LINEAR TIME DETECTION OF INHERENT PARALLELISM IN SEQUENTIAL PROGRAMS

Peter L. BIRD *

Computer Science and Engineering Department, College of Engineering and Computing Center, University of Michigan, Ann Arbor, MI 48109, USA

The topological sort can be used for the rapid detection of parallelism in sequential programs. Using this algorithm, one can detect both intrablock and interblock parallelism. The algorithm requires only information normally collected by an optimizing compiler.

1. Introduction

The execution speed of single processor computers is rapidly approaching physical device limitations. One way of achieving increases in the running speed of programs is to partition the problem into independent, concurrently executable computations. While the greatest speed increase should certainly come from algorithms which are structured for parallel execution (using parallel programming languages), there is much a compiler can do to automatically determine concurrently executable computations in sequential programs.

There are four tasks related to the automatic parallelization of sequential programs:

1) *Graph construction*
This involves building a graphical representation of the execution behavior and the data dependency relationships between computations in the program.
2) *Graph modifications*
This is a broad class of operations to modify the graphical representation of the program. It involves the insertion and deletion of both arcs and nodes in the graph. The deletion of arcs in the dependency graph is important to enhance parallelism. Analysis of array and pointer references is one class of graph analysis.
3) *Detection of parallelism*
This includes vectorization (the determination that a looping construction can be mapped to a vector operation) and the identification of operations which may be concurrently executed.
4) *Operation scheduling*
This involves the allocation of resources to perform the computation.

The first two tasks act only upon the abstract program structure. In contrast, the second two tasks map this graph onto the target architecture. The algorithm discussed in this paper is concerned with the third problem, the detection of parallelism, and to some degree with the scheduling of parallel computations. This is not to imply that the other tasks are unimportant. In particular, the number of concurrent operations in a program can be severely limited unless one breaks data dependencies. However, these techniques have been widely studied, and are outside the scope of this paper. See ref. [1] for a presentation of the problems involved in building program graphs; ref. [9] is an introduction to graph modifications.

A partial order is commonly used for representing the data dependency relations between the operations of a basic block of a program *. Recall that a partial order is reflexive, antisymmetric and transitive.

* This work was partially supported by Applied Dynamics International of Ann Arbor.

* Throughout this paper, we will use the operator δ to represent all data dependency relationships. Thus, $\mathbf{o}' \ \delta \ \mathbf{o}$ means that $\mathbf{o}'$ is dependent upon $\mathbf{o}$.

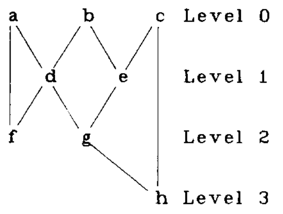

Fig. 1.

Consider the partial order of fig. 1. The nodes of this graph represent computational objects (either single operations or sets of operations) while the arcs represent dependency relationships between objects. Objects at the higher (numbered) levels are dependent upon objects at the lower levels (for example, **d** δ **a**). If we presume each object takes one cycle to execute, it is clear that the minimum execution time for the objects in fig. 1 is four time steps; this is the soonest object **h** could complete execution. The maximum number of useful processors would be three; this is the maximum number of objects at any level (on level 0).

The problem addressed in this paper is the determination of sets of objects which can be concurrently executed. Upon visual inspection of fig. 1, one can observe that the objects at each of the "levels" can be concurrently executed because there are no constraints between them. These sets can be found by making a "breadth-first" traversal of the graph. In addition to those sets, the following objects have no data interdependencies, and therefore can be concurrently executed:

$$\{ae\}$$
$$\{cd\} \quad \{cf\}$$
$$\{fe\} \quad \{fh\}$$

The topological sort [7] is a well-known algorithm used to embed a partial order in a linear order *. Historically, the topological sort has been used to reorder computations for sequential (SISD) machines [1]; any topological sort of a program block represents a legal execution order for the operations of the block. As shown below, it can also determine (in linear time) sets of program operations which can be performed in parallel.

* The resulting linear ordering is not necessarily unique. That is, there may be more than one possible linear embedding for a partial order.

2. The topological sort

The topological sort can be described as a strategy for traversing a partially ordered graph. If we view the arcs in the graph as constraints upon nodes, then the sort provides a method for releasing nodes which insures that no node will be released until all of its constraints are satisfied.

The implementation of this graph traversal is straightforward. For reasons of efficiency we require that all objects **o** in the graph maintain two values:

$$\text{dependency\_count}\ (\mathbf{o}) = \text{cardinality}\ (\{\mathbf{o}' \mid \mathbf{o}\ \delta\ \mathbf{o}'\})$$

and

$$\text{dependent\_nodes}\ (\mathbf{o}) = \{\mathbf{o}' \mid \mathbf{o}'\ \delta\ \mathbf{o}\}.$$

The first value is a count of the initial constraints for a node; this is the number of objects which must be released before **o** can be released. The second value is a list of objects which are dependent upon **o**.

For example, the equation set:
a : = constant;
b : = constant;
c : = a + b;
d : = a + constant;
would have the augmented data dependency graph of fig. 2. The nodes in this figure have a field containing the *dependency_count*, and a field representing the output of the node. The arcs in the figure represent the sets of *dependent_nodes*.

The topological sort works by selecting (for

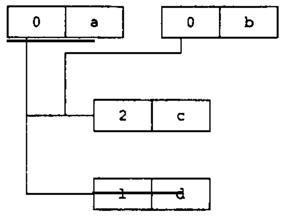

Fig. 2.

release) from among those nodes in the graph with no outstanding constraints. A suitable candidate can be found by examining those nodes whose *dependency_count* is zero. After a node has been released, all *dependent_nodes* have their *dependency_counts* decremented. Searching for a constraint free node could require a continuing traversal of the graph. We can avoïd these graph traversals by maintaining a set of mutually independent computations (which we'll call *MI*); this is the set of objects whose constraints have been satisfied. A pseudo-code description of the algorithm follows.

Input: an acyclic data dependency graph with counters

Output: a linear scheduling of computations in the graph

Initialization: $MI = \{\mathbf{o} \mid \text{counter } (\mathbf{o}) = 0\}$

```
Iteration: while ($MI \neq \emptyset$) do
  Select ($\mathbf{o} \in MI$)
  $MI := MI - \{\mathbf{o}\}$
  $(\forall\, \mathbf{o}')(\mathbf{o}'\ \delta\ \mathbf{o})$ do
    Counter ($\mathbf{o}'$): = counter ($\mathbf{o}'$) $- 1$
    If counter ($\mathbf{o}'$) $= 0$
      then $MI := MI \cup \{\mathbf{o}'\}$
  od
od
```

The **Select** procedure can incorporate a scheduling mechanism to determine which node to release. While this selection criteria could be complex (and time expensive), the cost of the traversal of the graph is quite small. If there are *m* objects in the graph, and *n* dependency relationships, then the cost of the graph traversal is

$$c_1 m + c_2 n$$

since each object is visited at most twice (once for initialization and once during iteration) while each dependency arc is traversed at most once.

One fairly simple, but nevertheless useful, selection mechanism constructs sets from the "levels" of the partial order. Rather than selecting a single node from *MI*, each iteration of the topological sort releases all nodes in *MI*. This amounts to a "breadth-first" traversal of the dependency graph. The number of sets constructed indicates the minimum possible execution time for the nodes of the graph, while the cardinality of the largest set is the maximum number of useful processors needed by the program.

There are three topics we need to examine when discussing the utility of the topological sort for parallelism detection. First, can the program section realistically be represented as a partial order? Does the graph of the program contain cycles? Second, what are the problems associated in building the graph of the data dependency relationships? Lastly, what kinds of scheduling are possible for the partitioned computations?

3. Definitions

Data dependencies between objects are determined by examining the input set and the output set of an object. The input set (denoted by **IN**[o]) is the set of operands used for computations, while the output set (denoted by **OUT**[o]) consists of the values generated during execution. The following definitions are useful for discussing these sets, and the dependency relationships between objects; for a discussion of the role of data flow analysis in program optimizations see ref. [1] or ref. [5].

A **basic block** (or block) is a linear sequence of program statements having one entry point and one exit point. Execution of any statement in a block implies execution of all statements in the block. A **flow graph** is the graphical representation of the control flow of a program. The nodes of the graph are basic blocks, and the arcs represent successor relationships.

A **definition** is the calculation performed by an object. An assignment statement stores a definition into a variable. A definition of a variable **reaches** a point in a program if there exists a path from the definition to that point (without a subsequent redefinition). A variable is **used** by a block if it appears as an operand in a statement in the block prior to a redefinition in the block.

The following definitions of data dependency relationships are from ref. [9]. Given two objects O_i and O_j (where O_i is positioned in the control flow prior to O_j) we define that $(O_j\ \delta\ O_i)$ if any of the following are true:

Flow dependence – $\mathbf{OUT}[O_i] \cap \mathbf{IN}[O_j] \neq \emptyset$.

Output dependence – $\mathbf{OUT}[O_i] \cap \mathbf{OUT}[O_j] \neq \emptyset$.

Anti-dependence – $\mathbf{IN}[O_i] \cap \mathbf{OUT}[O_j] \neq \emptyset$.

4. Intra-block analysis

This section presents the problems associated with manipulating the graph used to represent the basic block of a program. There are three points to examine. Can a block be represented as a partial order? What problems exist in building the program's graph? What kind of schedules can be built for the computations?

The computational objects in our discussion of blocks are statements *. The statements within a block have a linear order; there cannot be execution cycles within a block. This implies that there cannot be data dependency cycles in a block's data dependency graph, therefore this graph is a partial order.

The process of building the data dependency graph for a block has two steps. First, we construct a list of the outputs of each object in the block. These can be collected and sorted by name of the variable, and position of the statement in the block. If an array element is used for output, a conservative approximation would be to enter the array name into the list **. If an output is multiply defined in the list, then an output dependence exists between these objects. Renaming † all but the last occurrence of a scalar variable will break this type of dependence. Without extensive analysis, multiple outputs to array variables requires the establishment of an output dependence between the two objects.

The second step in graph construction is to insert the dependency arcs. This task uses the elements of **IN** †† (for each object in the block) to search the list of outputs. A **flow** dependence exists between definitions and their usage in the block. An **anti** dependence exists when a variable is used, then redefined within the block #.

The unrenamed elements of the output list and those variables which are **used** (before definition) constitute the elements of **OUT** and **IN** for the block. These sets will be used for interblock analysis.

There are two ways the computations in a block can be scheduled: synchronously or asynchronously.

The execution times for all objects in the block are known at compile time, so a deterministic schedule can be built. This avoids the cost of runtime synchronization mechanisms. By using an attribute grammar scheme for instruction selection [4], the compiler can collect data about the computation tree. This information would include the height of the expression, and resource needs of the tree. We can accurately determine the execution time of the tree (since we know the cost of each operation used by the tree), and thus can build a schedule for it. A microcode compaction strategy [3] could be used to match the needs of the expression trees with the available machine resources.

The asynchronous approach to scheduling oper-

* Another approach would be to take the operators of statements as our objects. Indeed, the operators and operands in a block are commonly represented by a **DAG** (a Directed Acyclic Graph) [1]. This representation could yield more parallel operations, since all operators in the block would be examined for parallel execution.

** Better analysis of array usage requires determining whether different indexing expressions could possibly reference the same array element. This requires examining the definition points for the variables used in indexing; these definitions may be generated within other blocks.

† Renaming is the assignment of a different storage location for a definition of a variable. Since only the last definition of a variable will reach the end of the block, all earlier definitions are live only within the block, and thus are temporary values.

†† All operands used to index an array element are part of **IN**, even if the array element itself is used only for output.

# For scalar variables, the only meaningful **anti** dependence is between the first usage of the variable in the block (if the definition comes from outside the block), and the definition which reaches the end of the block. This is true because all other definitions and usages could be renamed.

ations is equivalent to the dataflow machine model for computations [##]. This strategy has been reported in the literature [6], and is discussed in section 6.

In this section we have shown that a program block can be represented as a partial order. The process of graph construction is straightforward, although collecting accurate usage analysis for array and pointer references is difficult. The scheduling for parallel execution of intra-block objects can be can be done at compile time for a synchronous processor.

5. Inter-block analysis

This section discusses the relationship between basic blocks of programs. Again, there are three points to address. Can the control flow graph be augmented to reflect the data dependency relationships, and is this graph partially ordered? What problems exist in construction this graph? What kinds of schedules can be constructed for the parallel operations?

The data dependency problems encountered in inter-block analysis are "classic" data flow analysis problems [5]. We are more concerned with the data dependency relationships between blocks than with the program's control flow. For this reason, we will restrict our discussion to **structured** flow graphs (representing programs written without **GOTO**s). This insures that all nested control structures have exactly one entry and one exit point. Given this restriction, the process of representing the control flow graph as a partial order is straightforward.

In general, the control flow graph does contain data dependency cycles. These cycles can exist because of two kinds of control structures:

1) Iterative control structures. These occur wherever a **for** or **while** statement is used.
2) Recursive functions. This includes sets of mutually recursive functions.

Data dependency cycles must be eliminated before the topological sort can be used to detect parallelism.

Iterative cycles are "static"; their existence can be determined at compile time. Their effect on the graph can be minimized by encapsulating the cycling portion of the graph as a single node. In this way, the cycles are transformed to reflexive arcs in the graph. The resulting graph is a hierarchical representation of the program * which contains no cycles, only reflexive loops.

The data dependence cycles occurring in recursive functions cannot, in general, be processed entirely at compile time. This problem is discussed below.

Building the data dependency relationships between blocks of a program is similar to that of intra-block construction discussed above. Recall that **IN** and **OUT** for a block are calculated during intra-block analysis. The (possible) difficulty in building the data dependency graph comes from trying to construct **IN** and **OUT** for nested control structures. Since these control structures can contain other blocks, the process of determining the inputs and the outputs for the control structure requires collecting input/output information from the nested blocks.

Given the conditional control structure **B′**

```
if cond
   then $\mathbf{B}_1$
   else $\mathbf{B}_2$
fi
```

the input and output sets are determined by:

$$\mathbf{OUT}[\mathbf{B}'] = \cup\, \mathbf{OUT}[\mathbf{B}_i]$$
$$\mathbf{IN}[\mathbf{B}'] = \mathbf{IN}[\text{cond}] \cup \mathbf{IN}[\mathbf{B}_i].$$

Since either path of the conditional statement can be traversed, it is necessary to compose the input and output sets of **B′** from the input and output sets of the blocks along both program paths.

The nested program block **B′**

[##] That is, if we presume that our computational objects are the operators of the statements.

* Hierarchy graphs (or H-graphs) have been used to study the semantics of programming languages and machines. See ref. [10].

```
while cond do
  B_1
  B_2
od
```

requires the same equations for computing its input and output sets. Even if a **flow** dependence exists between $\mathbf{B}_1$ and $\mathbf{B}_2$, so that a definition computed in $\mathbf{B}_1$ is used in $\mathbf{B}_2$, $\mathbf{B}_1$ *may* not execute. In this event, the definition used in $\mathbf{B}_2$ could come from outside of the structure $\mathbf{B}'$. Therefore all definitions required by $\mathbf{B}_2$ must also be required by $\mathbf{B}'$.

The positioning of dependency arcs is done in the same fashion as in intra-block analysis. The relative position of the definition(s) and usage of a variable in the control flow is used to determine the type of dependence (**flow, anti** or **output**).

Graph construction for functions is difficult. The above discussions presume that there is a one to one relationship between a program's operations and its data dependency relationships. For a function, this is not always so. Consider a function which is multiply invoked *. Even though the function may not be recursive, and may be parameterless (so we avoid data dependency analysis), the data dependency graph of the function body *must* contain information about the invocation point (an invocation number) to permit possible parallel execution of the function body. For each invocation point, a separate dependency graph must exist.

The inclusion of parameters and different parameter passing disciplines [1] complicates analysis. Use of Call by *VALUE* and call by *RESULT* parameters ** permits the encapsulated analysis of the function body. Since these parameters are transferred only at the point of function invocation (or return), the actual variables used as parameters have no data dependency effect on the function body. While individual copies of a function's dependency graph need to be generated (one for each invocation point), there are no structural differences in the copies. Determining the data dependency arcs for the function at the invocation point is straightforward; no examination of the function body is necessary. *Value* parameters are elements of **IN**, while *result* parameters are elements of **OUT**.

The use of call by *NAME* or call by *REFERENCE* parameters can change the structure of the function's data dependency graph. These parameter passing disciplines **require** a full data dependency analysis for the function body. In addition, since either type of parameter can serve as input or output to the function, linking the invocation point to the surrounding graph requires the analysis of the use of the parameter within the function body.

Scheduling for interblock parallelism must generally be done at execution time †. Two types of scheduling mechanisms can be used: static and dynamic.

The synchronization mechanisms can be statically built into the program's code using either semaphores of COBEGIN/COEND structures [2]. This is effectively building the data dependency graph into object code of the program. Semaphores are used by Jajodia [6]. While they permit a faster release of data dependency constraints, their use would be incorrect for functions. The COBEGIN/COEND structure permits a execution time scheduler to provide the invocation values for functions, at the expense of a delay in releasing data dependency constraints.

The data dependency graph can also be passed to a runtime execution scheduler. An execution time scheduler could provide a "dataflow" schedule, where the computations are packets of operations, rather than single operations. Since a topological sort is not time intensive, its integration into a scheduler isn't expensive. The only difficulty is that the scheduler must allocate and release data dependency graphs for recursive functions.

There is a potential problem with recursive functions. Each invocation of the function could

* Further discussion of functions presumes that the function is invoked at different points of the program.

** These are the **in** and **out** parameter modifiers of *ADA*.

† If the loop is bounded at compile time, then a deterministic schedule could be built. Conditional statements within a bounded loop, however, would result in only an approximation for the execution time of the loop being available. If the loop is bounded at execution time, static analysis is inaccurate.

spawn concurrent tasks. The possibility exists that an exponential number of outstanding tasks could be generated. Consider the mutually recursive routines:

```
proc$P_a$    proc $P_y$
  ...          ...
   $B_x$;       $P_a$;
   $P_y$;       ...
   $B_z$;      end
  ...
end
```

If B_x, P_y and B_z are data independent, then each invocation of P_a could spawn three tasks. Their concurrent execution would cause a proliferation of tasks. A safe way to prevent this would be to "serialize" the parallelism at the point of recursive invocations. For the above example, this could be accomplished by insuring that P_y executed either before (or after) the pair $\{B_x, B_z\}$. In this fashion, only one task would be allowed to execute the call chain for recursive procedures.

We have shown that a structured program can be represented as a partial order, where looping constructs are encapsulated. The data dependency relationships can be found by synthesizing the input/output sets of the encapsulated code. Parallel operations must be synchronized at execution time. The synchronization mechanisms can be inserted in the generated code, or the data dependency graph can be passed to an execution time scheduler.

6. Related work

The topological sort has been reported as a method for parallelism detection in at least two recent papers [6,11]. While both papers describe the operation of the sort in differing degrees of detail, neither identifies the work as a topological sort.

The system described by Jajodia et al. [6], emphasizes analysis and scheduling of intra-block parallelism. Their computational "objects" are statements of the program. After determining dependency relationships, they place semaphores with **fork** and **join** operations in the code of the program to synchronize the parallel execution. They provide a proof that if two objects are bound by an **anti**-dependence * relationship, then variable renaming could break the dependence and allow for parallel execution. They use this theorem to show a scheme for the parallel execution of unrolled loops.

An important question, not addressed in the paper, is the tradeoff between the time savings gained from the parallel execution of the program statements versus the expense of executing the synchronization primitives. Since (at least one) semaphore signal is executed at the end of each statement, the runtime expense must be quite high. It is unnecessary to incur this expense, however, because the schedule for a block can be determined at compile time. However, it would be interesting to explore the utility of positioning the semaphore signals after definitions of elements of **OUT** for the whole block; this might speed the release of subsequent, dependent blocks.

Vairavan [11] introduces the idea of providing the data dependency graph to an execution time scheduler. They recognize that the cost of the graph traversal (by the topological sort) is relatively inexpensive, so the overhead for releasing data dependencies at execution time is low.

Neither paper discusses the problems of cycles in the dependency graphs. As discussed above, these can occur with LOOP constructs in the program, or with recursive functions. Jajodia [6] claims that functions which are "freed from side effects" can be scheduled in their fashion. However, because they position semaphores in the code body of their translated programs, the parallel execution of the function body would cause improper releasing of data dependencies. Recursive functions (and the proliferation of subtasks) are not addressed. Vairavan [11] also neglects these topics.

7. Conclusion

The topological sort has been shown to be an efficient method for the detection of parallel oper-

* The paper neglects the detection of output dependencies.

ations in a partially ordered representation of a sequential program. The algorithm can easily incorporate a scheduler for the concurrently executable operations. Intrablock operations (those within the basic block of a program) have an execution cost which can be determined at compile time. To avoid the overhead of runtime synchronization mechanisms, these operations can be scheduled at compile time for execution on a synchronized parallel processor. Basic blocks have execution times which aren't fixed at compilation time, so runtime synchronization between data dependent blocks is required. The synchronization mechanisms can be positioned in the program's code, or the data dependency graph of the program can be passed to a runtime execution scheduler.

Acknowledgements

J.R. Levenick and Uwe Pleban read earlier drafts of this paper and contributed greatly to its readability. Bertrand Meyer proposed a clearer presentation of the topological sort. John Stark contributed the use of his office for preparing this paper.

References

[1] A.V. Aho and J.D. Ullman, Principles of Compiler Design (Addison-Wesley, Reading, Massachusetts, 1977).
[2] G.R. Andrews and F.B. Schneider, ACM Computing Surv. 15 (1983) 3.
[3] S. Davidson, D. Landskov, B. Shriver and P. Mallett, IEEE Trans. on Computers C-30 (1981) 460.
[4] M. Ganapathi, PhD Thesis, University of Wisconsin, Madison (1980).
[5] M. Hecht, Flow Analysis of Computer Programs (North-Holland, New York, 1977).
[6] S. Jajodia, J. Liu and P. Ng, IEEE Trans. on Software Eng. SE-9 (1983) 436.
[7] D. Knuth, Fundamental Algorithms (Addison-Wesley, Reading, Massachusetts, 1968).
[8] D.J. Kuck, ACM Computing Surv. 9 (1977) 29.
[9] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure and M. Wolfe, Dependence Graphs and Compiler Optimizations, Proc. of ACM Symposium on Principles of Programming Languages (Jan. 1981) p. 207.
[10] T.W. Pratt, H-Graph Semantics, DAMACS Reports 81-15 and 81-16, Department of Applied Mathematics and Computer Science, University of Virginia, Charlottesville (September 1981).
[11] K. Vairavan and V. Vairavan, A Model for the Parallel Representation and Execution of Programs. Proc. of the IEEE Fourth Intern. Computer Software and Applications Conference (October 1980) p. 295.