicitly parallel programming languages. These languages have additional control and synchronization structures. It is more difficult to optimize programs written in these languages than in traditional languages.

In this paper, we show how to compute array section data flow information for a class of parallel programs, generalizing and extending the work of others. To illustrate the use of this information, we show how this information can be applied in compile-time program partitioning. We use information about array accesses to improve estimates of program execution time used to direct runtime thread scheduling.

**Keywords:** flow analysis, parallel programs, array section analysis, parallel program partitioning.

## 1 Introduction

Parallel computer architectures are becoming more common as computational demands increase and these new machines become more cost effective. Programming such machines using sequential languages has had only limited success [14, 4]. It is recognized that to achieve good performance on such machines, new parallel algorithms need to be developed [43]. Explicitly parallel programming languages such PCF Fortran [15], IBM Parallel Fortran [28], CC++ [8, 9], Parallel SETL [27], Concurrent Pascal [23], The Force [29], and Occam [35] allow the direct expression of such algorithms, and so are finding increasingly wider use.

Common parallel control constructs found in a number of explicitly parallel programming languages [15, 28, 23, 27, 29] are *co-begin,co-end* and *parallel loops.* Co-begin, co-end parallelism allows multiple blocks to execute in parallel. These languages commonly contain synchronization primitives to coordinate processes such as event variables and `Post` and `Wait` primitives. Commonly, co-begin, co-end blocks execute in a data-independent manner, except where synchronization is used. Matching `Post` and `Wait` statements require that all shared variables in the waiting processor be made consistent with those in the posting processor. Some languages, such as The Force [29], use explicit *produce* and *consume* notation to indicate data flow. These "unstructured synchronization" complicates the analysis of parallel programs.

Existing sequential analysis techniques for arrays ([19]) were not designed for parallelism and synchronization. In [20], we demonstrated that traditional data flow analysis techniques for sequential programs can not be directly applied to parallel programs and have shown extensions to handle `cobegin/coend` parallelism and synchronization for scalar data flow analysis.

Recently, several researchers have investigated analysis of *array sections* for optimizations on modern architectures. In this paper, we extend our previous analysis to compute data flow information on *array sections* for explicitly parallel programs with `cobegin/coend` and `doall` parallelism. In particular:

- Gross and Steenkiste[19] have a general framework for forward and backward array section and scalar analysis using interval-based algorithms. We extend their framework to include explicitly parallel programs, and use this framework to compute Reaching Definitions and Live Definitions.
- Granston and Veidenbaum [18] extended [19] to analyze array sections in programs containing statically scheduled `doall` parallelism without synchronization. We generalize this analysis to include programs with co-begin, co-end parallelism, a limited form of event variable synchronization and doall loops.
- We show how dataflow information can yield better execution time estimates when partitioning explicitly parallel programs.

Figure 1 shows an example parallel program with `doall` and `cobegin/coend` parallelism and `post/wait` synchronization. The reaching definitions at the end of the program are $\{A_7(1:5), A_{12}(6:10), A_2(11:20)\}$, and the reaching definitions at the end of statement (12) are $\{A_7(1:j-1), A_2(j:5), A_{12}(6:j+5), A_2(j+6:20)\}$. It is necessary to incorporate the effect of parallelism and synchronization in the data flow framework to determine these reaching definitions.

In this paper, we extend the array analysis in [19] to handle a more general class of parallel programs - these extensions are similar to those in [20], but modified to handle arrays. We have implemented these algorithms using the SETL prototyping language for a simple explicitly parallel language. Figure 2 shows the reaching definitions computed by our implementation for the program in Figure 1. In §5, we show how this reaching definitions information can be combined with other dataflow information to indicate what data is shared by the different threads in the parallel program.

## 1.1 Related Work

Traditionally data flow frameworks have been used to analyze sequential programs and compute data flow information such as reaching definitions, live variables, available expression etc. Analysis of arrays

is very important for scientific and numerical programs. Array analysis for sequential programs is a well-studied topic. Data dependence analysis [47, 2] is used for vectorization, parallelization and various other program restructuring techniques. Gross and Steenkiste [19] present a global data flow analysis framework for arrays in sequential programs that results in a uniform treatment of arrays and scalars in the compiler. Such analyses techniques can be used for global optimizations of both scalars and arrays.

Duesterwald *et al* present a data flow framework for array reference analysis that exploits fine-grain parallelism [13]. Granston *et al* [18] compute reaching array sections information for parallel programs. They consider *doall* parallelism without synchronization. Maydan *et al* [12] used *Last Write Trees* to analyze nested loops for array dataflow information. Gupta and Schonberg [22] present a data flow framework for computing data availability information in programs based on the framework in [19]. None of this work considers control parallelism or explicit synchronization. Cytron *et al* [11] used data flow analysis of automatically parallelized sequential programs to implement compiler-directed caching. Although similar information is computed, our analysis is more difficult due to explicit parallel and synchronization constructs.

## 1.2 Application to Partitioning

Partitioning parallel programs at compile-time to a granularity that fits the particular execution architecture has been shown to be a successful approach to achieving useful parallelism [30, 34, 40, 5, 17, 33, 38, 16, 49, 48, 39]. Previous work such as [40, 48] assume execution time estimates of the nodes and communication cost estimates on the edges of a graph to aid in partitioning decisions; nodes with costly communication between them can be merged to decrease the communication cost and thus the overall execution time. These communication cost estimates can be obtained from analysis of sequential programs, as in PTRAN [1], by explicit representation in the input parallel language, as in Jade [37], or by execution profiling [38].

In this section, we show that array section analysis can provide accurate detailed information to aid partitioning. To our knowledge, no other previous work [30, 34, 40, 5, 17, 33, 38, 16, 49, 48, 39] considered the use of data flow information to aid partitioning.

Figure 3 shows two example scheduling scenarios for a co-begin, co-end parallel section with three independent threads (T1, T2, T3) that must be mapped on two processors (the Black processor and the Gray processor). Thread T1 computes values for the odd elements at the beginning of an array (*i.e.*, X(1:K:2)), while thread T3 computes values for the even elements at the end of the same array (*i.e.*, X(L:N:2), where L is even and L < K). There are no write conflicts because of the even/odd partitioning, but communication might occur on cache-based multiprocessors if cache lines are larger than a single word. Thread T2 reads and writes values of array 'Y'. Before execution of either schedule, both array 'X' and 'Y' reside only on the Gray processor.

We assume that each thread executes roughly the same number of instructions. Without the use of array section analysis, we would assume each entire array must be shared. If the compiler assumes the tasks execute for the same duration, information about the communication between tasks may greatly influence thread scheduling. Conventional scheduling algorithms do not consider the actual data and its interactions between the different threads. However, consider the effect of this data interaction with

```
(1)   doall 10 i = 1, 20
(2)      A(i) =
(3)   10 end doall
(4)   cobegin
(5)   do 20 k = 1, 5
(6)
(7)      A(k) = G(A(k + 10)); post(ev1)
(8)
(9)   20 end do
      ||
(10)  do 30 j = 1, 5
(11)
(12)     wait(ev1); A(j+5) = G(A(j))
(13)
(14)  30 end do
(15)  coend
(16)  print A(15)
(16)  END
```

**Figure 1:** Example Parallel Program

| Node | *SReachIn* | *SReachOut* |
|---|---|---|
| (1) | | |
| (2) | | $A_2(i)$ |
| (3) | $A_2(1:20)$ | $A_2(1:20)$ |
| (4) | $A_2(1:20)$ | $A_2(1:20)$ |
| (5) | $A_2(1:20)$ | $A_7(1:5), A_2(6:20)$ |
| (6) | $A_7(1:k-1), A_2(k:20)$ | $A_7(1:k-1), A_2(k:20)$ |
| (7) | $A_7(1:k-1), A_2(k:20)$ | $A_7(1:k), A_2(k+1:20)$ |
| (8) | $A_7(1:k), A_2(k+1:20)$ | $A_7(1:k), A_2(k+1:20)$ |
| (9) | $A_7(1:5), A_2(6:20)$ | $A_7(1:5), A_2(6:20)$ |
| (10) | $A_2(1:20)$ | $A_7(1:5), A_{12}(6:10), A_2(11:20)$ |
| (11) | $A_7$(1:j-1), $A_2$(j:5), $A_{12}$(6:j+4), $A_2$(j+5:20) | $A_7$(1:j-1), $A_2$(j:5), $A_{12}$(6:j+4), $A_2$(j+5:20) |
| (12) | $A_7$(1:j-1), $A_2$(j:5), $A_{12}$(6:j+4), $A_2$(j+5:20) | $A_7$(1:j-1), $A_2$(j:5), $A_{12}$(6:j+5), $A_2$(j+6:20) |
| (13) | $A_7$(1:j-1), $A_2$(j:5), $A_{12}$(6:j+5), $A_2$(j+6:20) | $A_7$(1:j-1), $A_2$(j:5), $A_{12}$(6:j+5), $A_2$(j+6:20) |
| (14) | $A_7$(1:5), $A_{12}$(6:10), $A_2$(11:20) | $A_7$(1:5), $A_{12}$(6:10), $A_2$(11:20) |
| (15) | $A_7(1:5), A_{12}(6:10), A_2(11:20)$ | $A_7(1:5), A_{12}(6:10), A_2(11:20)$ |
| (16) | $A_7(1:5), A_{12}(6:10), A_2(11:20)$ | $A_7(1:5), A_{12}(6:10), A_2(11:20)$ |

**Figure 2:** Results of Data Flow Analysis of Example Program

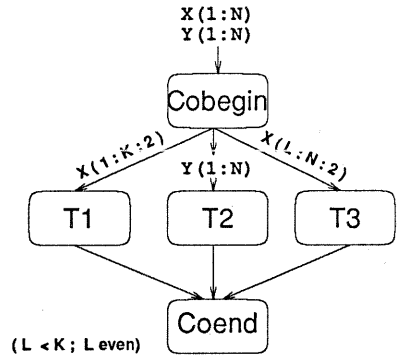
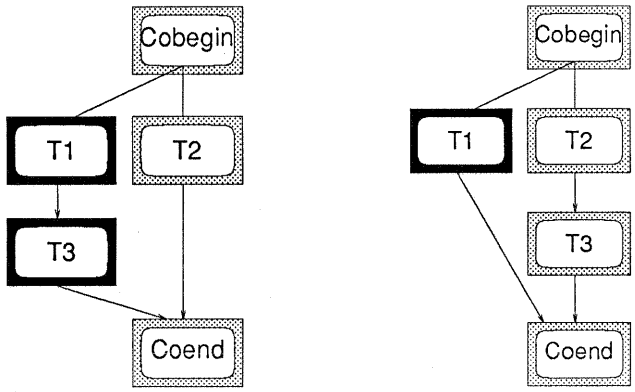



Diagram of Original Parallel Program Containing Three Threads




(A) First Scheduling Scenario (B) Second Scheduling Scenario

**Figure 3:** Sample Parallel Program Scheduling Decision

these two schedules on the Kendall Square Research KSR-1 [36]. The KSR-1 is a hierarchical, shared address space architecture using distributed caches.

Strong memory consistency is implemented using an *invalidation protocol*; before a write to variable $X$ is allowed to complete, all other readable copies of the cache line holding that variable are marked as invalid or removed from the cache of other processors. In schedule (A), shown in Figure 3, thread T1 executes on the Black processor, write-referencing all cache lines holding the array 'X'. This causes the Gray processor to invalidate portions of its copy of array 'X'. When the Black processor executes thread T3, no further communication occurs, because array 'X' has been invalidated on the Gray processor. In schedule (B), the Black processor starts executing thread T1, bringing array 'X' from the Gray processor. When the Gray processor begins executing thread T3, the newly updated contents of 'X' must be copied from the Black processor to the Gray processor. If T1 and T3 are executed concurrently on different processors, considerable *false sharing* [26] occurs, as the Black processor invalidates copies of data in the Gray processor and the Gray processor re-copies the data needed by T3.

Interprocessor cache references in the KSR1 are relatively expensive. Fetching (or invalidating) memory on the same ring of 32 processors takes $\approx$ 150 execution cycles; references to data on more

processors takes $\approx$ 450 execution cycles. The cache line size is 16 floating point values, so the Black processor would spend at least $\approx \lceil \frac{L-K}{16} \rceil \times 150$ machine cycles invalidating all other copies of the array 'X'. For a 128-element overlap, schedule (B) would cause the Gray processor to spend an additional $\approx$ 1200 machine cycles waiting for data from the Black processor. If thrashing occurs due to false sharing, this cost could be considerably greater. The relative importance of this additional delay depends on the amount of computation in T1, T2 & T3; scheduling algorithms need to know about both data flow and computation time.

At this level of architectural detail, the efficiency of scheduling this simple parallel program is greatly influenced by the variables used within each thread and the variables "passed" between synchronizing processors. For scalable, shared-address space architectures, such as the KSR1, the Convex SPP or the Cray T3D, knowing the actual section of data that is moved can tip scheduling decisions in an unexpected direction. Array section analysis is needed to determine the actual data sections shared by threads.

We use data flow analysis to determine the array sections communicated between nodes in a parallel flow graph. We use *reaching definitions* information to compute the *live definitions* at each node in the flow graph; these are definitions that may be needed in subsequent execution. This information is used to compute the *Send* and *Receive* sets, or the set of variables and array sections that may be transferred to or from another node. The *Send* and *Receive* sets can be used to guide scheduling decisions, as illustrated above.

In Section 2, we introduce the Parallel Flow Graph used in our work, discuss the representation of array sections, and give other necessary background. In Section 3, we discuss the interval-based algorithm to compute reaching array section definitions information in explicitly parallel programs. Section 4 discusses live-definitions analysis of array sections. Section 5 illustrates the use of reaching definitions and live definitions information in computing the *Send* and *Receive* sets at synchronization points in the program. Finally, in Section 6, we give some conclusions and directions for future work.

# 2 Background

We use the Parallel Flow Graph [44, 3, 7] to represent control flow and synchronization in parallel programs exhibiting `cobegin/coend` parallelism with `Post/Wait` synchronization. Nodes in this graph represent *extended basic blocks* with at most one `Post` statement and one `Wait` statement. We augment this graph with *doall* and *enddoall* nodes to represent `doall` loops. Edges represent parallel control flow, sequential control flow or synchronization - there is a synchronization edge from every `Post` statement to a corresponding `Wait` statement.

We use the *Bounded Regular Section Descriptor*(BRSD) [24] to represent a section of an array. This representation allows subsections of arrays *A(S)*, where $A$ is the name of the array and $S$ is a vector of subscript values such that each element is: (a) an expression of the form $a*k+b$, where $k$ is the induction variable (b) a triple, $l : u : s$ where $l$ is the lower bound, $u$ the upper bound and $s$ is the stride; or (c) $\perp$ indicating no knowledge of the subscript type. In this case, we make conservative assumptions, *e.g*, in the case of the reaching definitions problem, an array section is extended to include the entire array dimension. The representation of *definitions* of array sections includes the node in the PFG where the definition appears apart from the array name and subscript values. The different set operations, union,

intersection and subtraction, are defined on sets of array section descriptors [31]. For example, given the sets of array sections $S_1 = \{A(1:i), B(1:i-1)\}$ and $S_2 = \{A(i+1), B(1:20)\}$, the results of the various set operations assuming the index $i$ varies from 1 to 20 are: $(S_1 \cup S_2) = \{A(1:i+1), B(1:20)\}$, $(S_1 \cap S_2) = \{B(1:i-1)\}$, $(S_1 - S_2) = \{A(1:i)\}$.

We compute data flow information using Tarjan intervals [45]. Each loop is an interval entered by a unique interval header node. Intervals have unique entry (the header) and exit nodes. The outer most interval corresponds to the main program flow graph with interval entry and exit nodes equal to the program entry and exit nodes respectively.

Interval analysis proceeds in two phases: the *Elimination Phase* and the *Propagation Phase* [6]. In the elimination phase, data flow information is propagated from inner intervals to the top level. The data flow sets are computed for each interval individually assuming an initial value for the data flow information at the interval entry (exit) node for forward (backward) problems. Once an inner interval is analyzed, the data flow information at the exit of the interval (entry of the interval for backward problems) is summarized into a summary node, *i.e.*, the inner interval is logically collapsed. In the propagation phase, the data flow information is propagated from the top level (outer most interval) to inner most intervals.

# 3 Data Flow Analysis

In this section, we first explain how we identify intervals in a parallel program and the traversal order of the PFG to take into account synchronization. The rest of the section describes the interval-based data flow analysis methods for computing reaching definitions and live definitions information in programs that exhibit `cobegin/coend` parallelism and `post/wait` synchronization and `doall` parallelism. In §5 we show the *Send* and *Receive* sets that can be computed from this information.

## 3.1 Intervals and Traversal order of the PFG

Parallel programs with synchronization can exhibit loops with multiple entry or exit points as exemplified in Figure 1. Since we are only interested in identifying control flow loops as intervals and since synchronization edges do not carry control flow information, synchronization edges need not really play a role in computing intervals. We can therefore compute intervals from the *Control Flow Subgraph*(CFSG) of the PFG as long as this subgraph is reducible. The CFSG is a subgraph of the PFG that has the same nodes as in the PFG and only the control flow edges of the PFG; synchronization edges are not present in the CFSG. In addition, we also identify `doall` loops as intervals in the CFSG. Within intervals, for forward problems, we analyze nodes in topological sort order. The traversal order is reverse topological sort order for backward problems.

Although we compute intervals using the CFSG, the data flow analysis must consider the effect of synchronization since synchronization edges propagate data flow information. Analysis for forward and backward data flow problems in traditional interval-based methods considers one interval at a time. In our case, it is possible to have synchronization edges *between* intervals. We handle such a situation by traversing intervals related by synchronization edges such that every `post` node is traversed before the

corresponding `wait` node in a forward problem and every `wait` node is traversed before the corresponding `post` node in a backward problem[1].

**Theorem 1** *For deadlock free programs, the above traversal order results in* correct *data flow analysis as long as (a) the synchronization edge does not force the traversal of a node in an outer enclosing loop before an inner loop. and (b) there is no synchronization cycle[2] in the PFG.*

*Proof:* We will present a detailed proof of this theorem in a later paper; the intuition is as follows: In the case of a forward problem, situation (a) occurs when a `Post` node appears in an outer interval and the `Wait` node appears in an inner interval, but inner intervals must be traversed before outer intervals. The traversal order given above requires a `Post` node to be traversed before the corresponding `Wait` - a contradiction. The reasoning for backward problems is analogous. Situation (b) corresponds to a synchronization cycle and such a synchronization pattern calls for iterative analysis since data flow information may not be inferred in two passes through the program. We use a combination of interval-based and iterative approaches to handle such situations. This is similar to hybrid algorithms used to handle irreducible flow graphs [41, 42, 32].□.

In remainder of this paper, we assume programs with the property stated in the theorem. Figures 4 and 5 give the forward traversal orders for the program in Figure 1; Figures 6 and 7 illustrate the backward traversal orders. These figures show the Parallel Flow Graph of the example program - the numbers in parenthesis in each of these figures indicate the traversal order and the intervals corresponding to the loops are marked **I0**,**I1** and **I2**. Node (7) is the `Post` node and node (12) is the corresponding `Wait` node. Note that in the elimination phase, the intervals are logically collapsed into summary nodes and these summary nodes are traversed as part of the outer interval – the traversal numbers associated with **I0**, **I1** and **I2** corresponds to those of the respective summary nodes.

## 3.2 Reaching Definitions Analysis

A definition of an array section at node $n$ reaches a node $m$ in the Parallel Flow Graph if it has not been redefined on any path from $n$ to $m$ that must execute. We use the data flow sets given below in computing reaching array section definitions at any point in a parallel program. The data flow equations bear resemblance to the equations used in scalar analysis. However, the different set operations are on sets of array section definitions. In addition, we use interval analysis and additional data flow information to summarize information at interval entry and exit nodes. Interval analysis also simplifies the analysis of `doall` loops.

**Local Data Flow Sets:** We use the following local data flow sets:

*SGen*(**n**): downward exposed definitions of array sections that reach the end of the block.

*SKill*(**n**): definitions in other nodes that override the definition of an array section in $n$. *SKill*(n) is further subdivided into $SKill_s$(n) and $SKill_p$(n): $SKill_s$(n) corresponds to definitions in *SKill*(n)

[1] We assume that the loops in which such `post` and `wait` appear have conforming loop bounds and that loops are normalized.

[2] A synchronization cycle occurs when two loops that are in different parallel threads are related by synchronization edges and such edges force the two loops to execute in an interleaved fashion [7].

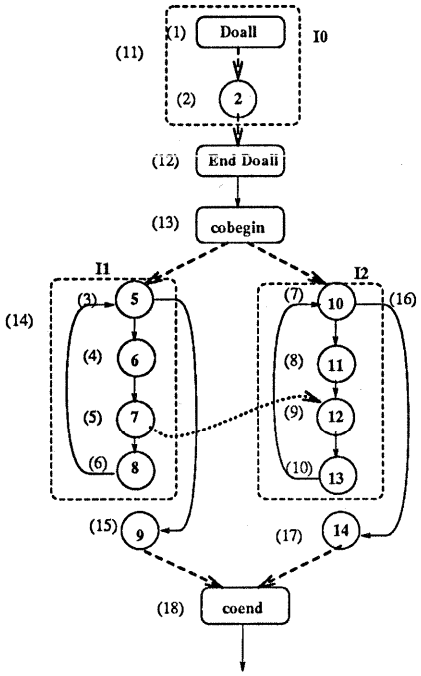



**Figure 4:** A traversal Order in Elimination phase for a forward data flow problem

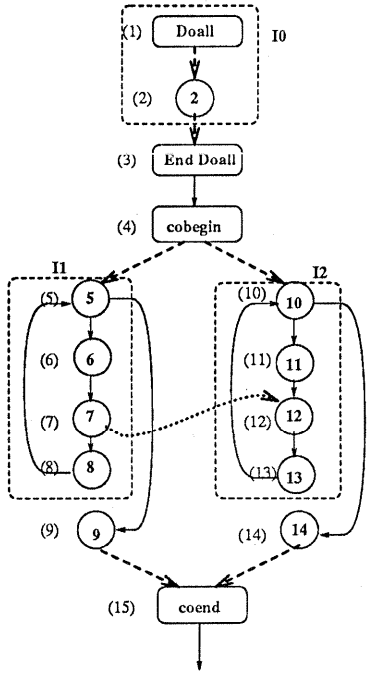



**Figure 5:** A traversal Order in Propagation phase for a forward data flow problem

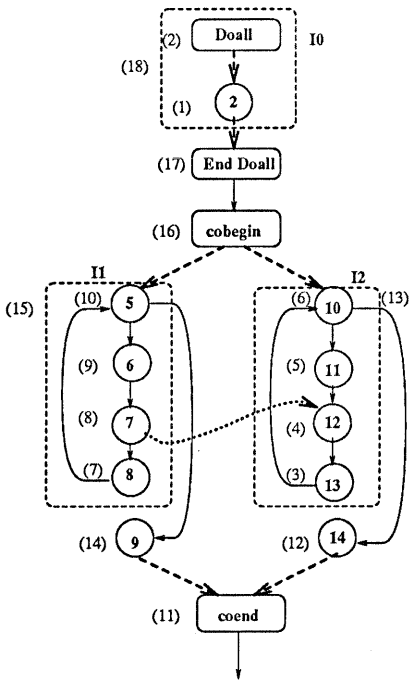



**Figure 6:** A traversal Order in Elimination phase for a backward data flow problem

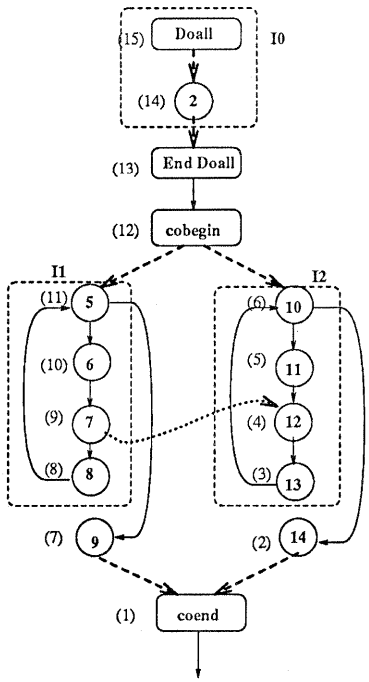



**Figure 7:** A traversal Order in Propagation phase for a backward data flow problem

that appear in the same sequential thread as $n$ and $SKill_p$(n) corresponds to such definitions that appear in threads different from that of $n$.

*DoesGen*(n): variables (not definitions) that *must* be defined in this node.

**Global Data Flow Sets**

*SSynch*(n): Definitions of array sections propagated to $n$ via synchronization edges.

$$SSynch(n) = \bigcap_{p \in Pred_{sy}} SReachOut(p) \cup \bigcup_{p \in Pred_p} SSynch(p) \cup \bigcap_{p \in Pred_s} SSynch(p) \cup \bigcup_{p \in (Pred_{sy} \cap Prec(n))} SReachOut(p) \quad (1)$$

$Pred_{sy}$, $Pred_p$ and $Pred_s$ refer to the synchronization, parallel and sequential predecessors respectively.

The definitions propagated to a waiting node from its synchronization predecessors are all those definitions that must reach the waiting node from other threads. Therefore, we make a conservative assumption that the confluence operator is intersection at sequential merge nodes and `Wait` nodes and union at parallel merge nodes. The confluence operator at a `Wait` node is union if we know the exact precedence information [21] between the `Wait` and the corresponding `Posts`.

*SAKillIn*(n): definitions of array sections that *must be* killed at the entry to $n$:

$$SAKillIn(n) = \bigcap_{p \in Pred_{sy}} SAKillOut(p) \cup \bigcap_{p \in Pred_s} SAKillOut(p) \cup \bigcup_{p \in Pred_p} SAKillOut(p) \quad (2)$$

The computation of *SAKillIn* is important to handle any definition $d$ that is killed in one sequential thread but not another and the two threads merge at a coend node: at such a parallel merge node, the definition $d$ must be omitted from the reaching definitions set since both the threads must always execute before coend. The presence of this set is also an artifact of our copy-in/copy-out semantics assumption as explained in [20]. We handle parallel merge nodes corresponding to *end-doall* nodes while summarizing information for such intervals.

*SReachIn*(n): definitions that reach the entry to $n$. This is computed as in the sequential case but excluding all those definitions that we know must be killed by the time $n$ executes.

$$SReachIn(n) = \bigcup_{p \in Pred} SReachOut(p) - SAKillIn(n) \quad (3)$$

*MustReachIn(n)*: variables (not definitions) that *must* reach the entry to $n$. Since we are interested in the variables that *must* reach the node, the confluence operator at sequential merge points is intersection. This is true since only one of the incoming edges execute at run time. Similarly, the confluence operator at a `wait` node is intersection since only one of the incoming synchronization edges must execute to activate the `wait` statement. However, since all the parallel flow edges execute, the confluence operator is union at parallel merge nodes. Since we are only interested in *variables* that must be defined by the time this node executes, there is no set analogous to the *SAKillIn* set that has to be subtracted.

$$MustReachIn(n) = \bigcap_{p \in Pred_s} MustReachOut(p) \cup \bigcap_{p \in Pred_{sy}} MustReachOut(p) \cup \bigcup_{p \in Pred_p} MustReachOut(p)$$

*SReachOut(n)*: definitions that reach the exit of $n$. This set is computed just as in the sequential case:

$$SReachOut(n) = (SReachIn(n) - SKill(n)) \cup SGen(n) \tag{5}$$

*SAKillOut(n)*: definitions that *must be* killed at the exit of $n$:

$$SAKillOut(n) = \begin{cases} SAKillIn(n) \cap SSynch(n) & \text{(cobegin)} \\ SAKillIn(n) \cup (SAKillIn(InFork(n)) - SReachOut(n)) & \text{(coend)} \\ (SAKillIn(n) \cup SKill_s(n) \cup (SKill_p(n) \cap SSynch(n))) - SGen(n) & \text{(otherwise)} \end{cases} \tag{6}$$

The first case in this equation represents all those definitions that are propagated to the current thread via synchronization edges and then later killed; the second case corresponds to the union of those definitions that are killed inside the current parallel block and those that are killed outside the parallel block and not defined in this parallel block; finally, the third case corresponds to all those definitions that are killed in the current node $n$. $SKill_p(n) \cap SSynch(n)$ are those definitions propagated via synchronization edges to $n$ and killed in $n$.

*MustReachOut(n)*: variables that *must* reach the exit of $n$:

$$MustReachOut(n) = MustReachIn(n) \cup DoesGen(n) \tag{7}$$

### 3.2.1 Elimination Phase

In the elimination phase, we initialize the global sets to $\emptyset$ at the entry to each interval and compute the above global data flow sets using the given equations. If $n$ is an interval exit node, we summarize information into the summary node $h$ as follows:

$$DoesGen(h) = \bigcup_{v \in MustReachOut(e)} Expand(v, j, \text{low:high}) \tag{8}$$

$$SGen(h) = \begin{cases} (\bigcup_{d \in SReachOut(e)} Expand(d, j, \text{low:high})) - LCKill(h) \\ \qquad\qquad \text{(if sequential do loop)} \\ \bigcup_{d \in SReachOut(e)} Expand(d, j, \text{low:high}) - \bigcup_{d \in SAKillOut(e)} Expand(d, j, \text{low:high}) \\ \qquad\qquad \text{(if parallel do loop)} \end{cases} \tag{9}$$

The function *Expand* expands every occurrence of an array reference, $a*j+b$, to $(a*\text{low}+b{:}a*\text{high}+b{:}a)$ [22]. $LCKill(h)$ gives the set of definitions that are killed in later iterations of the loop.

The *MustReachOut* set is used to compute *LCKill*. For example, for an interval exit node $n$ and loop bounds 1:10, assume $SReachOut(n) = \{A_1(i), A_2(i+2)\}$ and $MustReachOut(n) = \{A(i), A(i+2)\}$ at iteration $i$. From $SReachOut$(n), we infer that $A_2(i+2)$ *may* reach $n$. From $MustReachOut$(n), we infer that there exists some definition $d$ corresponding to variable $A(i)$ that *must be* defined in the loop in iteration $i$. Clearly for normalized loops, $d$ must kill $A_2(j)$ from a previous iteration $j$. Hence *LCKill* for the loop includes $A_2(3:10)$. Substituting this result in equation 9, $SGen(h)$ is $\{A_1(1:10), A_2(3:12)\} - \{A_2(3:10)\}$, *i.e.*, $\{A_1(1:10), A_2(11:12)\}$.

While summarizing information in *doall* loops, we do not consider the definitions in other *doall* iterations, because the different iterations of the *doall* execute in parallel. Definitions of the same array section occurring in multiple iterations correspond to a "potential anomaly" in the program. Our data flow analysis will be able to report such anomalies.

*SKill*(h) is computed using standard techniques based on *SGen*(h) and a knowledge of the definitions in other nodes in the interval. *SSynch*(h) is the set of all definitions propagated via synchronization edges to the exit of the loop for all values of the loop index.

### 3.2.2 Propagation Phase

If $n$ is an interval entry node, we initialize *SReachIn*(n) as the union of:

- Definitions reaching $n$ from outside the current loop for any iteration $i$ that are not overwritten in previous iterations and
- Definitions reaching $n$ from any previous iteration of the current loop.

We use the *MustReachOut* sets from the elimination phase in addition to the *SReachIn* and *SReachOut* sets to compute this information. *SAKillIn*(n) for any iteration is initialized as the union of definitions

that must be killed in previous iterations of the current loop and the *SAKillIn*(n) computed in the elimination phase. Again, in the case of `doall` loops, we do not consider the definitions in "lexically earlier" iterations while initializing *SReachIn*(n) and *SAKillIn*(n). *SSynch*(n), *SReachOut*(n) and *SAKillOut*(n) are computed using the same data flow equations as in the elimination phase. The data flow sets at other nodes are computed using the equations given in the elimination phase. Note that the *MustReachIn* and *MustReachOut* sets need not be computed in the Propagation phase since information is not summarized in this phase.

# 4 Live Definitions Analysis

In the *live definitions* problem [25], we say a definition $d$ of a variable $v$ is *live* at a point $p$ in a program if definition $d$ will be used after $p$ and before any redefinitions of variable $v$. We use interval analysis to compute live definitions information for array sections. Since the live definitions problem is a backward problem, the traversal orders in the Elimination and Propagation phases correspond to reverse topological orders as given in Section 3.1.

## Data Flow Sets:

The data flow sets used to compute live definitions information are:

*DefDef*($n$): the set of definitions in $n$.

*DefUse*($n$): the set of definitions that are used in block $n$ before the corresponding variable is redefined.

*LiveIn*($n$) : the definitions live at the entry to a node.

*LiveOut*($n$) : the definitions live at the exit of a node.

For example, in the code fragment for block $n$ in some CFG,

$$\begin{aligned} V_{n.1} &= W_{l.3} \\ W_{n.1} &= V_{n.1} + 1 \\ V_{n.2} &= W_{n.1} + U_{l.3} \\ \ldots &= V_{n.2} \end{aligned}$$

definitions $W_{l.3}$ and $U_{l.3}$ are from a previous block $l$[3]. Definitions $V_{n.1}, V_{n.2}$ and $W_{n.1}$ are in *DefDef*($n$), Definitions $W_{l.3}$ and $U_{l.3}$ are in *DefUse*($n$), because no definition of $U$ or $W$ occurs before $W_{l.3}$ and $U_{l.3}$ are used.

We use additional data flow sets to take care of explicit parallelism: In particular, because of the semantics of cobegin/coend, we need live information collected *inside* each parallel construct. Intuitively, this is true because parallel control flow requires that *all* the successors of a cobegin node *always* execute. Thus, if a definition $d$ is live at the coend but is not live at the entry point of *any* of

[3]We use the notation that definition $X_{i,j}$ denotes the $j^{th}$ definition of variable $X$ in block $i$.

the successors of the corresponding cobegin (due to a redefinition of the variable in this thread), then $d$ is not live at the cobegin as well. We use the *LocalLiveIn* and *LocalLiveOut* sets corresponding to the LiveIn and *LiveOut* sets within specific parallel constructs to handle parallelism.

## Data Flow Equations:

The data flow equations to compute live definitions information are given below. DDef(n) is simply *SGen*(n) computed in the reaching definitions analysis phase. The function *Subset* in equation 10 returns all those definitions of array sections used in $n$ that are subsets of definitions reaching $n$. The set $SUse(n)$ gives the set of array section variables used before redefined in node $n$, similar to the $Use(n)$ set for scalar variables. $DefDef(n)$ is the set of definitions generated in node $n$ and is simply $SGen(n)$ computed in section 3.2.

The definitions live at the entry to a node (equation 11)are those definitions that are *either* live at the exit of the node and are are not generated in this node *or* are used and not generated in this node. The definitions live at the exit of a node (equation 12) are all those definitions that are live at the entry of at least one successor node such that the definition also reaches this successor. While computing the *LiveOut* set, since this is an interval based approach, we consider only successor nodes that are in the current interval ($Succ_i$). Since synchronization successors (*SynchSucc*) can belong to other intervals and synchronization edges propagate data flow information, we consider such nodes as well. We have to consider reaching definitions information while computing $LiveOut(n)$ because a definition may be live at the entry to a successor node, $s$, but may not reach $s$ along the $n \rightarrow s$ edge.

We initialize the live definitions set at the exit of a parallel construct, *i.e.*, *LocalLiveIn*(coend) to *empty*, and compute the *LocalLiveIn* and *LocalLiveOut* sets for nodes within the parallel construct (including the cobegin node) using the equations 11 and 12, *e.g.*, with *LocalLiveIn* substituted for *LiveIn*. These local live sets are used to compute live definition information within specific threads of a parallel construct, and are used as well in the computation of the *Send* and *Receive* sets in section 5.

The actual live sets at any node within the parallel block (including cobegin) are then updated with information at the coend (equations 13 and 14), *i.e.*, any definition that is live at the exit of the parallel block is live at a node within the parallel block only if it also reaches the node.

The equations to compute reaching definitions (section 3.2) introduce additional data flow sets to handle the semantics of arbitrary Post/Wait synchronization. Since we use the reaching definitions information in computing live definitions, Post/Wait synchronization *does not* introduce any additional complexity on the above data flow equations to compute live definitions.

$$DefUse(n) = Subset(SReachIn(n), \bigcup_{v \in SUse(n)} Defs\,Of\,Var\ (v)) \tag{10}$$

$$LiveIn(n) = (LiveOut(n) - DefDef(n)) \cup DefUse(n) \tag{11}$$

$$LiveOut(n) = \bigcup_{s \in (Succ_i(n) \cup SynchSucc(n))} (LiveIn(s) \cap SReachIn(s)) \tag{12}$$

$$LiveIn(n) = LocalLiveIn(n) \cup (LiveIn(\text{coend}) \cap SReachIn(n)) \tag{13}$$

$$LiveOut(n) = LocalLiveOut(n) \cup (LiveOut(\text{coend}) \cap SReachOut(n)) \quad (14)$$

### Elimination Phase

In the elimination phase, the *LiveOut* and *LocalLiveOut* sets at the exit of an interval is initialized to $\emptyset$ and the live information within each interval is computed using the above equations, in the traversal order described in section 3.1. The summary information at summary nodes $h$ are computed as follows ($e$ is the corresponding interval entry node):

$$DefUse(h) = \begin{cases} (\bigcup_{d \in LiveIn(e)} Expand(d, j, \text{low:high})) - DefDef(h) & \\ \qquad\qquad \text{(if sequential do loop)} & \\ \bigcup_{d \in LiveIn(e)} Expand(d, j, \text{low:high}) & \\ \qquad\qquad \text{(if parallel do loop)} & \end{cases} \quad (15)$$

### Propagation Phase

The *LiveOut* and *LocalLiveOut* sets are intialized at the exit node $n$ of an interval as the union of:

- Definitions that are live outside the current loop for any iteration $i$ and that are not generated in later iterations and
- Definitions that are made live in any later iteration of the current loop.

For sequential do loops, the definitions outside the loop that are made live at the interval exit for some iteration of the loop is simply the intersection of the *LiveOut* set of the loop header and the *SReachOut* set of the interval exit node. Since the semantics of parallel loops differ from that of sequential loops, *LiveOut* at interval exit for parallel loops is initialized to the intersection of the definitions live at the *end-doall* node and the definitions that reach the *end-doall* node (equation 12). The traversal order of the PFG in this phase is the same as that mentioned in section 3.1.

We have implemented the interval-based algorithm to compute live definitions in parallel programs using the SETL prototyping language. We used this implementation to compute the live definitions for the program in Figure 1, and the results are shown in Figure 8.

## 5 Send and Receive Sets

Using the reaching definitions and live definitions information, it is possible to derive the data sets that must be communicated at synchronization points in the program. We call these the *Send* and *Receive* sets. For the programs that we consider, synchronization points correspond to: `cobegin`, *doall* nodes and their immediate successors; `coend`, *enddoall* nodes and their immediate predecessors; and corresponding `Post/Wait` nodes.

At a `cobegin` node, the data that must be sent to each forked thread consists of those definitions of variables that are used in the forked thread. The set of definitions that must be received by any thread

| Node | *LiveIn* | *LiveOut* |
|---|---|---|
| (1) | | |
| (2) | | $A_2(i)$ (if $i \in 11:15$) |
| (3) | $A_2(11:15)$ | $A_2(11:15)$ |
| (4) | $A_2(11:15)$ | $A_2(11:15)$ |
| (5) | $A_2(11:15)$ | $A_2(11:15)$ |
| (6) | $A_2(k+10:15)$ | $A_2(k+10:15)$ |
| (7) | $A_2(k+10:15)$ | $A_2(k+11:15), A_7(k:5)$ |
| (8) | $A_2(k+11:15)$ | $A_2(k+11:15)$ |
| (9) | $A_2(15)$ | $A_2(15)$ |
| (10) | $A_2(15)$ | $A_2(15)$ |
| (11) | $A_2(15)$ | $A_2(15)$ |
| (12) | $A_2(15), A_7(j:5)$ | $A_2(15)$ |
| (13) | $A_2(15)$ | $A_2(15)$ |
| (14) | $A_2(15)$ | $A_2(15)$ |
| (15) | $A_2(15)$ | $A_2(15)$ |
| (16) | $A_2(15)$ | |

**Figure 8**: Results of Live Definitions Analysis of Example Program

| Node | *Send* | *Receive* |
|---|---|---|
| (1) | | |
| (2) | $A_2(i)$ (if $i \in 11:15$) | |
| (3) | | $A_2(11:15)$ |
| (4) | $A_2(11:15)$ | |
| (5) | | $A(11:15)$ |
| (10) | | |
| (9) | | |
| (14) | | |
| (15) | | $A_2(15)$ |
| (7) | $A_7(1:5)$ | |
| (12) | | $A_7(1:5)$ |

**Figure 9**: Communication sets for example program

at a fork node, *i.e.*, the *Receive* set of a successor of a cobegin node is that subset of *Send*(cobegin) that is used locally within this thread.

$$Send(\texttt{cobegin}) = Live\,Out(\texttt{cobegin}) \tag{16}$$

$$Receive(s) = LocalLiveIn(s) \tag{17}$$

where, $s$ is a successor of the cobegin node.

The data that should be sent to the coend node are all definitions of variables that reach the coend and used in subsequent nodes. This includes definitions sent from the predecessors of coend and definitions sent from the cobegin:

$$Receive(\texttt{coend}) = LiveIn(\texttt{coend}) \tag{18}$$

The *Send* at a predecessor, $p$, of a coend node is computed as follows:

$$Send(p) = LiveOut(p) - Send(\texttt{cobegin}) \tag{19}$$

*i.e.*, the set of definitions that are live at the exit of $p$ and that are not directly propagated to the coend node from the corresponding cobegin node. This ensures that only the variables defined or communicated to the thread in which $p$ appears need to be communicated to the coend node.

The *Send* and *Receive* sets at *doall* and *end-doall* nodes are:

$$Send(doall) = LiveOut(doall) \tag{20}$$

$$Receive(end\text{-}doall) = LiveIn(end\text{-}doall) \tag{21}$$

The *Receive* set at the interval entry node $s$ for a *doall* loop is:

$$Receive(s) = LiveIn(s) \tag{22}$$

The *Send* set at the interval exit node $p$ for a *doall* loop is:

$$Send(p) = LiveOut(p) \tag{23}$$

Similarly, the *Send* and *Receive* sets at posting and waiting nodes can be computed based on the live definitions and reaching definitions information at these nodes. A definition is in the *Send* set of a posting node if it is in the *LiveIn* set of at least one of its synchronization successors in the Parallel Flow Graph and *reaches* the exit of the posting node. A definition is in the *Receive* set of a waiting node if it is in the *Send* set of at least one of its synchronization predecessors. For a posting node $p$ and a waiting node $q$, $Send(p)$ and $Receive(q)$ are defined as follows:

$$Send(p) = \Big( \bigcup_{s \in SynchSucc(p)} LiveIn(s) \Big) \cap ReachOut(p) \tag{24}$$

$$Receive(q) = \bigcup_{s \in Pred_{sy}(q)} Send(s) \tag{25}$$

The different semantics of post-wait synchronization and parallel control flow at cobegin, coend nodes accounts for the slight difference in the computation of the *Send/Receive* sets at these nodes. We used our implementation to compute the *Send/Receive* sets for the example in Figure 1, shown in Figure 9.

We have also investigated other representations for computing data flow information, such as Static Single Assignment form [10]. Multiple definitions of different but intersecting array sections cannot be represented as $\phi$-functions in SSA without loss of information, though such merging would be of benefit for the same array sections. Therefore, in the general case, merging of information at control flow joins in SSA would not be of benefit if more precise information is to be kept. However, SSA merge points in the flow graph are the points where the single section actually sent in a multiple element *Send* set can be determined, and so are useful for this purpose.

# 6 Conclusions

We have described interval-based algorithms to compute reaching definitions and live definitions information of array sections for programs containing co-begin, co-end constructs, parallel loops, and limited post and wait synchronization; this is a larger class of parallel programs than previously considered in the literature. The degree of difficulty computing such information is clearly related to the presence of post and wait synchronization and the way it is used in the program.

In order to use interval analysis, we have constrained the allowed synchronization patterns. We plan to use the methods of [41, 42, 32], used to handle irreducible flow graphs, to reduce these restrictions. We will isolate the difficult-to-analyze construct in an interval and perform iteration to compute the needed information for that interval. These results can then be integrated into the method we have presented here. However, just as the difficulties with handling irreducible sequential programs have given rise to the use of easier-to-analyze structured constructs, the difficulty of handling such unstructured synchronization suggests the use of more structured and less general synchronization constructs.

We have also shown how to use the results of array section analysis to compute *Send* and *Receive* sets for these parallel programs. These sets provide communication cost estimates useful in partitioning and scheduling of parallel programs, since they identify the actual variables and array sections communicated. We have illustrated here how the use of these sets would be beneficial in making scheduling decisions in two different architectural settings. These *Send* and *Receive* sets can also be used for a variety of other purposes. For instance, they could be used to generate a message-passing program from the input parallel program.

# Acknowledgements

Harini Srinivasan's work was supported by an IBM Graduate Fellowship.

# References


[1] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. *J. Parallel and Distributed Computing*, 5(5):617–640, October 1988.

[2] John R. Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] Vasanth Balasundaram and Ken Kennedy. Compile-time detection of race conditions in a parallel program. In *Proc. 3rd International Conference on Supercomputing*, pages 175–185, June 1989.

[4] W. Blume. Success and limitations in automatic parallelization of the perfect benchmark programs. Technical report, U. of IL-Center for Supercomputing Research and Development, July 1992. CSRD Rpt. No. 1249.

[5] Shahid H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Transactions on Computers*, C-37:48–57, January 1988.

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

[7] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of event synchronization in a parallel programming tool. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 21–30, Seattle, Washington, March 1990. ACM Press.

[8] K.M. Chandy and C.F. Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the Fourth Workshop on Parallel Computing and Compilers*. Springer-Verlag, 1992.

[9] K.M. Chandy and C.F. Kesselman. *Research Directions in Object Oriented Programming*, chapter C++: A Declarative Concurrent Object Oriented Programming Notation. MIT Press, 1993.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[11] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic management of programmable caches (extended abstract). *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988. Also available as CSRD Rpt. No. 728 from U. of Ill.-Center for Supercomputing Research and Development.

[12] D. E. Maydan, S. Amarasinghe and M. S. Lam. Array data flow analysis and its use in array privatization. In *Conf. Record 20th Annual ACM Symp. Principles of Programming Languages*, Charleston, South Carolina, January 1993.

[13] E. Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis and its use in optimizations. *Proceedings of the SIGPLAN 93 conference on programming language design and implementation*, pages 68–77, June 1993.

[14] R. Eigenmann and W. Blume. An effectiveness study of parallelizing compiler techniques. *Proceedings of the International Conference on Parallel Processing*, August 1991.

[15] Parallel Computing Forum. PCF Parallel Fortran Extensions. *FORTRAN Forum*, 10(3), September 1991. Special Issue.

[16] A. Gerasoulis, Venugopal, and T. Yang. Clustering task graphs for message passing architectures. *Proceedings of the 4th ACM Inter. Conf. on Supercomputing (ICS 90)*, pages 447–456, June 1990.

[17] Milind Girkar and Constantine Polychronopoulos. Partitioning programs for parallel execution. Technical report, U. of IL-Center for Supercomputing Research and Development, July 1988. CSRD Rpt. No. 765 Also in Proc. of ACM 1988 Int'l. Conf. on Supercomputing, St. Malo, France, July 4-8, 1988, pp. 216-229.

[18] E. Granston and A. Veidenbaum. Detecting Redundant Accesses to Array Data. In *Proc. of Supercomputing 1991*, pages 854–965, Albuquerque, NM, November 1991.

[19] Thomas Gross and Peter Steenkiste. Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compiler. *Software - Practice and Experience*, 20(2):133–155, February 1990.

[20] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. *Proceedings of SIGPLAN Conference on Principles and Practice of Parallel Programming*, May 1993.

[21] Dirk Grunwald and Harini Srinivasan. Efficient Computation of Precedence Information in Explicitly Parallel Programs. In *Proc. of the Sixth Workshop on Languages and Compilers for Parallel Computing* [46].

[22] Manish Gupta and Edith Schonberg. A Framework for Exploiting Data Availability to Optimize Communications. In *Proc. of the Sixth Workshop on Languages and Compilers for Parallel Computing* [46].

[23] P. Brinch Hansen. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–206, June 1975.

[24] P. Havlak and K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[25] Matthew S. Hecht. *Flow Analysis of Computer Programs.* North Holland, New York, 1977.

[26] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, California, 1990.

[27] S. Flynn Hummel and R. Kelly. A rationale for massively parallel programming with sets. *Journal of Programming Languages*, 1993. Published by Chapman and Hall. To appear.

[28] IBM. *Parallel Fortran Language and Library Reference*, March 1988. Pub. No. SC23-0431-0.

[29] Harry F. Jordan, Muhammad S. Benten, Gita Alaghband, and Ruediger Jakob. The force: A highly portable parallel programming language. In Emily C. Plachy and Peter M. Kogge, editors, *Proc. 1989 International Conf. on Parallel Processing*, volume II, pages II–112 – II–117, St. Charles, IL, August 1989.

[30] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, February 1970.

[31] C. Koelbel. Compiling Programs for Nonshared Memory Machines. Technical report, Purdue University, August 1990.

[32] Thomas J. Marlowe. *Data Flow Analysis and Incremental Iteration.* PhD thesis, Rutgers University, October 1989.

[33] C. McCreary and H. Gill. Automatic determination of grain size for efficient parallel processing. *CACM*, 32(9):1073–1078, September 1989.

[34] Jih-Kwon Peir. *Program Partitioning and Synchronization on Multiprocessor Systems.* PhD thesis, University of Illinois at Urbana-Champaign, March 1986. Report No. UIUCDCS-R-86-1259.

[35] Dick Pountain and David May. A tutorial introduction to OCCAM Programming. March 1987.

[36] Kendall Square Research. Technical summary. Technical report, Kendall Square Research, 1992.

[37] M. Rinard, D. Scales, and M. Lam. Heterogeneous Parallel Programming in Jade. In *Proc. of IEEE Supercomputing 92*, pages 245–258, Nov. 1992.

[38] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing. This monograph is a revised version of the author's Ph.D. dissertation published as Technical Report CSL-TR-87-328, Stanford University, April 1987.

[39] Vivek Sarkar. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. *IBM Journal of Research and Development*, 35(5/6):779–804, September/November 1991.

[40] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. *ACM Conference on Lisp and Functional Programming*, pages 202–211, August 1986.

[41] J. T. Schwartz and M. Sharir. Tarjan's fast interval finding algorithm. Technical report, Courant Institute, New York University, 1978. SETL Newsletter Number 204.

[42] J. T. Schwartz and M. Sharir. A design for optimizations of the bitvectoring class. Technical report, Courant Institute of Mathematical Sciences, New York University, September 1979. Courant Computer Science Report No. 17.

[43] L. Snyder. Type architecture, shared memory, and the corollary of modest potential. *Annual Review of Computer Science*, pages 289–318, 1986.

[44] Harini Srinivasan and Dirk Grunwald. An Efficient Construction of Parallel Static Single Assignment Form for Structured Parallel Programs. Technical Report CU-CS-564-91, University of Colorado at Boulder., December 1991.

[45] R. E. Tarjan. Testing Flow Graph Reducibility. *Journal of Computer and System Sciences*, 9(1):355–365, 1974.

[46] *Proc. of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[47] Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *International J. Parallel Programming*, 16(2):137–178, April 1987.

[48] T. Yang and A. Gerasoulis. Pyrros: Static task scheduling and code generation for message passing multiprocessors. In *Proc. of 6th ACM Inter. Conf. on Supercomputering (ICS 92)*, pages 428–437, July 1992.

[49] T. Yang and A. Gerasoulis. A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors. In *Proc. of IEEE Supercomputing 91*, pages 633–642, Nov. 1991.