Automatic partitioning of a program dependence graph into parallel tasks

by V. Sarkar

In this paper, we describe a general interprocedural framework for partitioning a program dependence graph into parallel tasks for execution on a multiprocessor system. Partitioning techniques are necessary to execute a parallel program at the appropriate granularity for a given target multiprocessor. The problem is to determine the best trade-off between parallelism and overhead. It is desirable for the partitioning to be performed automatically, so that the programmer can write a parallel program without being burdened by details of the overhead target multiprocessor, and so that the same parallel program can be made to execute efficiently on different multiprocessors. For each procedure, the partitioning algorithm attempts to minimize the estimated parallel execution time. The estimated parallel execution time reflects a trade-off between parallelism and overhead and is minimized at an optimal intermediate granularity of parallelism. Execution-profiling information is used to obtain accurate execution-time estimates. The partitioning framework has been completely implemented in the PTRAN system at the IBM Thomas J. Watson Research Center. Partitioned parallel programs generated by this prototype system have been executed on the IBM 3090™ and RP3 multiprocessor systems.

1. Introduction

Partitioning is the bridge between *ideal parallelism* and *useful parallelism.* Ideal parallelism is the parallelism revealed by the control and data dependences [1] of a program. Any two statement execution instances that are not directly or indirectly related by control or data dependences can *potentially* be executed in parallel. Useful parallelism is a subset of ideal parallelism that is suitable for execution on a specified multiprocessor system [2, 3]. The best choice of useful parallelism depends on statement

execution times, synchronization, communication and scheduling overheads, and resource limits such as the number of processors available. The problem is to determine the best trade-off between parallelism and overhead. It is desirable for the partitioning to be performed automatically, so that the programmer can write a parallel program without being burdened by details of the overhead of the target multiprocessor, and so that the same parallel program can be made to execute efficiently on different multiprocessors.

There are two kinds of partitioning that can be performed on a parallel program: *control partitioning*—partition the program into parallel tasks so as to balance parallelism and overhead optimally [2], and *data partitioning*—partition data across tasks so as to further reduce communication overhead and improve data locality [4]. There is a close interaction between these forms of partitioning; together, they yield a complete partition of the program control and data structures. This paper concentrates on control partitioning, because our experience has primarily been with uniform shared-memory multiprocessor systems in which data partitioning is not an issue; on such multiprocessor systems, data locality is implicitly achieved by control partitioning [5, 6].

The *program dependence graph* (PDG) [1] is a popular and general representation of control and data dependences in a parallel program. A PDG node represents an arbitrary sequential computation (e.g., a basic block, a statement, or an operation). An edge in a PDG represents a *control dependence* or a *data dependence.* PDGs do not contain any artificial sequencing constraints from the program text; they reveal the ideal parallelism in a program. PDGs can represent automatically detected parallelism as well as programmer-specified parallelism. They have been shown to be useful for solving a variety of problems, including optimization [1], vectorization [7], translation to dataflow machines [8, 9], code generation for large-instruction-word machines [10, 11], merging versions of a program [12], and automatic detection and management of parallelism [3, 13–15]. Therefore, PDGs are a natural choice for program representation for the partitioning problem. Useful parallelism is represented by a partition of the PDG into subgraphs (tasks) so that all the nodes within a task execute sequentially but the tasks themselves can be executed concurrently.

In this paper, we describe a general interprocedural framework for partitioning the PDG of each procedure into parallel tasks for execution on a specified multiprocessor system. Our approach is interprocedural in that procedures are visited in a bottom-up traversal of the call graph, and the results obtained from partitioning the PDG of a procedure are incorporated at all call sites for that procedure. For each procedure, the partitioning algorithm attempts to minimize the estimated average parallel execution time for a single call to that procedure. Thus, when finally partitioning the main procedure at the root of the call graph, the partitioning algorithm attempts to minimize the estimated parallel execution time for the entire program. The estimated parallel execution time reflects a trade-off between parallelism and overhead and is minimized at an optimal granularity of parallelism. Execution profiling information is used to obtain accurate estimates of average sequential and parallel execution times [16].

The partitioning framework has been completely implemented in the PTRAN compiler system at IBM [3, 13–15]. The input to the PTRAN compiler is usually a sequential FORTRAN program, though the compiler also accepts `PARALLEL DO` constructs in the input. Though the PTRAN implementation deals only with the FORTRAN language, the partitioning framework is applicable to any programming language for which a PDG can be computed (e.g., C and Pascal). The PTRAN analyzer [13] uncovers the ideal parallelism in the program by building a PDG for each procedure. The useful parallelism is then selected by the PTRAN partitioner (or process former), which is the subject of this paper. The output generated by the PTRAN compiler is a parallel FORTRAN program, in which the useful parallelism is expressed by `PARALLEL DO` and `PARALLEL SECTIONS` constructs (similar to `doall` and `cobegin-coend`, respectively [17]). These constructs are part of the IBM VS FORTRAN 2.5 parallel language [18], which is supported on the IBM 3090™ multiprocessor system. These constructs were also supported (with slightly different syntax) by the IBM Parallel FORTRAN language [19]. The partitioned PDGs generated by PTRAN have also been targeted to (i.e., translated for execution on) the RP3 multiprocessor system [20] and can easily be targeted to other multiprocessor systems that support similar parallel extensions to FORTRAN.

The novelty of the approach presented in this paper is that it provides a general partitioning framework that supports both automatically detected and user-specified parallelism, exploits both loop parallelism and nonloop (statement) parallelism, allows arbitrary nesting of parallelism, both within and across procedure calls, and uses architectural parameterization to target to different multiprocessor systems.

The rest of the paper is organized as follows. Section 2 describes the *target multiprocessor model* assumed in this work. Section 3 reviews the definition of the PDG and discusses in detail the *forward control dependence graph* (FCDG), a variant of the PDG that is used by PTRAN. Section 4 describes how the FCDG is annotated with *average execution frequencies* and how execution-profile information is used to obtain accurate estimates of average frequencies. Section 5 describes how frequency values and execution times of primitive operations are used to

estimate *average sequential execution times.* Section 6 defines the *initial task tree* obtained from the FCDG. Section 7 defines the *parallel execution time* of a task tree, which is the cost function to be minimized by the *general partitioning algorithm* described in Section 8. Section 8 also contains a partitioning example and some experimental results. Section 9 describes a specialized partitioning algorithm for *loop-only parallelism*; this algorithm is also used as a pre-pass to the general partitioning algorithm described in Section 8. Section 10 discusses how various *loop transformations* such as loop distribution, loop vectorization, and loop fusion can be incorporated into the partitioner. Finally, Section 11 discusses related work, and Section 12 contains the conclusions of this paper.

2. Multiprocessor model

The output generated by the PTRAN partitioner is a partition of the PDG, for each procedure in the program. A PDG partition is represented by a *task tree* structure, in which a task corresponds to a sequential computation defined by a PDG subgraph, and an edge in the task tree corresponds to a parallel construct (PARALLEL DO or PARALLEL SECTIONS) that relates a child task to its parent task.

The PARALLEL DO statement denotes the beginning of a parallel loop, in which all iterations may be executed concurrently. Its syntax is like that of the FORTRAN DO statement—it specifies an index variable with expressions for lower and upper bounds and optionally specifies a step expression and a target statement label. The PARALLEL DO construct has an implicit "barrier synchronization" at the end: The statement following the PARALLEL DO construct is executed only after all iterations of the PARALLEL DO have completed execution. The task containing the PARALLEL DO statement is called the *parent task*. The task containing iterations of the PARALLEL DO is called the child task. Let p be the number of processors working on the PARALLEL DO construct; then p instances of the child task are created at run time, each to execute approximately $1/p$ of the loop iterations. The run-time system determines how loop iterations are assigned to processors. Currently, the run-time systems implemented on the IBM 3090 multiprocessor use a fixed-chunk-size approach, in which fixed-size groups (chunks) of loop iterations are assigned to processors. Other possibilities for run-time scheduling of parallel loops include guided self-scheduling [21] and factoring [22, 23]. A PARALLEL DO statement may be annotated with directives to specify the number of processors that should be used to work on the loop and the chunk size that should be used when assigning loop iterations to processors.

The PARALLEL SECTIONS statement denotes the beginning of a group of "sections" (tasks) that may be executed in parallel. It is terminated by an END SECTIONS statement. Each task begins with a declaration of the form SECTION i, where i is a positive integer constant that identifies the task being defined. The SECTION i declaration may be followed by a clause of the form WAITING(n_1, n_2, $\cdots$). If the WAITING clause is present, it means that task i can start execution only after tasks n_1, n_2, $\cdots$ have completed. As a simple way of ensuring that the precedence constraints are acyclic, the WAITING clause is allowed to refer only to tasks that have been previously declared in the PARALLEL SECTIONS statement. Like the PARALLEL DO, the PARALLEL SECTIONS statement has an implicit barrier synchronization at the end. Figure 5 in Section 8 shows a parallel program containing PARALLEL DO and PARALLEL SECTIONS constructs.

The PARALLEL DO and PARALLEL SECTIONS constructs specify *tasks* at compile time that may be dynamically instantiated at run time. Note that the parallelism expressed by these constructs may be arbitrarily nested and may cross procedure boundaries (e.g., if a procedure containing a parallel construct is itself called from within a parallel construct). As with other parallel language systems that support nested parallelism, the VS FORTRAN 2.5 and Parallel FORTRAN run-time systems [18, 19] begin program execution by creating a fixed number of operating system processes (called "FORTRAN processors"), usually one for each real processor. After this, all task scheduling is performed by executing library routines on the FORTRAN processors, without requesting any services from the operating system. Each FORTRAN processor selects work to do from a shared queue. The VS FORTRAN 2.5 and Parallel FORTRAN run-time systems can be executed on both the MVS and VM operating systems [18, 19].

The target multiprocessor parameters currently used by the partitioner are as follows (assuming that all time parameters are expressed as multiples of the processor cycle time):

- Execution times of primitive operations (add, multiply, etc.). These values are used to compute the local execution time, $cost(u)$, of each node u in the PDG.
- P (≥ 1), the average number of processors available to the user. P need not be an integer. For example, P may be set to 3.5 for a four-way multiprocessor in which the system uses half the available time slices on one of the four processors.
- $T_{\text{start-up}}$, the start-up overhead of a task.
- $T_{\text{fork-join}}$, the total fork-join overhead incurred in a parent task (PARALLEL DO or PARALLEL SECTIONS) for creating and terminating its child tasks. $T_{\text{fork-join}}$ is modeled as a linear function of k, the number of child tasks created by the parent task: $T_{\text{fork-join}} = T_{\text{parent}} + kT_{\text{child}}$ (k is the number of processors executing a PARALLEL DO construct or the number of sections in a PARALLEL SECTIONS construct).

- T_{signal} and T_{wait} are the *signal* and *wait* overheads of a WAITING clause in the PARALLEL SECTIONS construct. For each pair of tasks (T_i, T_j), such that task T_j waits for task T_i, T_{signal} is added to the execution time of task T_i and T_{wait} is added to the execution time of task T_j.

The values of $T_{start\text{-}up}$, T_{parent}, T_{child}, T_{signal}, and T_{wait} depend on the implementation of the run-time system as well as the hardware characteristics of the target multiprocessor. In the future, we shall also include memory-access costs in our model, using parameters for the cache and other levels of the memory hierarchy as in [6].

3. Program dependence graph

- *Background*

A *program dependence graph* (PDG) consists of a set of nodes connected by *control dependence* edges and *data dependence* edges [1]. A PDG node represents an arbitrary sequential computation (e.g., a basic block, a statement, or an operation). The current PTRAN implementation uses a statement-level PDG. A control or data dependence edge is an ordered pair of the form (n_a, n_b) (indicating a dependence from node n_a to node n_b), augmented with some context information. For a control dependence edge, the context information is the *branch label* or predicate value that caused the dependence [1]. For a data dependence edge, the context information includes the nature of the dependence (*flow/anti/output*) [24], the pair of read/write accesses (contained within nodes n_a and n_b) that caused the dependence, the *direction vector* [25] of the dependence, and, possibly, the *distance vector* [26] of the dependence.

Building a PDG from a sequential program is a well-known procedure. The starting point is to build a *control flow graph* (CFG) for the program [27]. Each node in the CFG corresponds directly to a node in the resulting PDG. An algorithm for computing the PDG control dependence edges from the control flow graph was given in [1]; an improved version of the algorithm appears in [28]. The PDG data dependence edges are data flow edges (def-use, use-def, and def-def chains) [27] that have been refined or eliminated by data dependence tests for array accesses [3, 25, 29]. The control dependence edges and the PDG nodes together form the *control dependence subgraph* of the PDG; similarly, the data dependence edges and the PDG nodes together form the *data dependence subgraph* of the PDG.

We devote the remainder of this section to discussing the *forward control dependence graph* (FCDG) [15, 30, 31], a variant of the standard definition of a PDG control dependence subgraph [1][1]. The major advantages of the FCDG over the standard PDG control dependence subgraph is that the FCDG has a hierarchical structure that is consistent with the loop-nesting structure of the original CFG, and that the FCDG is acyclic, thus simplifying the algorithms for identifying nonloop parallelism and for process formation [3, 15, 31, 32]. However, the definition of the FCDG presented in this section reflects the current PTRAN implementation [3] and differs slightly from earlier definitions of forward control dependence presented in [15, 30, 31].

As in computing the PDG, the starting point for computing the FCDG is the original control flow graph (CFG). The FCDG is computed from the CFG as follows:

1. Compute the *interval structure* (Definition 2 below) of the CFG [33, 34]. The interval structure captures the program loop-nesting structure in *hdr*, *hdr_parent*, and *hdr_lca* data structures.
2. Build the *augmented control flow graph* (ACFG) by inserting *preheader* and *postexit* nodes (along with appropriate incoming and outgoing edges) into the CFG. The locations of the *preheader* and *postexit* nodes are determined by the CFG interval structure [15].
3. Compute the *forward control flow graph* (FCFG) from the ACFG by inserting an *exits* node for each interval, retargeting each *back edge* (Definition 2) in the interval to the *exits* node, and inserting edges from the *exits* node to each *postexit* node of the interval [3]. The FCFG is a variant of the ACFG in which each back edge from the ACFG is made to go in the "forward" direction to a special *exits* node. Therefore, the FCFG is an acyclic graph.
4. Finally, compute the *forward control dependence graph* (FCDG) from the FCFG, by using any of the known algorithms to compute control dependence [1, 28]. The control dependence edges along with the set of FCFG nodes define the FCDG. For a given node Y in the FCDG, we define the *control conditions of node Y* as the set

   $$CC(Y) = \{(X, L) | Y \text{ is control dependent on } X \text{ with label } L\}.$$

   Two nodes are said to be *identically control dependent* if and only if they have the same set of control conditions in the FCDG.

The following subsections describe these steps in more detail by discussing *interval structure*, the *augmented control flow graph*, the *forward control flow graph*, and the *forward control dependence graph*.

- *Interval structure*

We start with a definition of the control flow graph.

[1] Only the definition of control dependence is affected by this variation. Our work uses the standard definition for a PDG data dependence subgraph [1, 24].

Definition 1 A *control flow graph* $CFG = (N_c, E_c, T_c)$ consists of

- N_c, a set of nodes.
- $E_c \subseteq N_c \times N_c \times \{T, F, U, Z_i\}$, a set of *labeled* control flow edges. If any two edges, (a_1, b_1, l_1) and (a_2, b_2, l_2), in E_c have the same source and destination nodes, they must have distinct labels (i.e., $a_1 = a_2 \wedge b_1 = b_2 \Rightarrow l_1 \neq l_2$).
- T_c, a node-type mapping. $T_c(n)$ identifies the type of node n as one of the following values: *cfgtop*, *cfgbot*, *preheader*, *postexit*, *exits*, *other*. □

We assume that *CFG* contains two distinguished nodes of type *cfgtop* and *cfgbot*, respectively, and that for any node N in *CFG,* there exist directed paths from *cfgtop* to N and from N to *cfgbot*. The node types *preheader*, *postexit*, and *exits* are reserved for special nodes created during the construction of the ACFG and the FCFG. The node type *other* is used for all other nodes.

The notation used for CFG edge labels is as follows: Labels T and F represent *true* and *false* conditional branches, respectively. Label U represents an *unconditional* branch. Labels of the form Z_i represent *pseudo* control-flow edges and indicate that the corresponding branch can never be taken in the original program. However, the insertion of these pseudo edges provides a convenient structure to the FCDG, as described later. We assume that we may use an unlimited number of labels of the form Z_i for pseudo edges in order to maintain the property that all edges from a CFG node have distinct labels.

The loop-nesting structure of the original CFG is defined by its *interval structure* [33, 34] (see Definition 2 below). Currently, PTRAN considers only single-entry loops as candidates for loop parallelism. In practice, this is not a significant restriction. All structured loops (e.g., `do`, `while`, `repeat-until`) are single-entry even though they may contain multiple exits; also, most unstructured loops (built out of `goto` statements) found in real programs are single-entry as well. A multiple-entry loop can be transformed into multiple single-entry loops by *node splitting* [27, 35], in which a separate copy of the loop is included with each loop entry. Node splitting is practical only when the expansion in code size does not become prohibitive.

Definition 2 [33, 34] A *back edge* in *CFG* is an edge (x, h, l) such that node h dominates node x (node x is called a *latch node*, and node h is called a *header node*) [27]. A back edge defines a *strongly connected region* $str(h, x)$, which consists of the nodes and edges belonging to all paths from node h to node x, along with the back edge (x, h, l). Consider the set $B(h) \neq \emptyset$ of back edges targeted to header node h. The union of the strongly connected regions defined by the back edges in $B(h)$ is called the *interval with header h* denoted by $I(h)$.

Intervals may be nested, interval I_1 being a *subinterval* of interval I_2 if I_1 is a subgraph of I_2. An interval may contain arbitrarily many subintervals. We represent interval nesting by a mapping called *hdr_parent*, in which $hdr\_parent(h_1) = h_2$ indicates that the interval with header node h_1 is an immediate subinterval of the interval with header node h_2. $hdr\_parent(h) = 0$ indicates that the interval with header node h is the *outermost* interval.[2] *hdr_parent* defines a directed tree on all header nodes. *hdr_lca* is a mapping such that $hdr\_lca(h_1, h_2) = h_3$ indicates that header node h_3 is the *least common ancestor* of header nodes h_1 and h_2 in this tree.

Finally, we observe that a node may be contained within several enclosing intervals. We use the term *node-n interval* to mean the innermost interval containing node n, and we define $hdr(n)$ to be the header node of the innermost interval containing node n. □

Definition 2 is based on the definition of intervals in [34], which is essentially equivalent to the definitions formulated by Schwartz and Sharir [33] and by Graham and Wegman [36]. However, it differs from the definition of intervals due to Allen and Cocke [37] (also in [27] and [35]), which does not require an interval to be strongly connected. This distinction is discussed in detail in [3].

As mentioned earlier, PTRAN considers only single-entry loops as candidates for loop parallelism. For the sake of simplicity, the current implementation of the partitioner invokes the partitioning algorithm only for procedures that contain no multiple-entry loops—i.e., for procedures that have a *reducible* [27, 35, 38, 39] CFG. This is an *ad hoc* restriction that can be removed in the future by serializing only the region corresponding to the multiple-entry loop, while attempting to exploit parallelism outside that region. Each serialized multiple-entry loop can simply be replaced by a single acyclic node in the CFG, thus satisfying the partitioner requirement that the CFG be reducible. The advantage of working with a reducible CFG is that it has a unique interval structure.

- *Augmented control flow graph (ACFG)*

The next step after determining the interval structure is to build an augmented control flow graph (ACFG) [15]. Compared to the original CFG, the ACFG makes loop structure evident by the insertion of *preheader* and *postexit* nodes. Formally, $ACFG = (N_a, E_a, T_a)$ is computed from $CFG = (N_c, E_c, T_c)$ as follows:

1. Initialize, $N_a \leftarrow N_c$; $E_a \leftarrow E_c$; $T_a \leftarrow T_c$.

[2] Strictly speaking, there may be several outermost intervals that are not connected together strongly. For convenience, we assume that there is exactly one outermost interval—the one containing the *ENTRY* node. This assumption is satisfied if the outermost interval is required only to be connected rather than strongly connected. One could also add pseudo control flow edges that result in a single strongly connected outermost interval.

```
PARAMETER(ISIZE = 800000)
REAL*4 X(ISIZE), Y(ISIZE), Z(ISIZE)
1:  ENTRY
2:  Y(1) = 0
3:  Z(1) = 0
4:  PI = 3.14159265
5:  READ(5,*)N
6:  IF (N .LE. ISIZE) THEN
7:    IF (N .GE. 1) THEN
8:      DO I = 1, N
9:        X(I) = SIN(PI*I/FLOAT(N))*PI/FLOAT(N)
10:     END DO
11:     DO J = 2, N
12:       Y(J) = X(J)/2.0 + Y(J - 1)
13:     END DO
14:     DO K = 2, N
15:       Z(K) = X(K)/3.0 - Z(K - 1)
16:     END DO
17:     WRITE(6,*) Y(N), Z(N)
        STOP
      END IF
    END IF
18: WRITE(6,*) 'Bad value of N'
    END
```

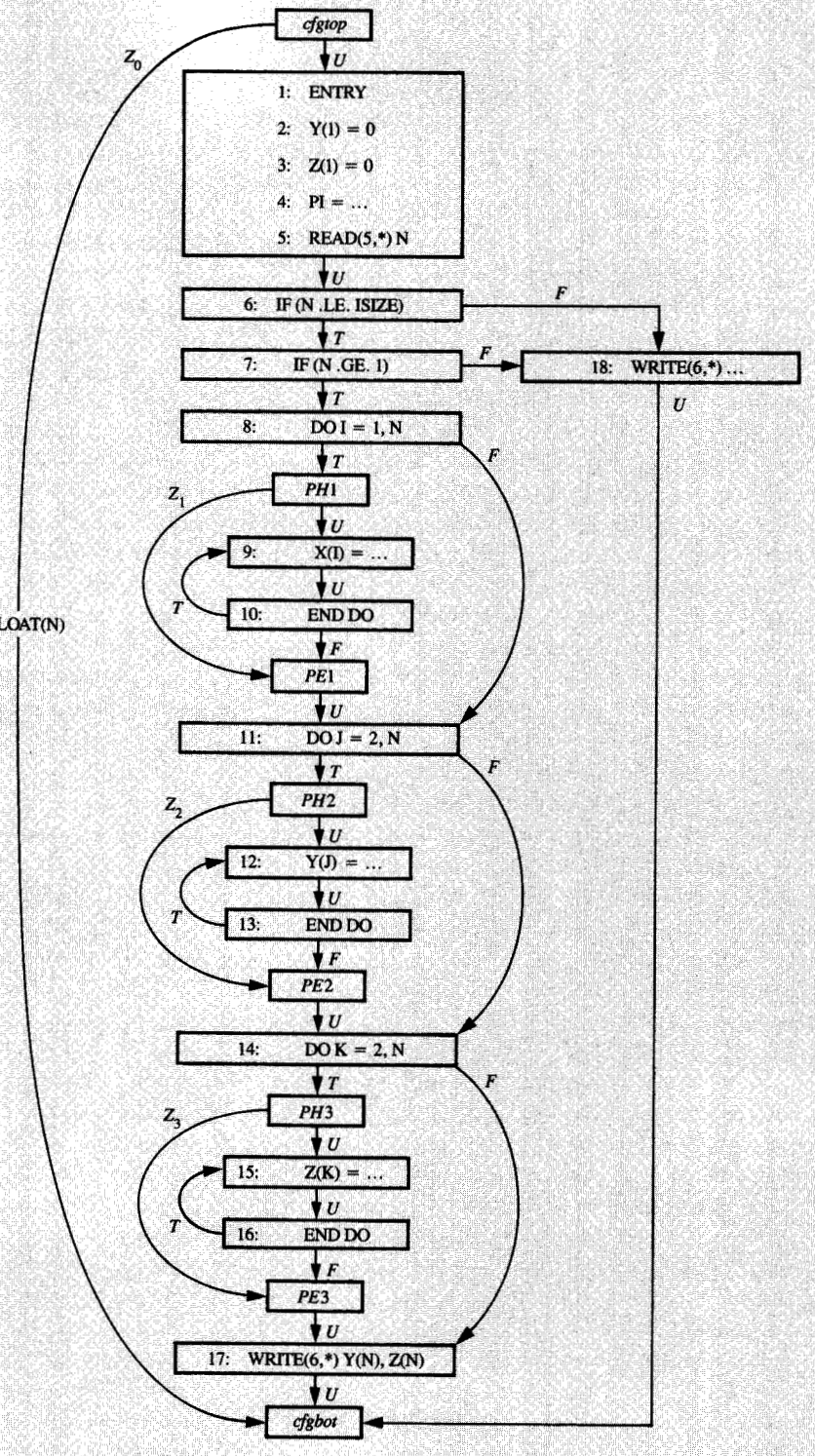

Figure 1

Example FORTRAN program and its augmented control flow graph.

2. For each header node h in CFG,
   (a) Create a new *preheader* node ph, add it to N_a, and designate it as the *preheader* of h [i.e., $T_a(ph) = preheader$].
   (b) For each control flow edge (u, h, l) in E_c in CFG, if $hdr\_lca[hdr(u), h] \neq h$, then (we have an interval entry),
   Replace (u, h, l) with (u, ph, l) in $ACFG$.
   (c) Add an unconditional branch from ph to h [i.e., add edge (ph, h, U) to E_a].

3. For each control flow edge (u, v, l) in CFG, if $hdr\_lca[hdr(u), hdr(v)] \neq hdr(u)$, then (we have an interval exit)
   (a) Create a new *postexit* node, pe, and add it to N_a.
   (b) Replace edge (u, v, l) with edges (u, pe, l) and (pe, v, U).
   (c) Add a *pseudo* control flow edge from the preheader node of the node-u interval to the new *postexit* node pe. The edge label has the form Z_i. (Each new *pseudo* control flow edge receives a distinct label.)

Figure 1 shows a FORTRAN program and its ACFG. The three pairs of preheader-postexit nodes ($PH1$, $PE1$), ($PH2$, $PE2$), ($PH3$, $PE3$) arise from the intervals defined by the three DO loops in the program. Labels Z_0, Z_1, Z_2, Z_3 identify the four *pseudo* control flow edges in the example ACFG. Note that the STOP statement is represented by an unconditional branch to *cfgbot*. Also note that each DO statement contains a conditional branch denoting a *null-iteration-range* test: If the branch condition evaluates to T, at least one iteration is executed and the flow of control is passed on to the preheader node; if the branch condition evaluates to F, no iteration is executed and the loop body is skipped. The END DO statement serves the purpose of a *repeat-until* test performed at the end of each iteration.

- *Forward control flow graph (FCFG)*

The forward control flow graph, $FCFG = (N_f, E_f, T_f)$, is then computed from the augmented control flow graph $ACFG = (N_a, E_a, T_a)$ as follows [3]:

1. Initialize, $N_f \leftarrow N_a$; $E_f \leftarrow E_a$; $T_f \leftarrow T_a$.
2. For each interval in $ACFG$,
   (a) Add an *exits* node, e, to N_f.
   (b) Add *pseudo* control flow edges in E_f from node e to each postexit node pe in the interval. The labels of the edges have the form Z_i.
3. For each back edge (u, h, l) in $ACFG$,
   Replace edge (u, h, l) in E_f with edge (u, e, l), where e is the *exits* node for the header-h interval.

The only difference between an ACFG and its FCFG is that the back edges from the ACFG are "rewired" to their corresponding *exits* nodes in the FCFG. An *exits* node summarizes all possible paths for the remaining iterations in the interval and is followed by a branch to all *postexit* nodes in the interval. If the interval has a single *postexit* node, the insertion of a new *exits* node is unnecessary, since all back edges can be directly rewired to the *postexit* node (this is the case for all intervals in Figure 1).

- *Forward control dependence graph (FCDG)*

Having constructed the FCFG as described in the previous subsection, we compute the FCDG by using any known algorithm to compute control dependence [1, 28]. The control dependence edges, along with the set of FCFG nodes, define the FCDG. **Figure 2** shows the FCDG derived from the ACFG of Figure 1 (the *freq*, *cost*, and *seqtime* annotations are discussed later, in Sections 4 and 5).

We now summarize the important properties of the FCDG:

- The FCDG is acyclic (follows from the fact that the FCFG is acyclic).
- If the CFG is a structured control flow graph (obtained from `begin-end`, `if-then-else`, and `while` control structures only [40]), the FCDG must be a tree (each node can have at most one control condition—see [30] for details). Note that the FCDG in Figure 2 is not a tree because node 18 has two control conditions (predecessors in the FCDG). This is consistent with the fact that the original control flow graph in Figure 1 is not structured (because of the STOP statement).
- An FCDG node is directly or indirectly control dependent on the preheader nodes of all intervals that enclose the node (follows from the pseudo edges inserted in the ACFG construction).

For a structured program, the FCDG looks quite similar to the *abstract syntax tree* [27] of the program. This is because the control flow information of a structured program is captured accurately by the program syntax. Many parallelizing compilers for structured programming languages (e.g., the SISAL [41, 42] and Id [43] compilers) depend on the program syntax for providing control dependence information. However, the program syntax cannot provide control dependence information for unstructured programs, and the more general approach of computing PDG control dependences or of computing the FCDG (as in PTRAN) must be used instead. Not only is the FCDG more general than an abstract syntax tree, but it also has a simpler execution model and semantics. In an abstract syntax tree, each control structure is assigned a distinct node type with special-case semantics, whereas in

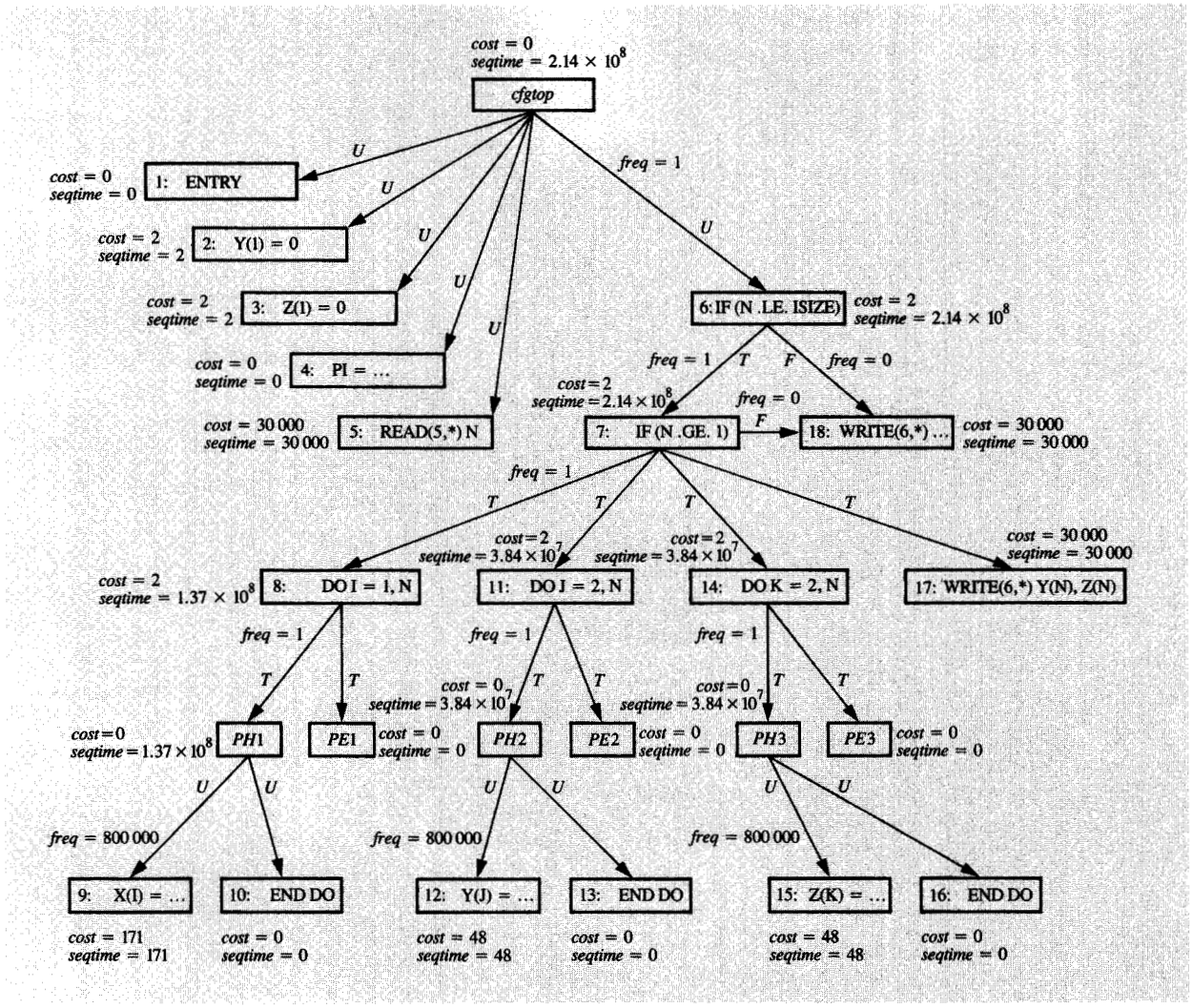

Figure 2

Forward control dependence graph (FCDG) for program of Figure 1 (frequency values obtained from execution profile, for input N = 800 000).

the PDG/FCDG approach, all control structures are expressed using the same primitives of basic blocks and conditional and unconditional branches. Finally, the algorithmic complexity of computing the FCDG from the CFG is just linear in the size of the FCDG.

In conclusion, we state that the FCDG is a simple, general, and efficient representation of the control dependences in a program. Experiences with control dependence in the PTRAN system [15] and elsewhere [1, 7, 10–12] have shown that it is a powerful representation for various analyses, optimizations, and transformations. The following sections demonstrate that the FCDG also lends itself very naturally to solving the partitioning problem.

4. Automatic execution profiling—Determining average execution frequencies

Automatic execution profiling is an empirical means of obtaining average loop frequencies and conditional-branch probabilities in a program. A *counter-based* profiling system has been implemented in PTRAN as a support facility for the partitioning framework [3, 16]. When profiling is requested, PTRAN generates an instrumented program that is equivalent to the input program but augmented with profiling statements to initialize and update counter variables. The instrumented program is then executed by the user for any input that he chooses, and the final values of the counter variables are stored in the program database.

To address the efficiency concerns associated with counter-based profilers, we developed three effective optimizations that reduce the number of counters required to profile a program [3, 16]. These optimizations usually reduce the overhead of counter-based profiling to 5% or less, thus making it practical for profiling to be performed on every run (sequential or parallel) of a user program. In other work, a spanning-tree algorithm is used for optimized counter-based profiling [44, 45]. **Table 1** compares the overhead of the QP profiling tool [45] with that of PTRAN for four FORTRAN SPEC benchmark programs. (The "slow" and "quick" entries are for unoptimized and optimized profiling, respectively.) For these programs, the PTRAN profiler uses many fewer counter operations than the QP tool. Though the spanning-tree algorithm from [44, 45] subsumes two of the three profiling optimizations performed by PTRAN, the third profiling optimization performed by PTRAN (updating iteration counts at loop entry/exit) has a big impact in reducing the number of counter increments. Another reason for the difference in profiling overhead is that PTRAN profiling is performed for a statement-level CFG, which usually contains fewer basic blocks than the instruction-level CFG used in [44, 45].

Regardless of how the frequency information is obtained, it is used to label all edges in the FCDG with *relative* frequency values, according to the following definition.

Definition 3 Given an edge (u, v, l) in the forward control dependence subgraph (FCDG), the execution frequency of the edge is denoted $freq(u, l)$. We use $freq(u, l)$ instead of $freq(u, v, l)$ because the frequency of edge (u, v, l) depends only on the control condition (u, l) and not on the target node v. Also, $freq(u, l)$ is a nonintegral value in general, because it represents an *average* execution frequency over all execution instances of node u. $freq(u, l)$ is defined as follows:

- When u is a preheader node and l is U (the label that identifies loop entry); i.e., when (u, v, l) represents a *loop-control dependence*:

  $freq(u, l)$ = *average number of iterations* for the u interval
  = average number of times the interval header node is executed each time the interval (loop) is entered.

  In this case $freq(u, l) \geq 1$.
- For all other cases; i.e., when (u, v, l) represents a *branch-control dependence*:

  $freq(u, l)$ = *branch probability* of label l in node u
  = fraction of times that node u takes the branch labeled l.

  In this case, $0 \leq freq(u, l) \leq 1$. □

Table 1 Comparison of the profiling algorithms in [45] with the PTRAN profiling algorithm.

SPEC benchmark	*Number of counter increments*		
	QP tool (slow)	*QP tool (quick)*	*PTRAN*
DODUC	130,897,009	45,651,338	6,380,840
NASA7	298,530,617	254,628,038	1,235,007
MATRIX300	60,035,631	54,951,383	1,451,143
TOMCATV	35,012,274	27,762,776	236,389

As an example, the edges of the FCDG in Figure 2 are annotated with relative frequency values based on the execution profile obtained from an input value of $N = 800000$. (A frequency value is shown only once for each label.)

5. Determining average execution times

In this section, we describe how average execution times can be computed for all nodes in the FCDG. Once frequency values have been obtained as described in the previous section, the other necessary input is the execution times of *primitive* operations (e.g., load, store, add, and multiply). We do not discuss the possible techniques for obtaining the costs of primitive operations, except to point out that these costs depend on the target architecture. For the purpose of this description, it is assumed that the primitive operation costs have already been accumulated for each FCDG node and that $cost(u)$ is the sum of the primitive operation costs for node u. One approach for estimating primitive operation costs is described in [46].

We define the average sequential execution time of node u, $seqtime(u)$, to be the sum of $cost(u)$ and the frequency-weighted execution times of the children of node u in the FCDG. The computation of $seqtime(u)$ is based on two simple rules (further details are given in [16]):

1. $seqtime(u) = cost(u) + \Sigma_{(u,v,l)\in FCDG} freq(u, l) seqtime(v)$.

   This rule assumes that the execution time of node v is independent of which conditional branch caused it to execute.
2. If node u is a procedure or function call, $cost(u)$ also includes $seqtime(entry)$, where *entry* is the entry node of the FCDG of the called subprogram. This rule assumes that the execution time of a procedure call is independent of the call site.

Rules 1 and 2 implicitly dictate how the execution-time values should be computed. Rule 1 requires that the nodes be visited in a bottom-up traversal of FCDG. Rule 2 requires that the procedures be visited in a bottom-up

traversal of the call graph, so that the root procedure (main program) is visited last. For the purpose of this discussion, we assume that the call graph is acyclic, which is a reasonable assumption for FORTRAN programs. In previous work [2], we discussed how cyclic call graphs and recursive programs can be handled for the single-assignment programming language SISAL [41]. Those interprocedural techniques are equally applicable to the partitioning framework described in this paper.

Reference [16] also describes how this approach to computing average execution times can be generalized to compute *execution-time variance*. To define variance precisely, let X_v be the random variable corresponding to the execution time of node v. So far, we have discussed the computation of the average execution time of node v [i.e., $seqtime(v)$], which is the expected value of X_v [denoted by $E(X_v)$]. By extending rules 1 and 2 above to compute the expected value of X_v^2 [denoted by $E(X_v^2)$], we obtain the variance by using the well-known definition, $var(X_v) \equiv E(X_v^2) - [E(X_v)]^2$. The standard deviation is defined by $std\_dev(X_v) \equiv \sqrt{var(X_v)}$, and the normalized standard deviation standard deviation is defined by $\sigma(X_v) = std\_dev(X_v)/seqtime(v)$.

This framework for computing average execution times and variance has been implemented in PTRAN. The average execution times are used to guide the partitioner component, as described in later sections. An interesting application of variance information is for determining the optimal *chunk size* for the execution of parallel loops on multiprocessors [47]. When the execution time of the loop body has zero variance, the preferred chunk size value for N iterations on P processors is N/P rounded up, since that provides perfect load balancing with the smallest overhead. However, when the variance is large, the optimal chunk size value decreases so as to provide better load balancing, despite the increased overhead due to a larger number of chunks. We intend to use the variance information in the future to guide the run-time system in selecting an appropriate chunk size for a parallel loop. Variance information will also be useful in determining whether a region of code should be scheduled at compile time or at run time. If the variance is small, the execution times are fairly predictable, and we can use compile-time scheduling algorithms such as those in [2, 48] to map computations directly onto the processors; if the variance is large, the execution times are less predictable, and run-time scheduling is a better choice.

The nodes of the FCDG in Figure 2 are annotated with *cost* and *seqtime* values computed using the relative frequency values determined in Section 4. The *cost* values were obtained by using an internal trace option in the VS FORTRAN compiler. An option in PTRAN is used to automatically pick up the *cost* values from the listing file generated by the VS FORTRAN compiler. The *cost* values are scalar execution times estimated from the operations revealed by the intermediate program representation used by the VS FORTRAN compiler; these estimates are approximate because they do not reflect the actual instructions that are selected for a statement when target code is generated.

6. Task-tree definition and initialization

Sections 4 and 5 described how the FCDG is annotated with average execution frequencies and average sequential execution times. This section defines a *task partition* as a mapping (*taskid*) from nodes to tasks and defines a *task tree* as the reduced graph obtained by applying the *taskid* mapping on the FCDG. This section also describes how the initial task tree is constructed to provide the starting point for the partitioning algorithm in Sections 8 and 9. The partitioning algorithm iteratively merges adjacent tasks in the task tree on the basis of critical-path-length values and overhead values until the task tree is reduced to a single task. Among all task trees in this iterative sequence, the one with the smallest parallel-execution-time value is selected as the optimized task partition for the current procedure.

A task partition is defined for a single procedure. It is simply a mapping, called *taskid*, from nodes to tasks that specifies which nodes belong to the same task. For example, if $taskid(u) = taskid(v)$, nodes u and v belong to the same task. By applying a given *taskid* mapping on the FCDG we obtain a reduced graph on the tasks, containing the edges

$$E_T = \{[taskid(u), taskid(v), (u, l)] \mid taskid(u) \neq taskid(v) \text{ and } (u, v, l) \text{ is an edge in } FCDG\}.$$

An edge in E_T has the form $[T_a, T_b, (u, l)]$, where T_a is the source task, T_b is the destination task, and (u, l) is the control condition of the edge in *FCDG* that corresponds to edge $[T_a, T_b, (u, l)]$ in E_T.

Since the parallel FORTRAN language generated by PTRAN can express only structured parallelism, the reduced graph defined by edges in E_T must be a fan-out tree (i.e., must not contain any fan-in tasks). Recall that it is possible that FCDG is not a fan-out tree, because an FCDG node may have multiple predecessors (e.g., node 18 in Figure 2). To obtain a tree-structured task partition from a non-tree-structured FCDG, we first merge each fan-in node in the FCDG with its dominator in the FCDG. This yields a program with structured parallelism, while permitting unstructured sequential code to occur within a task. This tree-structure constraint will be removed in future work, when unstructured models of parallelism such as the MGOTO operator in [49] are considered.

Thus, for a given *taskid* mapping that satisfies the tree-structure constraint on *taskid*, the set of edges in the *task tree* is given by E_T, the set of edges in the reduced graph.

The task-tree edges define the parallel constructs present in the partitioned program as follows. Consider a parent task T_i with m child tasks, $T_{j1}, \cdots, T_{jm}$, for control condition (u, l) (defined by the task tree edges $[T_i, T_{j1}, (u, l)], \cdots, [T_i, T_{jm}, (u, l)]$). Two cases arise (examples of both cases can be seen later, in Figure 3):

1. (u, l) *represents a null-iteration-range branch condition.* In this case, control condition (u, l) represents a `PARALLEL DO` construct. Node u (contained in task T_i) must correspond to a `DO` statement or a `PARALLEL DO` statement in the input program. Let ph be the preheader node for this loop. ph represents the execution of multiple iterations of the loop (Definition 3) and is contained in task $T_{ph} = taskid(ph)$, which must be one of the child tasks in the set $\{T_{j1} \cdots, T_{jm}\}$. When node u is executed in task T_i and the null-iteration-range test for node u returns a branch-condition value of $l = T$, p new tasks are created as dynamic instantiations of T_{ph} to execute loop iterations in parallel, as described in Section 2. The value of p is selected by the compiler and the parallel run-time system as the ideal number of processors for the given loop. If the loop contains a loop-carried dependence, the compiler forces $p = 1$ so as to serialize the loop (the partitioning algorithm will then merge child tasks $T_{j1}, \cdots, T_{jm}$ with parent task T_i). As described later, a *parloop* mapping is used to identify loops that may legally be executed in parallel.
2. (u, l) *does not represent a null-iteration-range branch condition.* In this case, control condition (u, l) represents a `PARALLEL SECTIONS` construct. When the (u, l) control condition is reached in task T_i (i.e., when the branch with label l is taken from node u in task T_i), m new tasks are created corresponding to $T_{j1}, \cdots, T_{jm}$, as described in Section 2. As described later, *wait_pred* and *wait_succ* mappings identify the necessary synchronization among the m child tasks in the task tree.

The initial task tree reveals the maximum possible parallelism that can be exploited in the input program, given the constraints imposed by its control and data dependences and by the structure of the parallel constructs supported by the target multiprocessor. The steps involved in constructing the initial task tree are as follows:

1. *Initialize.* Start by placing each FCDG node in a separate task, $taskid(u) = u$.
2. *Merge fan-in nodes.* If the FCDG is not a fan-out tree, each *fan-in* node is merged with its *dominator* [27] in the FCDG, along with all intermediate nodes. The resulting task graph is a fan-out tree. The merging is performed by successive applications of the *merge_children* transformation described later, in Section 8.
3. *Identify parallel loops.* The control and data dependence constraints derived for the program are used to determine whether iterations of a given loop can be executed concurrently. If there are any *loop-carried* control or data dependences, the loop must be executed sequentially. A premature exit from within a `DO` loop is an example of a loop-carried control dependence. Loop-carried data dependences occur when different loop iterations may perform conflicting read/write accesses on a shared variable. Constant propagation, induction variable analysis, and privatization enhance loop parallelization by removing potential loop-carried data dependences [3, 13, 32]. The result of this step is stored in a Boolean mapping called *parloop*. *parloop(ph) = true* indicates that it is legal to execute the loop with preheader node ph as a `PARALLEL DO`.
4. *Identify waiting constraints.* The previous step ensures that all loop-carried dependences are satisfied by executing some loops sequentially. This step ensures that all *loop-independent* data dependences are satisfied, by using *waiting* synchronizations wherever necessary [31]. The result of this step is stored in two task-set mappings called *wait_pred* and *wait_succ*; $wait\_pred(T_a)$ is the set of tasks that T_a must wait for (by using the `WAITING` clause in the `PARALLEL SECTIONS` construct), and *wait_succ* is simply the inverse of *wait_pred*. If tasks T_a and T_b are related by a waiting synchronization, T_a and T_b must be identically control-dependent siblings in the task tree (i.e., they must have the same parent task, T_p, and the edges from T_p to T_a and T_b must have the same control condition). The set of waiting synchronizations among task-tree siblings can be determined from the loop-independent data dependences computed for the program [31]. Note that the parent–child waiting synchronizations are implicit in the `PARALLEL DO` and `PARALLEL SECTIONS` constructs represented by the task tree.

Figure 3 shows the initial task tree obtained for the program shown in Figures 1 and 2. The *parloop*-mapping values are shown as annotations to the preheader nodes. The synchronization edges (waiting constraints) are shown as dotted arrows. Note that nodes 6, 7, and 18 were placed together in the same task because node 18 is a fan-in node and node 6 is its dominator in the FCDG. Also, each of the sibling node pairs—(*PH*1, *PE*1), (*PH*2, *PE*2), and (*PH*3, *PE*3)—were merged for convenience, so that for a given `DO` node u, there is at most one task-tree edge with a control condition of the form (u, T). This merging of *preheader* and *postexit* nodes does not incur any loss of useful parallelism, since it involves only pseudo-nodes, which have zero cost.

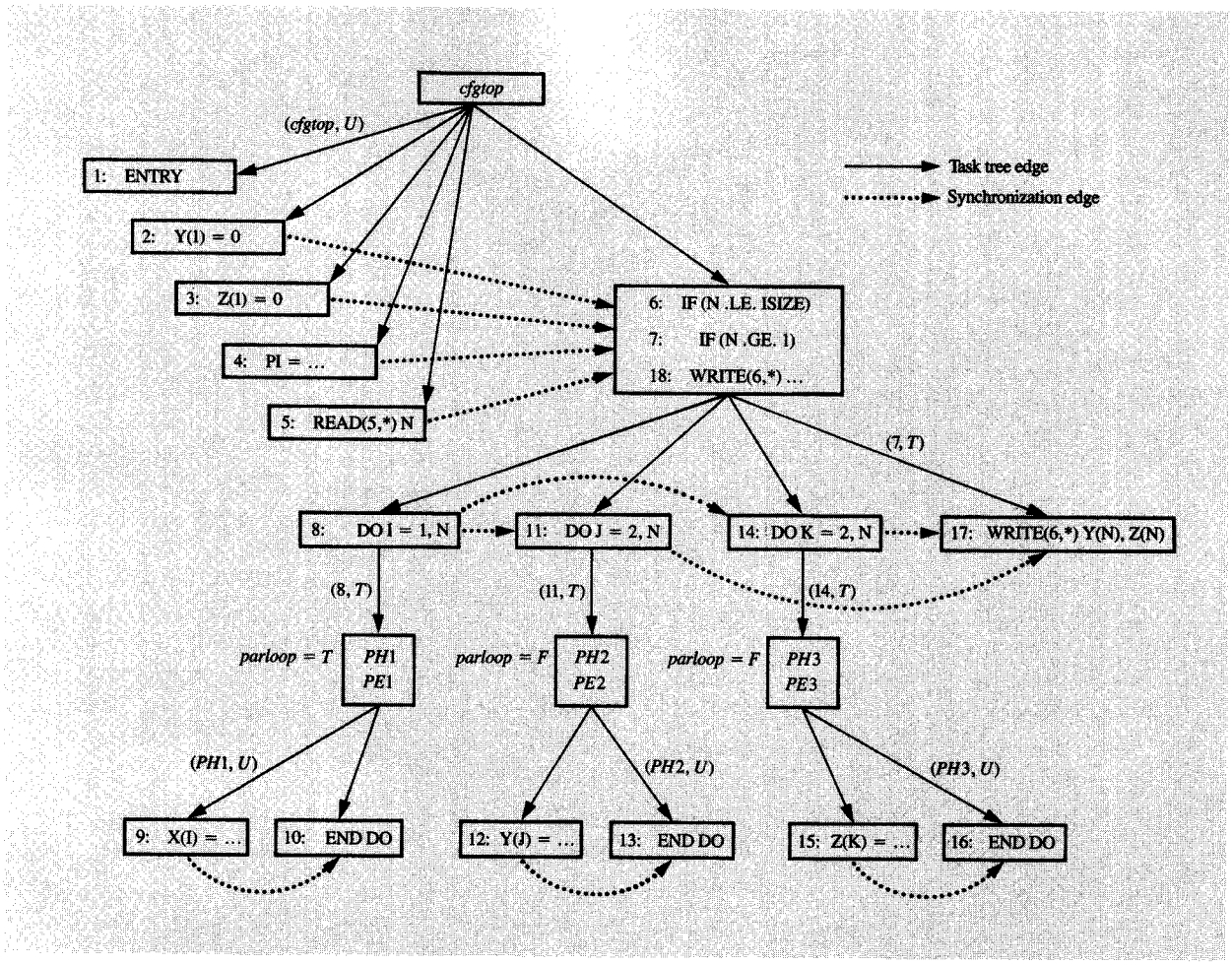

Figure 3

Initial task tree for program of Figure 1.

The definitions of execution frequencies and execution times in an FCDG (Sections 4 and 5) can now be extended to a task tree as follows:

- $freq[T_a, (u, l)]$ is the relative execution frequency of control condition (u, l) in the task tree, just like $freq(u, l)$ in the FCDG. Usually $freq[T_a, (u, l)] = freq(u, l)$. The only exception occurs for branch conditions related to a PARALLEL DO construct—i.e., when (u, l) represents a null-iteration-range branch condition. Let ph be the preheader node for the loop represented by node u. In the FCDG, we defined $freq(ph, U) = n$ to be the average number of iterations in the loop with preheader node ph. In the task tree, control condition (u, T) represents the concurrent execution of multiple instances of the child task, and control condition (ph, U) represents the execution of iterations within a single instance of the child task.

Therefore, $freq[T_a, (u, T)] = p$, the average number of execution instances created for the child task, and $freq[taskid(ph), (ph, U)] = n/p$, the average number of iterations executed within one instance of the child task.

For the task tree in Figure 3, $freq[T_8, (8, T)]$ is set equal to 4 (the number of processors), and $freq[T_{PH1}, (PH1, U)]$ is set equal to 800000/4 = 200000, since $parloop(PH1) = T$ identifies a loop that can be executed in parallel. However, $parloop(PH2) = F$ identifies a loop that must be executed serially and therefore has $freq[T_{11}, (11, T)]$ set equal to 1 and $freq[T_{PH2}, (PH2, U)]$ set equal to 800000. As with the FCDG, the task-tree relative edge frequencies can be used to compute total task frequencies (for a single call of the procedure) as follows:

$totfreq[taskid(cfgtop)] = 1,$

$$totfreq(T_b) = \sum_{[T_a, T_b, (u,l)] \in E_T} freq[T_a, (u, l)]\, totfreq(T_a).$$

- Just as we defined $cost(u)$ to be the average local execution time of FCDG node u (i.e., the sum of the primitive operation costs in node u), we define $cost[T_a]$ to be the average "local" execution time of task T_a—i.e., the total execution time of all operations and statements local to task T_a. The term $cost[T_a]$ is obtained as a frequency-weighted sum of the $cost(u)$ values for all FCDG nodes that belong to task T_a:

$$cost(T_a) = \sum_{taskid(u)=T_a} [totfreq(u)/totfreq(T_a)]\, cost(u).$$

  If $totfreq[T_a]$ equals zero, $cost[T_a]$ is also set to zero.
- Finally, $seqtime[T_a]$ is the average sequential execution of task T_a and its children:

$$seqtime(T_a) = cost(T_a) + \sum_{(T_a, T_b, (u,l)) \in E_T} freq[T_a, (u, l)]\, seqtime(T_b).$$

We conclude this section on task-tree initialization by discussing a simple prepartitioning optimization called *threshold merge*. Since the execution time of the partitioning algorithm is a function of the size of the initial task tree, it is sometimes desirable to further reduce the size of the initial task tree for efficiency reasons by using a *granularity threshold value*, $seqtime_{min}$ [2]. The value of $seqtime_{min}$ should be chosen to suit the granularity of the target multiprocessor system (e.g., set $seqtime_{min} = T_{start\text{-}up}$). We use $seqtime_{min}$ to merge tasks in the initial task tree, so as to obtain a smaller task tree that maximizes the number of tasks while ensuring that each task T_a satisfies $seqtime[T_a] \geq seqtime_{min}$ (the only exception occurs when $seqtime[cfgtop] < seqtime_{min}$, in which case the entire procedure is merged into a single task anyway). This merging can be done without using any data dependence information, as follows:

1. Use $seqtime_{min}$ to identify *fringe* nodes in the task tree. Node T_f is a fringe node if and only if $seqtime(T_f) < seqtime_{min}$ and each ancestor T_a of T_f has $seqtime(T_a) \geq seqtime_{min}$.
2. For each descendant T_d of a fringe node T_f, merge task T_d into task T_f, using the *merge_children* transformation described in Section 8.
3. For each fringe node T_f, let $\langle T_1, \cdots, T_k \rangle$ be the sequence of nodes that are identically control dependent with T_f in the task tree. Use a dynamic programming algorithm (similar to the one in [50]) to partition $\langle T_1, \cdots, T_k \rangle$ into the maximum number of subsequences such that each subsequence has $\Sigma_{T_i} seqtime(T_i) \geq seqtime_{min}$. Each subsequence is then merged into a single task, using the *merge_siblings* transformation described in Section 8.

The possibility of using only threshold-merge as a task-partitioning algorithm was investigated for parallel SISAL programs in [51]. The main limitation of that approach is that data dependence information is not used when building the partition.

7. Partition cost function

In this section, we describe how a task partition is *evaluated*. An important feature of our work is that we present a single objective cost function that can be used to compare two different task partitions and decide which one is better. This is in contrast to other work (e.g., [52, 53]) in which the objectives are to maximize parallelism and minimize overhead, but there is no suggestion of how the two should be traded off. The cost function defined in this section is used by the partitioning algorithms in Sections 8 and 9. The cost function can also be used to provide feedback to the user about the estimated parallel execution time and speedup of the partitioned program.

The cost function to be minimized is *partime*, the estimated parallel execution time of the task tree on the target multiprocessor. Though the value of *partime* depends on P (the number of processors available), the complexity of computing *partime* is linear in the size of the task tree and does not depend on P. The rules for computing *partime* in a task tree are more complicated than the rules presented in Section 6 for computing *seqtime*, because *partime* must take the parallelism into account as well. We begin with a few definitions:

- $parfreq[T_a, (u, l)]$ is the relative *parallel frequency* of the task-tree edge from T_a with control condition (u, l). Usually, $parfreq[T_a, (u, l)]$ is the same as the relative sequential frequency for task-tree edges, $freq[T_a, (u, l)]$, defined in Section 6. The only exception occurs for control condition (u, T), where FCDG node u is a PARALLEL DO statement created by the user or the compiler; since this control condition represents the parallel execution of multiple instances of the child task, we set $parfreq[T_a, (u, T)] = 1$.
- $\#forks[T_a, (u, l)]$ is the average *number of forks* performed by task T_a when taking the conditional branch represented by control condition (u, l). If (u, l) corresponds to a PARALLEL DO, $\#forks[T_a, (u, l)] = freq[T_a, (u, l)]$ is the average number of child task execution instances created for the PARALLEL DO. If (u, l) corresponds to a PARALLEL SECTIONS construct, $\#forks[T_a, (u, l)] = \|\{T_b \mid \exists [T_a, T_b, (u, l)] \in E_T\}\|$ is the number of parallel sections created by the PARALLEL SECTIONS construct.

- $\#waits(T_a) = \|wait\_pred(T_a)\|$ is the average *number of waits* performed by task T_a in a PARALLEL SECTIONS construct.
- $\#signals(T_a) = \|wait\_succ(T_a)\|$ is the average *number of signals* performed by task T_a in a PARALLEL SECTIONS construct.
- $labels(T_a) = \{(u, l) \mid \exists[T_a, T_b, (u, l)] \in E_T\}$ is the set of control conditions (labels) emanating from task T_a in E_T.
- $ovhd(T_a)$ is the total overhead for task T_a, based on the target-multiprocessor overhead parameters mentioned in Section 2:

$$ovhd(T_a) = T_{start\text{-}up} + \#waits(T_a)T_{wait} + \#signals(T_a)T_{signal} + \sum_{(u,l)\in labels(T_a)} freq[T_a, (u, l)] \times \{T_{parent} + \#forks[T_a, (u, l)]T_{child}\}.$$

- The fork-join overhead incurred by task T_a is based on the T_{parent} and T_{child} overhead parameters defined in Section 2. The synchronization overhead incurred by task T_a is assumed be directly proportional to the number of predecessor and successor tasks of T_a in *wait_pred* and *wait_succ* respectively. If the synchronization overhead is further optimized (e.g., by using counting semaphores as in [54]), then the reduced number of synchronization operations should be used instead of *#waits* and *#signals*. We also define $total\_ovhd = \Sigma_{task T_a} totfreq(T_a)ovhd(T_a)$ to be the total overhead incurred by all tasks in the task tree.
- $total\_time(T_a)$ is the average *total* sequential execution time (including overhead) of task T_a and its children. It is computed like $seqtime(T_a)$, which was defined in Section 6:

$$total\_time(T_a) = cost(T_a) + ovhd(T_a) + \sum_{(u,l)\in labels(T_a)} freq[T_a, (u, l)] \sum_{[T_a,T_b,(u,l)]\in E_T} total\_time(T_b).$$

With the above definitions, the following rules show how *partime* can be recursively computed for all tasks by a bottom-up traversal of the task tree, using two intermediary cost values for control conditions, *label_critpath* and *label_partime:*

- *label_critpath.* For any control condition (u, l) in the task tree, compute $label\_critpath[T_a, (u, l)]$, the *critical path length* of the child tasks with control condition (u, l) and parent task T_a (task T_a is uniquely defined, because a control condition must have a unique parent task in the task tree). If (u, l) represents a PARALLEL SECTIONS construct, the *label_critpath* value is set to the critical path length of the precedence graph defined by the *wait_pred* and *wait_succ* relations, computed using $partime(T_c)$ as the execution time for child task T_c. For a PARALLEL DO construct, the *label_critpath* value equals the *partime* value for a single execution instance of its child task.
- *label_partime.* For any control condition (u, l) in the task tree, compute $label\_partime[T_a, (u, l)]$, the estimated *parallel execution time* of the child tasks with control condition (u, l) and parent task T_a:

$$label\_partime[T_a, (u, l)] = \max\{label\_critpath[T_a, (u, l)], \sum_{[T_a,T_c,(u,l)]\in E_T} total\_time(T_c)/P\}.$$

$label\_partime[T_a, (u, l)]$ differs from $label\_critpath(T_a, u, l)$ because it uses $total\_time(T_c)$ and the number of processors, P, to enforce a stronger lower bound on the parallel execution time.
- *partime.* For a given task T_a, compute $partime(T_a)$, the *parallel execution time* of task T_a:

$$partime(T_a) = cost(T_a) + ovhd(T_a) + \sum_{(u,l)\in labels(T_a)} \max\{parfreq[T_a, (u, l)]\, label\_partime[T_a, (u, l)], freq[T_a, (u, l)] \sum_{[T_a,T_b,(u,l)]\in E_T} total\_time(T_b)/P\}.$$

$partime(T_a)$ is obtained from the frequency-weighted sum of the parallel execution times of the control conditions in $labels(T_a)$. As in *label_partime*, the $total\_time(T_a)/P$ term is used to obtain a stronger lower bound on the parallel execution time.

As with the computation of average sequential execution times in Section 5, the computation of *partime* is interprocedural. The *partime* and *total_time* values obtained for the root task of a partitioned procedure are stored as summary information to be used at all call sites of the procedure. This interprocedural approach requires that the procedures be partitioned in a bottom-up traversal of the call graph, so that the summary information for a procedure is available whenever any of its call sites is examined.

The quantity *partime* nicely expresses the trade-off between parallelism and overhead in a way that takes into account the number of processors available on the target multiprocessor. If the granularity of the task partition is too fine, *partime* will be large because of excessive overhead (defined by the *ovhd* values). If the granularity is too coarse, the values of *label_critpath* and *label_partime* will be large because of loss of parallelism, causing *partime* to be large once again. *partime* is usually minimized at some intermediate granularity.

Apart from helping us determine an optimal granularity, the estimated parallel execution time defined by *partime*

can be shown to provide lower and upper bounds on the actual parallel execution time, $partime_{actual}$, of the task partition on the given multiprocessor:

$$(1 - \epsilon)partime \leq partime_{actual} < 2(1 + \epsilon)partime,$$

where ϵ is the relative error in the compile-time estimates of frequencies, execution times, and overhead values. A proof of a similar result was provided in [2]. This result assumes that the tasks are scheduled with no unforced idleness and that the overhead components of different tasks can be executed in parallel. The factor of two is a worst-case upper bound, based on a result due to Graham [55]. This worst-case upper bound is achieved only when the *max* terms are nearly equal in the calculations of *label_partime* and *partime* described above. If, as is usually the case, one term is significantly larger than the other in each *max* operation performed, the upper bound will be $\approx(1 + \epsilon)partime$, without the factor of two [2].

The task-tree *partime* cost function defined in this paper is an extension of the $F(\Pi)$ macro-dataflow partition cost function defined in [2, 56].

8. Partitioning algorithm

• *Description of algorithm*

The partitioning algorithm attempts to minimize the *partime* value of a given procedure. It starts with the initial task tree defined in Section 6 and iteratively merges tasks on the basis of overhead and critical-path-length values. Tasks are merged using two primitive merge transformations on the task tree:

1. *merge_siblings*(T_a, T_b) merges tasks T_a and T_b, which must have the same parent task and control condition in the task tree. The effect of *merge_siblings* is to combine two separate sections (tasks) from a `PARALLEL SECTIONS` construct into a single section (task).
2. *merge_children*[T_a, (u, l)] merges task T_a with all its child tasks that have control condition (u, l). The effect of *merge_children* is to replace a `PARALLEL DO` with a sequential loop or to replace a `PARALLEL SECTIONS` construct with a sequential execution of the sections.

The iterative merging is continued until the entire procedure is in a single task. The *partime* values are calculated for all intermediate partitions, and the partition with the smallest value of *partime* is reconstructed and passed on to the code generator as the optimized task partition for the current procedure. (Since the task tree is incrementally updated in each iteration, it is more efficient to reconstruct the optimized task partition at the end than to store a copy of the partition in each iteration.)

The general structure of the partitioning algorithm is as follows:

1. Start with the initial task tree defined in Section 6.
2. Repeat steps 3, 4, and 5 until no further merging is possible (i.e., until all nodes have been placed in the same task). Keep track of the best *partime* value obtained among all partitions generated during the following iterations.
3. Pick the task with the largest *average decrease in overhead*. Call the selected task T_a. The average decrease in overhead for a task is calculated by summing the decrease in *total_ovhd* obtained over all possible merging choices for the task and then dividing by the number of merging choices. The average decrease in overhead can be computed efficiently (usually in constant time per merging choice) by examining only the parent, children, and siblings of the task in the task tree [see the definitions of $ovhd(T_a)$ and *total_ovhd* in Section 7].
4. Evaluate the parent, sibling, and child tasks of T_a as candidates for merging with T_a. Of these tasks, pick the one that yields the smallest value of the *critical path length* (*critpath*) of the entire task tree, when it is merged with T_a. For each merging choice, it takes at most time linear in the size of the task tree to evaluate the new critical path length. Call the selected task T_b.
5. Merge tasks T_a and T_b using *merge_siblings* or *merge_children*. Update all the task-tree data structures incrementally in at most time linear in the height of the task tree and the number of sibling tasks of T_a and T_b.
6. When no further merging is possible, reconstruct the partition with the best *partime* value by reinitializing the task tree and repeating steps 3, 4, and 5, until the partition with the best *partime* value is obtained.

The main issue in the partitioning algorithm is the choice of tasks to be merged in each iteration. In step 3, task T_a with the largest *average decrease* in overhead is chosen as the primary candidate for merging. The goal of this step is to obtain the largest reduction in the overhead component of *partime*. In previous work [2], the task with the largest *total* overhead was selected. However, experience has shown that selection of the task with the largest average decrease in overhead usually leads to a larger overhead reduction. In step 4, the task that yields the smallest critical path length when merged with T_a is chosen. This rule attempts to find a task that can be merged with T_a, while giving up as little parallelism as possible. These choices were also used in previous work on partitioning SISAL programs [2, 56, 57].

An alternative rule that was considered for the partitioning algorithm was to choose the pair of tasks (T_a, T_b) that yields the smallest value of *partime* in each iteration (i.e., a greedy algorithm to minimize *partime*). In practice, however, the greedy algorithm turned out to be a poor choice, because it would often return the sequential

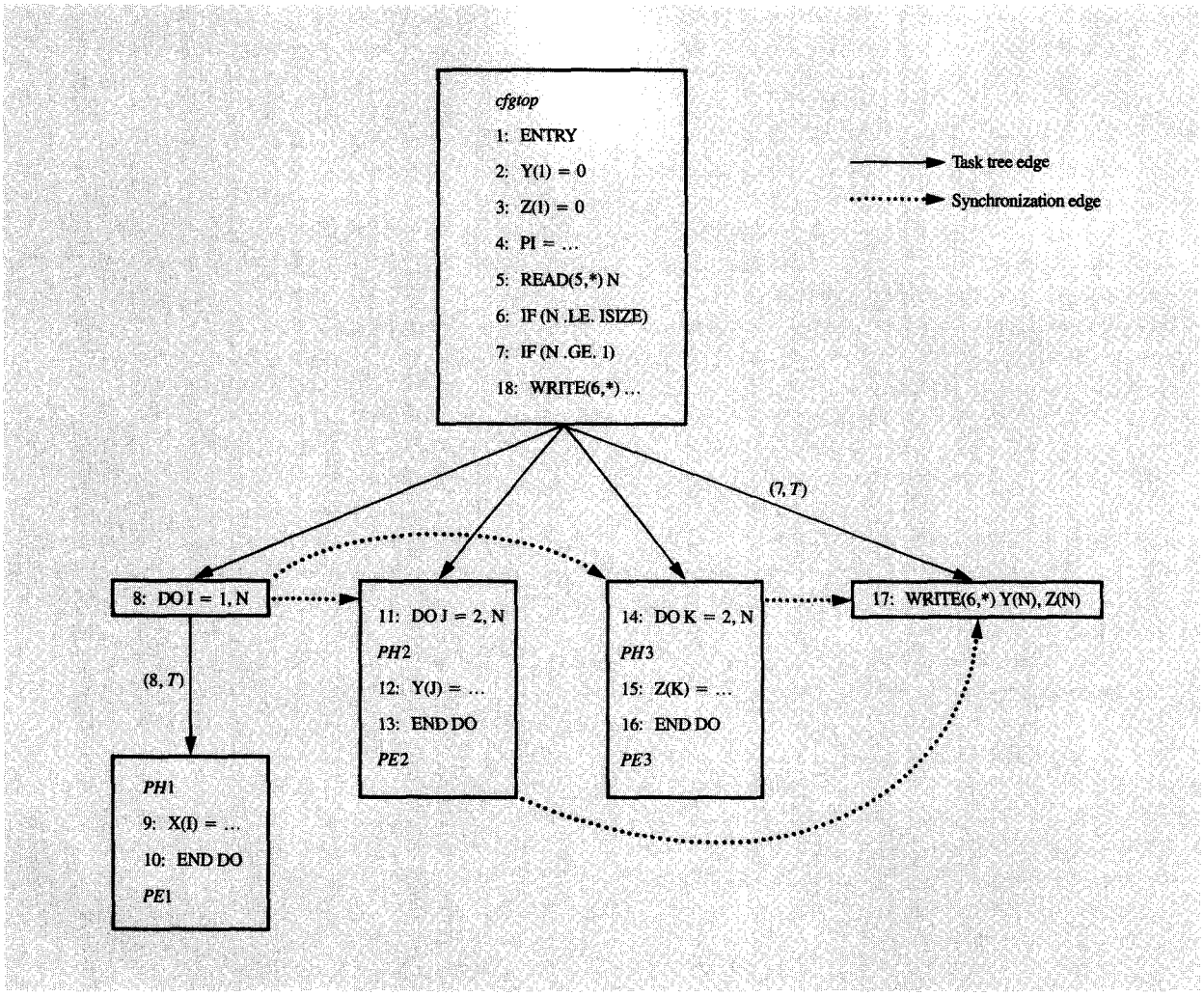

Figure 4

Final task tree for program of Figure 1.

partition as the best partition (due to the large overhead of fine-grain parallelism, the sequential partition usually has a smaller *partime* value than that of the initial task tree). Instead, by choosing task T_b as the one that yields the smallest *critical path length* in step 4, we force the partitioning algorithm to explore intermediate partitions with more parallelism and to avoid the sequential partition as long as possible.

The *wait_pred* and *wait_succ* constraints have a subtle influence on which pairs of tasks can be merged by *merge_siblings*(T_a, T_b). In particular, the merge must not introduce a cycle in the waiting constraints. This condition will be satisfied if and only if there is no directed path of length greater than one between T_a and T_b in *wait_pred* or *wait_succ*. In previous work on partitioning SISAL programs for macro-dataflow execution [2, 56], we avoided introducing a cycle by imposing a *convexity constraint* and merging all tasks in the *convex hull* of T_a and T_b in the data dependence graph. In the present work, we merge tasks T_a and T_b only if their convex hull consists of just T_a and T_b. It is easy to prove that any valid set of sibling tasks can be merged by successive calls to *merge_siblings*, with this constraint.

Note that *merge_children*[T_a, (u, l)] merges task T_a with *all* of the T_a child tasks that have control condition (u, l). This all-or-nothing approach was chosen for the sake of simplicity. If we allow a single child task (say, task T_b) to be merged with its parent, we may have to split the original PARALLEL SECTIONS construct into two PARALLEL SECTIONS constructs, one for the T_b predecessor

Table 2 Performance measurements for final task tree of Figure 4.

Task	*freq*	*parfreq*	*cost*	*ovhd*	*total_time*	*partime*
CFGTOP . . .			30008	1260	2.14×10^8	7.27×10^7
8: DO I= . . .	1	1	2	1270	1.37×10^8	3.42×10^7
PH1 . . .	4	1	3.42×10^7	10	3.42×10^7	3.42×10^7
11: DO J= . . .	1	1	3.84×10^7	10	3.84×10^7	3.84×10^7
14: DO K= . . .	1	1	3.84×10^7	10	3.84×10^7	3.84×10^7
17: WRITE . . .	1	1	30000	10	30010	30010

sibling tasks and one for the T_b successor sibling tasks (according to the waiting constraints). Also, for any sibling of T_b (say, T_s) that could be executed concurrently with task T_b, we have to decide which of the two new PARALLEL SECTIONS constructs should contain T_s. Further merges of individual child tasks could lead to further fragmentation of the PARALLEL SECTIONS constructs. These complications are avoided by requiring that all child tasks with the same control condition be merged simultaneously with the parent. It would be easy to extend this rule to allow merging an individual child task with its parent in certain benign cases (e.g., if the task has empty *wait_pred* and *wait_succ* sets). However, an extension for the general case would be more complicated.

A rudimentary worst-case execution-time analysis of the partitioning algorithm now follows. Let N be the number of tasks in the initial task tree. Steps 3, 4, and 5 are performed at most N times, because each iteration reduces the number of tasks by at least one[3]. Step 4 takes constant time if we store the "average decrease in overhead" values for all tasks in a heap data structure. The complexity of maintaining the heap data structure is as follows. For any task, we define *adjacency* as the sum of numbers of parent, sibling, and child tasks. We define A as the maximum adjacency over all tasks in the task tree. Initializing and maintaining the heap takes a total of $O[N(\log N + A)]$ time during the whole algorithm, since it takes $O(A)$ time to compute a new "average decrease in overhead" value. Step 5 takes $O(AH)$ time, where H is the maximum height of the task tree, since A is the number of candidate tasks considered for T_b, and $O(H)$ is the time required to compute a new critical path length value.

Thus, the total execution time complexity is $O[N(\log N + AH)]$. In the best case (i.e., when each task has at most a small, constant number of children and siblings), A is constant and $H = O(\log N)$, yielding an $O(N \log N)$ execution time. In the worst case, $A = O(N)$ and $H = O(N)$, yielding an $O(N^3)$ execution time. Empirical evidence shows that, in practice, the actual execution time is almost always $O(N \log N)$.

• *Partitioning example and experimental results*

Figure 4 shows the final task tree obtained by applying this partitioning algorithm to the initial task tree in Figure 3. The target multiprocessor parameters used (cf. Section 2) were

- P = four processors.
- $T_{\text{start-up}} = 10$ cycles.
- $T_{\text{fork-join}} = 60 + 300k$ cycles, where k is the number of child tasks created (i.e., $T_{\text{parent}} = 60$ and $T_{\text{child}} = 300$).
- $T_{\text{signal}} = T_{\text{wait}} = 0$. For the sake of simplicity, the cost of a waiting synchronization was assumed to be zero.

Table 2 summarizes the *freq*, *parfreq*, *cost*, *ovhd*, *total_time*, and *partime* values of all tasks in the final task tree (the *freq* and *parfreq* entries give the corresponding values for the task-tree edge connecting a task to its parent). The task-tree edge with control condition (8, T) is the only edge with *parfreq* < *freq*; it represents a PARALLEL DO for which four instances of the child task are created at run time. The *partime* value of the root task is 7.27×10^7 cycles; this is the estimated parallel execution time of the entire program. Since the estimated sequential execution time of the program was 2.14×10^8 cycles (Figure 2), the estimated speedup of the partitioned program is $2.14 \times 10^8 / 7.27 \times 10^7 = 2.9$.

The Parallel FORTRAN code generated from the final task tree is shown in **Figure 5**. It contains a PARALLEL SECTIONS construct and a PARALLEL DO construct, as dictated by the final task tree in Figure 4. The estimated sequential execution times from Figure 2 are shown to the right of the output code. (An option in PTRAN is used to control the inclusion of these *seqtime* values in the output code listing.)

To corroborate the estimated execution times with their actual values, we executed the sequential and parallel versions of this example program on an IBM 3090 system[4].

[3] The only case in which a merging iteration does not reduce the number of tasks is when the partitioner considers different values for p, the number of processors to be assigned to a parallel loop. Using the approach from [2], we successively try to set p to the values P, $P/2$, $P/4$, $\cdots$, 1, where P is the number of physical processors available to the user (Section 2). Therefore, the number of iterations could be $N \log (P)$ in the worst case, which can still be assumed to be $O(N)$ in practice if $\log (P)$ is assumed to be a constant.

[4] The measurements were made using the Parallel FORTRAN run-time system [19]. The parallel program used for the measurements was generated by PTRAN but differed slightly from the code shown in Figure 5 because of syntactic differences between the VS FORTRAN 2.5 parallel language [18] and the Parallel FORTRAN language [19]. The measurement version of the program contained the PARALLEL LOOP and PARALLEL CASES keywords instead of PARALLEL DO and PARALLEL SECTIONS, and also contained a call to PXSPIN to set the *busy-wait* tuning parameter BW to 1000.

In both cases, we measured the elapsed wall-clock time. For greater accuracy, we enclosed each program in a sequential loop with 100 iterations. The resulting measurements of elapsed time were 270 s and 94 s for the sequential and parallel versions, respectively. Therefore, the actual speedup obtained was 270/94, which also equals 2.9!

The average sequential execution time measured for a single instance of the example program is 270/100 = 2.7 s. Since the cycle time of the 3090 system is 15 ns, this amounts to an execution time of $2.7/15 \times 10^{-9} = 1.8 \times 10^{8}$ cycles. However, the estimated *seqtime* value in Figure 2 is 2.14×10^{8} cycles, which is an overestimate by about 20%. Similarly, the average parallel execution time of a single instance of the example program is 0.94 s, or $0.94/15 \times 10^{-9} = 6.27 \times 10^{7}$ cycles, whereas the estimated *partime* value in Figure 4 is 7.27×10^{7} cycles, an overestimate of about 16%. Further experimentation is necessary to evaluate the accuracy of execution-time estimates for different programs.

Table 3 contains parallel-execution-time measurements for a small collection of programs from the GENESIS [58], PERFECT [59], and SPEC [60] benchmark sets. These programs are much larger than the example program discussed in Figures 1–5 and are more representative of real applications. The first column of numbers contains the sequential execution times for these programs (measured on an IBM 3090J system using the Parallel FORTRAN compiler [19]). The second column of numbers contains the elapsed times obtained by using the automatic parallelization facility of the Parallel FORTRAN compiler and executing the program on four processors. The third column of numbers contains the elapsed times (also on four processors) obtained by using PTRAN as a source-to-source translator followed by the Parallel FORTRAN compiler (with automatic parallelization turned off). Although the automatic parallelization performed by PTRAN resulted in better performance compared to the automatic parallelization performed by the Parallel FORTRAN compiler, the actual performance improvement compared to the sequential code was quite modest. The performance was limited because of serial regions in these programs that were not amenable to automatic parallelization. Similar results have been reported by other researchers [61]. For these programs, the partitioning techniques described in this paper served the purpose of eliminating useless parallelism and avoiding anomalous situations in which the parallelized code runs slower than the sequential code. The partitioning techniques will be more useful for PDGs with larger amounts of parallelism; such PDGs should result from programs in which the serial regions are explicitly parallelized by the user—e.g., by using PARALLEL DO statements or array-language notation.

9. Partitioning for loop-only parallelism

As described in the previous section, the partitioning algorithm deals with loop parallelism and nonloop parallelism in a common framework. The problem of finding a partition with the smallest value of *partime* is NP-complete [2]. The task-partitioning algorithm in Section 8 is an approximation algorithm that works well in practice. It is generally intractable to use an exponential-time optimal algorithm for partitioning task trees from real programs, considering that the initial task tree often contains 500 to 1000 nodes.

However, it is generally tractable to use an exponential-time algorithm for loop-only parallelism, since very few programs have more than ten parallel loops in a loop nest. Exponential-time loop-nest-traversal algorithms are frequently used in commercial vectorizing and parallelizing compilers (e.g., [62]). The PTRAN partitioner performs a "loop-only pre-pass," which identifies the optimal set of loops to be serialized if there is no nonloop parallelism in the program (i.e., if *wait_pred* and *wait_succ* impose a total order rather than a partial order on each set of sibling tasks with the same control condition). Loops that are serialized in this loop-only pre-pass are then marked as sequential before the general partitioning algorithm of Section 8 is performed. The justification for this approach is that if a loop is made sequential when only loop parallelism is considered, it should also be made sequential if nonloop parallelism is considered as well. If the target multiprocessor system supports only loop parallelism, the output of the loop-only pre-pass can be passed directly to the code generator.

The loop-only pre-pass consists of the following steps:

1. Start with the initial task tree defined in Section 6.
2. For each task T_a with control condition $(u, l) \in labels(T_a)$ such that (u, l) does not correspond to a PARALLEL DO construct, perform *merge_children*$[T_a, (u, l)]$. All of the control conditions in the resulting task tree now correspond to parallel loops. In the case of the program of Figure 1, the initial task tree from Figure 3 is collapsed into just two tasks—one containing nodes $\{PH1, 9, 10, PE1\}$, and the other containing all the remaining nodes.
3. For each PARALLEL DO in the task tree, perform a local analysis of execution times and overheads to determine whether it is never profitable to execute the loop concurrently (e.g., when $seqtime < T_{parent}$). If so, perform *merge_children*$[T_a, (u, l)]$, where (u, l) is the control condition of the PARALLEL DO.
4. Remove the root task so as to decompose the task tree into subtrees. Each subtree corresponds to a nest of parallel loops with its root at the task containing the preheader node of the outermost loop in the loop nest.

```
      PROGRAM MAIN                                                    seqtime
      REAL X(800000), Y(800000), Z(800000)
      CONTINUE                                                        2.14+08
      Y(1) = 0                                                        2.00000
      Z(1) = 0                                                        2.00000
      PI = 3.14159265                                                 0.00000
      READ(5,*) N                                                     30000.0
      IF (N .LE. 800000) THEN                                         2.14+08
        IF (N .GE. 1) THEN                                            2.14+08
          PARALLEL SECTIONS
          SECTION 1
C*PTRAN PREFER PROCS(4)
            PARALLEL DO 91002 I = 1, N                                1.37+08
              X(I) = (SIN(PI * I / FLOAT(N)) * (PI / FLOAT(N)))       171.000
91002       CONTINUE                                                  0.00000
          SECTION 2, WAITING(1)                                       0.00000
            DO 91005 J = 2, N                                         3.84+07
              Y(J) = X(J) / 2.0 + Y(J - 1)                            48.0000
91005       CONTINUE                                                  0.00000
          SECTION 3, WAITING(1)
            DO 91008 K = 2, N                                         3.84+07
              Z(K) = X(K) / 3.0 - Z(K - 1)                            48.0000
91008       CONTINUE                                                  0.00000
          SECTION 4, WAITING(2,3)
            WRITE(6,*) Y(N), Z(N)                                     30000.0
          END SECTIONS
        ELSE
91012     CONTINUE                                                    0.00000
          WRITE(6,*) 'Bad value of N'                                 30000.0
        ENDIF                                                         0.00000
      ELSE
        GOTO 91012                                                    0.00000
      ENDIF
      END
```

Figure 5

Output Parallel FORTRAN code generated from final task tree for program of Figure 1.

Table 3 Comparison of sequential and parallel (on four processors) execution times.

Benchmark program	*Elapsed time on an IBM 3090J system* (s)		
	Sequential	*Automatic parallelization by Parallel FORTRAN compiler*	*Automatic parallelization by PTRAN*
CG2BIG (GENESIS)	244	244	90
QCD1V (GENESIS)	899	475	387
XYVEC3 (GENESIS)	262	176	170
TFS (PERFECT)	288	185	162
NASA7 (SPEC)	259	261	178
TOMCATV (SPEC)	218	(terminates prematurely)	130

The nest of parallel loops may have a general tree structure; we do not require the nest to be linear or to consist only of adjacent loops.

5. For each subtree (loop nest), find the optimal subset of parallel loops that should be executed concurrently in order to obtain the smallest *partime* value for the entire loop nest. In the worst case, the execution time of this step will be $O(L \times 2^L)$, where L is the number of parallel loops in the loop nest.

10. Incorporating loop transformations into the partitioner

The partitioning algorithm described in the previous sections assumes a fixed loop configuration for the input program. However, loop transformations are the driving force behind many of today's vectorizing and parallelizing compilers [25]. Various loop transformations have been defined over the years in order to deal with different optimization goals. Loop transformations such as

interchange, reversal, skewing, blocking, coalescing, and parallelization can change only the execution order of iterations in a perfect loop nest and are known as *iteration-reordering* loop transformations [63]. Other loop transformations, such as distribution, fusion, alignment, unrolling, and peeling, can change the execution order of iterations and statements in the loop body and are known more generally as *statement-reordering* loop transformations.

Loop transformations are usually applied to a single loop nest at a time. The definition of a loop nest assumed in loop transformations is usually more restrictive than the definition in Section 9. (E.g., loop transformations usually assume that all loops are DO loops, that the loop nests are linear, and that the loops are perfectly nested.) Even for this restricted model, it is highly intractable to determine the best choice of loop transformations for a given loop nest. Many transformation systems (e.g., Parafrase [64]) perform loop transformations on all loop nests in some predetermined sequence. Because of the inherent complexity of the loop-transformation problem, some transformation systems (e.g., ParaScope [65, 66]) allow the programmer to interactively specify the desired sequence of loop transformations.

In the previous sections, we described how the PTRAN partitioner addresses the problem of selecting an optimized task partition by using a global framework for average execution times and frequencies. Incorporating loop transformations within this global framework would surely make the transformation problem harder than that for a single loop nest. However, we perform two important simplifications that make the combined problem of task partitioning and loop transformations tractable, while still yielding a good optimized solution to the general problem:

1. *Divide and conquer* We decompose the entire partitioning and loop-transformation process into three stages—prepartitioning, partitioning, and postpartitioning. We then examine each loop transformation separately and carefully decide at which stage it should be performed, trying to place as few loop transformations in the partitioning stage as possible. For instance, it is appropriate to perform loop vectorization in the prepartitioning stage since it is a transformation with a large number of constraints. Once the decision to vectorize has been made in the prepartitioning stage, it will not be revoked by the later stages; thus the expense of backtracking is avoided.
2. *Focus on hot spots* Our global execution-time framework can be used to identify the *hot spots* in the input program, i.e., the regions which dominate the program execution time. We need not spend any effort on partitioning or transforming regions with negligible execution time, since doing so will have little or no impact on the performance of the optimized program. We discuss one form of hot-spot analysis in Section 6, where the granularity threshold value $seqtime_{min}$ is used to control the size of the initial task tree. Similarly, we can perform parallelizing loop transformations only on loops whose total execution time contribution is larger than some factor (say, ϵ) of the program sequential execution time. It is important to make ϵ depend on the number of processors, P, because the execution time of a region may become more significant as the remainder of the program is parallelized. In [2], we proposed a value of $\epsilon = 0.01/P$ when performing node expansion for compile-time scheduling.

The remainder of this section is organized as follows. The next subsection briefly discusses some of the loop transformations that are suitable for the prepartitioning stage. The following subsection discusses how the *loop fusion* transformation can be incorporated into the partitioning stage. The last subsection briefly discusses some of the loop transformations that are suitable for the postpartitioning stage.

- *Prepartitioner loop transformations*

Loop distribution

Most loop-transformation systems (e.g., Parafrase [64]) begin by distributing loops around strongly connected components of the dependence graph [25, 67]. This approach of *full distribution* (also known as π-block partitioning) provides the greatest flexibility in choosing individual loop configurations and task partitions and is a natural candidate for the prepartitioning stage. However, if the final code contains several distributed loops, it may incur excessive overhead because of poor data locality and extra increment-and-test code. After the flexibility of full distribution has been exploited in the prepartitioning stage, we suggest performing loop fusion in the partitioning stage in order to remove (as far as possible) the overheads introduced by loop distribution. *Name-only distribution* [68] is a more coarse-grained form of loop distribution that has also been proposed as a way of obtaining the benefits of loop distribution without excessive overhead.

Locality-improving loop transformations

Recent work has shown how loop transformations, such as interchange, reversal, skewing, and blocking (tiling), can be used to restructure innermost loop nests for improved data-access locality in the cache and other levels of the memory hierarchy [5, 6, 69–71]. These transformations are crucial for efficient use of modern high-performance uniprocessor and multiprocessor systems. More importantly, the performance degradation that results from not performing these transformations becomes more severe

as the number of processors increases in a multiprocessor system [5]. Therefore, it is important to perform such locality-improving loop transformations in the prepartitioning stage.

Loop vectorization

Loop vectorization has been studied in great detail [25, 67]. In fact, the whole area of automatic parallelization was initially motivated by the need for compilers that do a good job of loop vectorization. The decision to vectorize a loop depends on control and data dependences, on hardware constraints (not all operations can be executed in vector mode), and on cost considerations (number of iterations of the loop, memory access costs, etc.) [62]. As mentioned earlier in this section, we suggest that loop vectorization should be performed in the prepartitioning stage because it has a large number of constraints. Loop vectorization is performed by strip-mining the loop on the basis of the hardware vector length [25]; the outer sectioning loop is still eligible for other loop transformations, such as parallelization and interchange.

- *Loop fusion*

As mentioned earlier, we depend on loop fusion to remove (as far as possible) any overheads that are introduced by loop distribution. There is a strong interaction between the considerations for loop fusion and task partitioning, so it is natural to perform loop fusion in the partitioning stage. For instance, loop fusion may tend to fuse two loops together to reduce the number of cache misses, whereas the partitioner may tend to place them in separate tasks for the sake of parallelism (thus making them ineligible for fusion). By incorporating the loop-fusion decisions in the partitioner, we can trade loop-fusion benefits with parallelism benefits in a common framework. Also, the structure of our partitioning algorithm lends itself quite nicely to loop fusion. We start with a fine-grained initial task tree and with fully distributed loops. As we merge tasks, we can also decide how the loops within a task should be fused.

Most commercial vectorizing and parallelizing compilers attempt to fuse only adjacent loops that originate from the same source statement (e.g., by distribution of a `DO` loop or by translation of a Fortran 90 `WHERE` construct into multiple loop nests). Our proposed loop-fusion algorithm is more general in that it may fuse any set of loops that are potentially fusible; the loops may be nonadjacent and may not even have been derived from the source statement.

To incorporate loop fusion within the partitioning algorithm, we employ a procedure that selects an optimized fusion configuration for all loops in a given task. This procedure is described in [72] and uses an extended version of the max-flow/min-cut algorithm [73]. As each iteration of the partitioning algorithm considers different pairs of tasks to merge, the fusion procedure can be used to estimate the fusion benefits that will result from each possible merge. These fusion costs are then incorporated into the global *partime* value used to identify the best partition. This approach enforces the constraint that only loops in the same parallel task are eligible for fusion.

- *Postpartitioner loop transformations*

After the partitioner has determined how the program should be decomposed into parallel tasks, it is important to perform intratask code optimizations so as to improve the sequential execution times of the tasks as well. Our goal is to obtain speedup via parallelism without slowing down the individual tasks due to reduced optimization. Loop transformations that are used to improve the sequential execution time of a program should be performed in the postpartitioning stage. Loop unrolling is an important technique for exposing more instruction-level parallelism. Scalar replacement [74] is a technique for replacing array accesses with a scalar temporary in order to obtain an improved register allocation. Such transformations can be applied profitably to intratask computations.

11. Related work

The general problem of determining the optimal granularity of program decomposition has been addressed in other work, as discussed below. However, these approaches differ from the work presented in this paper by assuming a more restrictive program model and/or a more restrictive parallel execution model. There has also been much previous work in the area of graph partitioning to minimize intercluster communication. While it may be possible to use some of those techniques for partitioning programs to minimize communication overhead, the graph model assumed is usually too abstract for those results to be applicable to program graphs.

Some of the ideas in our work on partitioning PDGs for fork-join execution evolved from our previous work on partitioning SISAL programs for macro-dataflow execution [2, 56]. However, there are fundamental differences between the two pieces of work because of differences in the language model (FORTRAN is an imperative language, whereas SISAL is an applicative, single-assignment language) and in the parallel execution model (fork-join vs. macro-dataflow). In this paper, the FCDG is used to uncover the structure of the FORTRAN program from the control flow graph, to store average execution frequencies and average execution times, and to provide the basis for determining the final task partition. In SISAL [41], the graphical program representation (IF1) is derived from the syntax of the program; IF1 cannot express imperative and unstructured language features, such as those found in FORTRAN. Also, the target model in the present work is a structured fork-join parallel program, as opposed to the

macro-dataflow execution model assumed in [2, 56]. Macro-dataflow is a more restrictive model because it does not allow the possibility of a parent macro-actor creating other macro-actors and then suspending itself until their completion. Recently, we have extended the SISAL partitioner to support a fork-join parallel execution model [57], but the differences in the language models (FORTRAN vs. SISAL) still remain.

Apart from the fundamental differences in parallel language and execution models, the work described in this paper contains several novel extensions and improvements, compared with our earlier work in [2, 56]. Section 4 describes optimizations for efficient execution profiling that are based on the FCDG; these optimizations could also be used in a syntax-based representation such as IF1. Section 5 mentions how the approach to computing average execution times can be extended to compute variance as well. Compile-time estimation of variance helps quantify the "execution-time predictability" of a region of code and provides confidence intervals on the estimated execution times. As discussed in Section 7, the *partime* cost function is a refinement of the cost function defined in [2, 56] and has been found to yield better task partitions. There are important differences between the partitioning algorithm in Section 8 and the earlier algorithm in [2]. In Section 8, the first task selected as a merging candidate is the one with the largest *average decrease* in overhead; in [2], it is the one with the largest *total* overhead. In the merging of sibling tasks, Section 8 considers only task pairs (T_a, T_b) such that $convexhull(T_a, T_b) = \{T_a, T_b\}$; in [2], any two sibling tasks could be merged by merging the entire set of tasks $convexhull(T_a, T_b)$ in a single step. Also, the algorithm in Section 8 returns the partition found to have the smallest *partime* value, whereas the algorithm in [2] returns the partition found to have the smallest $F(\Pi)$ value (a different cost function). Finally, Sections 9 and 10 describe important loop transformations and optimizations that were not considered in [2] and describe how they are incorporated into the general partitioning framework.

McCreary and Gill propose a graph-decomposition technique for decomposing a dataflow graph into "clans" (tasks) [75]. The program representation is assumed to be a weighted acyclic dataflow graph. The decomposition technique is based on a result due to Ehrenfeucht and Rosenberg [76] that is used to parse the dataflow graph into a hierarchy of clans. There are three kinds of nodes in the parse tree: linear, primitive, and independent. Parallelism is exploited only within independent nodes. The parse tree can be viewed as a task tree, with a fork operation at each independent node. To determine how much parallelism should be exploited at an independent node, it is further assumed that its child nodes have identical execution and communication times. Moreover, the acyclic dataflow-graph representation is restrictive because it does not include loops and conditionals, and the execution-time model is restrictive because all children of an independent node are assumed to have the same execution time.

Bokhari uses a polynomial-time *sum-bottleneck-path* algorithm for the optimal assignment (partitioning) of modules of a parallel program to processors in a host-satellite multicomputer system [77, 78]. The algorithm finds the optimal solution for two restricted parallel program structures: a multiple-chain structure and a single-tree structure. The objective function to be minimized has the form

$$\max_{1 \le i \le P} \left\{ \begin{array}{l} \text{Max (total execution time of all modules} \\ \qquad \text{assigned to processor } i, \\ \qquad \text{communication time incurred by processor } i \\ \qquad \text{over the host-satellite link)} \end{array} \right\}$$

This is a simple cost function that is really a lower bound on the execution time of the *bottleneck* processor. Note that this cost function ignores intermodule precedence constraints. Also, the module assignments (partitions) have a very restricted structure: A chain can be partitioned only into contiguous subchains, and a tree can be mapped only onto a single-level processor tree in which the root processor must be the host system. With these simplifications and restrictions, the partitioning problem is no longer NP-complete and can be solved optimally in polynomial time. Though the restricted problem structure is useful for certain signal-processing and image-processing applications, it is far removed from the general program model assumed in our work, namely the PDG. The PDG captures the control and data dependences that occur in a program with arbitrary control flow and data flow. In contrast, the program model assumed in [77, 78] is a static collection of modules with a restricted interconnection structure.

The chain-partitioning algorithm from [77, 78] was also independently presented by Polychronopoulos in [79, 80]. Both versions of the chain-partitioning algorithm take $\theta(m^3n)$ time and $\theta(m^3n)$ space to find an optimal chain partition for m modules on n processors. In other work, we described a dynamic-programming algorithm (similar to the one in [50]) for the same problem that finds an optimal chain partition in $O(m^2n)$ time and $\theta(m^2)$ space. Experimental measurements show that the average execution time of the algorithm is usually $O(m^2)$ in practice, rather than $O(m^2n)$.

We have also done some work in the area of static (compile-time) scheduling [2, 48]. This approach first expands (unrolls) the program graph of the main procedure and then maps computations onto processors in order to

minimize the estimated parallel execution time (which includes synchronization and communication overhead). The output of the scheduler is a partition of the program graph into threads (one per processor) that may have arbitrary synchronizations and communications among themselves. This execution model differs from the fork-join execution model because threads cannot be created dynamically during program execution. Also, this approach was developed for SISAL programs and has all the restrictions, discussed earlier, due to the language model. It would be interesting to generalize the compile-time scheduling work for a FORTRAN-language model, using some of the extensions presented in this paper (the FCDG, the initial task tree, etc.). In fact, there is now a growing interest in using a static scheduling approach for executing FORTRAN programs on distributed-memory multiprocessor systems [81].

The area of optimizing loop-only parallelism has received considerable attention in the literature [25]. The framework presented in this paper includes both loop parallelism and nonloop parallelism. Future advances in the area of optimizing loop parallelism can be incorporated into the partitioning framework using the techniques discussed in Section 10.

The general problem of determining the optimal chunk size of a parallel loop in the presence of overhead and variance was studied by Kruskal and Weiss [47]. The chunking approach from [47] has been extended by Flynn and Hummel [22] to a sequence of multiple batches, each batch using a progressively smaller chunk size than the previous batch (this approach is called *factoring*). Rules for approximating the optimal number of batches and the optimal chunk size for each batch are also given in [22]. Experimental results for the factoring approach are given in [82]. Both the chunking [47] and the factoring [22] approaches are performed on the basis of estimated loop iteration execution times, variances, and scheduling-overhead values. Since these cost values are an inherent part of the partitioning framework presented in this paper, the chunking and factoring approaches to run-time scheduling should benefit from the compile-time cost estimates determined by the partitioner.

In the area of graph partitioning to minimize intercluster communication, the partitioning algorithm due to Kernighan and Lin [83] is well known. The problem addressed there is that of finding an optimal partition of a graph into k clusters of equal size, such that the sum of the intercluster edge weights is minimized; [83] contains an approximation algorithm for this problem. This algorithm assumes that all node weights are equal. To handle arbitrary node weights, [83] suggests replacing a node of weight W with W nodes of weight 1, mutually connected by edges of sufficiently high cost. Though this replacement correctly models a node with a nonunit weight, it is achieved at the expense of increasing the time and space requirements of the algorithm.

An optimal algorithm for partitioning trees was given by Lukes in [84]. The problem addressed is that of partitioning a tree into clusters so that the sum of the node weights in each cluster is less than a given size and the sum of the intercluster edge weights is minimized. This objective function is more general than that of Kernighan and Lin because nodes may have arbitrary weights and the clusters need not be of equal size. However, the partitioning algorithm is more restrictive because it can be applied only to trees instead of to general graphs. Also, a structural constraint is placed on the tree partition similar to that of partitioning a chain into contiguous subchains: An arbitrary order and direction is imposed on the nodes and edges of the tree, and only adjacent nodes (according to this order) may belong to the same cluster.

12. Conclusions

In this paper, we have presented a general interprocedural framework for partitioning a program dependence graph into parallel tasks. The framework is novel in that it exploits both loop parallelism and nonloop parallelism, supports both automatically detected and user-specified parallelism, and uses parameterization to specify the different multiprocessor systems, all within a common framework and environment. This approach can be used to run the same parallel program on a variety of shared-memory multiprocessors. Such a system greatly simplifies the problems of creating, debugging, and porting efficient parallel programs on different multiprocessor systems. Though the partitioning techniques have been implemented in PTRAN, the basic approach is general and is applicable to any environment for which a program dependence graph representation can be obtained.

Acknowledgments

I am indebted to the other members of the PTRAN group—Frances Allen, Michael Burke, Philippe Charles, Jong-Deok Choi, Ron Cytron, Jeanne Ferrante, Edith Schonberg, and David Shields—for providing me a stimulating research environment for this work. I thank Raymond Ellersick for providing execution-time cost estimates from the VS FORTRAN compiler, and Clare Hartnett for providing performance tuning feedback on the RP3.

3090 is a trademark of International Business Machines Corporation.

References

1. J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Program. Lang. & Syst.* **9**, No. 3, 319–349 (July 1987).

2. Vivek Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessors," *Research Monographs in Parallel and Distributed Computing*, MIT Press, Cambridge, MA, 1989.
3. Vivek Sarkar, "The PTRAN Parallel Programming System," *Parallel Functional Programming Languages and Compilers*, Boleslaw K. Szymanski, Ed., Addison-Wesley Publishing Co., Reading, MA, 1991, pp. 309–391.
4. H. Zima, H.-J. Bast, and M. Gerndt, "SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization," *Parallel Computing* **6**, 1–18 (1986).
5. Michael E. Wolf and Monica S. Lam, "A Data Locality Optimization Algorithm," *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, Toronto, June 1991, pp. 30–44.
6. Jeanne Ferrante, Vivek Sarkar, and Wendy Thrash, "On Estimating and Enhancing Cache Effectiveness," *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991. To be published by Springer-Verlag.
7. William Baxter and J. R. Bauer III, "The Program Dependence Graph in Vectorization," *Proceedings of the Sixteenth ACM Principles of Programming Languages Symposium*, January 11–13, 1989, Austin, TX, pp. 1–11.
8. Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe, "Gated Single-Assignment Form: Dataflow Interpretation for Imperative Languages," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 257–271.
9. Micah Beck and Keshav Pingali, "From Control Flow to Dataflow," *Technical Report TR 89-1050*, Department of Computer Science, Cornell University, Ithaca, NY, October 1989.
10. Rajiv Gupta and Mary Lou Soffa, "A Reconfigurable LIW Architecture and Its Compiler," *Proceedings of the 1987 International Conference on Parallel Processing*, St. Charles, IL, August 1987.
11. R. Gupta and M. L. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Trans. Software Eng.* **16**, No. 4, 421–431 (April 1990).
12. S. Horwitz, J. Prins, and T. Reps, "Integrating Non-Interfering Versions of Programs," *ACM Trans. Programming Lang. & Syst.* **11**, No. 3, 345–387 (July 1989).
13. Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *J. Parallel & Distr. Computing* **5**, No. 5, 617–640 (1988).
14. Frances Allen, Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, and Vivek Sarkar, "A Framework for Determining Useful Parallelism," *Proceedings of the ACM 1988 International Conference on Supercomputing*, St. Malo, France, July 1988, pp. 207–215.
15. Ron Cytron, Jeanne Ferrante, and Vivek Sarkar, "Experiences Using Control Dependence in PTRAN," *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua, Eds., MIT Press, Cambridge, MA, 1990, pp. 186–212.
16. Vivek Sarkar, "Determining Average Program Execution Times and Their Variance," *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, Portland, OR, July 1989, pp. 298–312.
17. P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Trans. Software Eng.* **SE-1**, 199–206 (June 1975).
18. *IBM VS FORTRAN Version 2 Release 5 Language and Library Reference*, Order No. SC26-4221-7; available through IBM branch offices.
19. *IBM Parallel FORTRAN Language and Library Reference*, Order No. SC23-0431-0; available through IBM branch offices.
20. G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, 1985, pp. 764–771.
21. Constantine D. Polychronopoulos and David J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. Computers* **C-36**, No. 12, 1425–1439 (December 1987).
22. Lawrence E. Flynn and Susan Flynn Hummel, "Scheduling Variable-Length Parallel Subtasks," *Research Report RC-15492*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1990.
23. S. F. Hummel and E. Schonberg, "Low-Overhead Scheduling of Nested Parallelism," *IBM J. Res. Develop.* **35**, No. 5/6, 743–765 (1991, this issue).
24. D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Conference Record of the 8th ACM Symposium on Principles of Programming Languages*, Williamsburg, VA, 1981, pp. 207–218.
25. Michael J. Wolfe, "Optimizing Supercompilers for Supercomputers," *Research Monographs in Parallel and Distributed Computing*, MIT Press, Cambridge, MA, 1989.
26. L. Lamport, "The Parallel Execution of DO Loops," *Commun. ACM* **17**, 83–93 (February 1974).
27. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., Reading, MA, 1986.
28. Ron Cytron, Jeanne Ferrante, and Vivek Sarkar, "Compact Representations for Control Dependence," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 337–351.
29. Michael Burke and Ron Cytron, "Interprocedural Dependence Analysis and Parallelization," *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, July 1986, pp. 162–175.
30. Wilson C. Hsieh, "Extracting Parallelism from Sequential Programs," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1988.
31. Ron Cytron, Michael Hind, and Wilson Hsieh, "Automatic Generation of DAG Parallelism," *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989, pp. 54–68.
32. Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh, "Automatic Generation of Nested, Fork-Join Parallelism," *J. Supercomputing* **2**, No. 3, 71–88 (July 1989).
33. J. T. Schwartz and M. Sharir, "A Design for Optimizations of the Bitvectoring Class," *Courant Computer Science Report No. 17*, Courant Institute of Mathematical Sciences, New York University, September 1979.
34. Michael Burke, "An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis," *ACM Trans. Program. Lang. & Syst.* **12**, No. 3, 341–395 (July 1990).
35. Matthew S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, Inc., 1977.
36. Susan L. Graham and Mark Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis," *J. ACM* **23**, No. 1, 172–202 (January 1976).
37. F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Commun. ACM* **19**, No. 3, 137–147 (March 1976).

38. F. E. Allen, "Control Flow Analysis," *ACM SIGPLAN Notices* **19**, No. 6, 13–24 (1970).
39. Robert Tarjan, "Testing Flow Graph Reducibility," *J. Computer & Syst. Sci.* **9**, No. 3, 355–365 (December 1974).
40. C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Commun. ACM* **9**, No. 5, 366–371 (May 1966).
41. J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas, "SISAL: Streams and Iteration in a Single Assignment Language, Reference Manual Version 1.2," *Technical Report No. M-146, Rev. 1*, Lawrence Livermore National Laboratory, Livermore, CA, 1985.
42. John T. Feo, David C. Cann, and Rodney R. Oldehoeft, "A Report on the SISAL Language Project," Technical report, Lawrence Livermore National Laboratory, Livermore, CA, January 1990.
43. Arvind, M. L. Dertouzos, R. S. Nikhil, and G. M. Papadopoulos, "Project Dataflow: A Parallel Computing System Based on the Monsoon Architecture and the Id Programming Language," *Computation Structures Group Memo 285*, MIT Lab for Computer Science, Cambridge, MA, March 1988.
44. Aaron J. Goldberg and John L. Hennessy, "Performance Debugging Shared Memory Multiprocessor Programs with MTOOL," *Proceedings of the Supercomputing '91 Conference*, Albuquerque, NM, November 1991, pp. 481–490.
45. Thomas Ball and James R. Larus, "Optimally Profiling and Tracing Programs," *Proceedings of the ACM Symposium on Principles of Programming Languages*, Albuquerque, NM, 1992, pp. 59–70.
46. P. Bose, "Early Performance Estimation of Super Scalar Machine Models," *Proceedings of the IEEE International Conference on Computer Design*, Cambridge, MA, October 1991, pp. 388–391.
47. Clyde Kruskal and Alan Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Trans. Software Eng.* **SE-11**, No. 10, 1001–1016 (October 1985).
48. Vivek Sarkar and John Hennessy, "Compile-Time Partitioning and Scheduling of Parallel Programs," *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986, pp. 17–26.
49. J. Ferrante and M. E. Mace, "On Linearizing Parallel Code," *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages*, January 1985, 179–189.
50. Vivek Sarkar, "Instruction Reordering for Fork-Join Parallelism," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 322–336.
51. V. Sarkar, S. Skedzielewski, and P. Miller, "An Automatically Partitioning Compiler for SISAL," *CONPAR 88*, Cambridge University Press, Great Britain, 1988, pp. 376–383.
52. M. L. Campbell, "Static Allocation for a Dataflow Multiprocessor," *Proceedings of the 1985 International Conference on Parallel Processing*, St. Charles, IL, August 1985, pp. 511–517.
53. P. Hudak and B. Goldberg, "Serial Combinators: Optimal Grains of Parallelism," *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985, pp 382–399.
54. Vivek Sarkar, "Synchronization Using Counting Semaphores," *Proceedings of the ACM 1988 International Conference on Supercomputing*, St. Malo, France, July 1988, pp. 627–637.
55. R. L. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM J. Appl. Math.* **17**, No. 2, 416–429 (March 1969).
56. Vivek Sarkar and John Hennessy, "Partitioning Parallel Programs for Macro-Dataflow," *ACM Conference on Lisp and Functional Programming*, Cambridge, MA, August 1986, 202–211.
57. Vivek Sarkar and David Cann, "POSC—A Partitioning and Optimizing SISAL Compiler," *Proceedings of the ACM 1990 International Conference on Supercomputing*, Amsterdam, the Netherlands, June 1990, pp. 148–163.
58. A. J. G. Hey and C. J. Scott, "Report of the State-of-the-Art and Evaluation Workpackage," Technical report, Esprit-2 project P2447 (Genesis pre-study), Southampton University, England, June 1989.
59. M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin, "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers, the Performance Evaluation Club (Perfect)," *Technical Report No. 827*, University of Illinois Center for Supercomputing Research and Development, November 1988.
60. "Systems Performance Evaluation Cooperative," K. Mendoza, Ed., *SPEC Newsletter* **1**, August 1989.
61. Rudolf Eigenmann and William Blume, "An Effectiveness Study of Parallelizing Compilers," *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1991, Vol. II, pp. 17–25.
62. Randolph G. Scarborough and Harwood G. Kolsky, "A Vectorizing FORTRAN Compiler," *IBM J. Res. Develop.* **30**, No. 2, 163–171 (March 1986).
63. Vivek Sarkar and Radhika Thekkath, "A General Framework for Iteration-Reordering Loop Transformations," *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, June 1992. (To appear).
64. D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Proceedings of CompSAC 80 (Fourth International Computer Software and Applications Conference)*, Chicago, October 1980, pp. 709–715.
65. K. Kennedy, K. S. McKinley, and C. Tseng, "Analysis and Transformation in the ParaScope Editor," *Proceedings of the ACM 1991 International Conference on Supercomputing*, Cologne, Germany, June 1991, pp. 433–447.
66. K. Kennedy, K. S. McKinley, and C. Tseng, "Interactive Parallel Programming Using the ParaScope Editor," *IEEE Trans. Parallel & Distr. Syst.* **2**, No. 3, 329–341 (July 1991).
67. John R. Allen, "Dependence Analysis for Subscripted Variables and Its Application to Program Transformation," Ph.D. thesis, Rice University, Houston, TX, 1983.
68. Walid Abu-Sufah, "Improving the Performance of Virtual Memory Computers," Ph.D. thesis, University of Illinois at Urbana-Champaign, 1979.
69. Francois Irigoin and Remi Triolet, "Supernode Partitioning," *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, 1988, pp. 319–329.
70. Kyle Gallivan, William Jalby, and Dennis Gannon, "On the Problem of Optimizing Data Transfers for Complex Memory Systems," *Proceedings of the ACM 1988 International Conference on Supercomputing*, St. Malo, France, July 1988, pp. 238–253.
71. Allan K. Porterfield, "Software Methods for Improvement of Cache Performance on Supercomputer Applications," Ph.D. thesis, Rice University, Houston, TX, May 1989.
72. Guang R. Gao, Russell Olsen, Vivek Sarkar, and Radhika Thekkath, "Collective Loop Fusion by Array Contraction," *Proceedings of the Fifth Workshop on*

Languages and Compilers for Parallel Computing, New Haven, CT, August 1992, to appear.
73. Robert Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, 1983.
74. David Callahan, Steve Carr, and Ken Kennedy, "Improving Register Allocation for Subscripted Variables," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 53–65.
75. C. McCreary and H. Gill, "Automatic Determination of Grain Size for Efficient Parallel Processing," *Commun. ACM* **32**, No. 9, 1073–1078 (September 1989).
76. A. Ehrenfeucht and G. Rosenberg, "A Decomposition Theorem for 2-Structures," *Technical Report No. CU-CS-328-86*, University of Colorado, Boulder, 1986.
77. Shahid H. Bokhari, *Assignment Problems in Parallel and Distributed Computing*, Kluwer Academic Publishers, Norwell, MA, 1987.
78. Shahid H. Bokhari, "Partitioning Problems in Parallel, Pipelined, and Distributed Computing," *IEEE Trans. Computers* **37**, 48–57 (January 1988).
79. Constantine D. Polychronopoulos, "On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems," *CSRD Report No. 595*, Center for Supercomputing Research and Development, University of Illinois, August 1986.
80. Milind Girkar and Constantine Polychronopoulos, "Partitioning Programs for Parallel Execution," *Proceedings of the ACM 1988 International Conference on Supercomputing*, St. Malo, France, July 4–8, 1988, pp. 216–229.
81. Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and Min-You Wu, "An Overview of the FORTRAN D Programming System," *Technical Report COMP TR91-154*, Department of Computer Science, Rice University, Houston, TX, March 1991.
82. Susan Flynn Hummel, Edith Schonberg, and Lawrence Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops," *Supercomputing 91*, November 1991.
83. B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Syst. Tech. J.*, pp. 291–307 (February 1970).
84. J. A. Lukes, "Efficient Algorithm for the Partitioning of Trees," *IBM J. Res. Develop.* **18**, 217–224 (May 1974).

Received October 15, 1990; accepted for publication June 17, 1991

Vivek Sarkar *IBM Palo Alto Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304.* Dr. Sarkar received his Ph.D. degree in 1987 from Stanford University. He joined the IBM Thomas J. Watson Research Center in 1987 as a Research Staff Member in the Parallel Translation (PTRAN) project. Since 1990 he has been a Senior Programmer at the IBM Santa Teresa Laboratory, where he works on advanced compiler optimizations for exploiting parallelism and data locality. Dr. Sarkar has published extensively in the area of compiling for parallelism, and is the author of a monograph titled "Partitioning and Scheduling Parallel Programs for Multiprocessor Execution" (MIT Press, 1989). His current research interests are in the areas of program optimization, transformation, and partitioning for exploiting shared-memory/distributed-memory multiprocessor parallelism, instruction parallelism, and data locality.