DYNAMIC DETECTION OF PARALLELISM IN PASCAL-LIKE PROGRAM

Zheng Yong , Qian Jiahua

Dept. of Computer Science, Fudan University
Shanghai, P.R.China

ABSTRACT

The execution time of Pascal-like programs can be decreased by putting solutions to problems in their maximally parallel forms. In this study Pascal-like programs are considered, and analysis of parallelism is performed by using a dynamic recent data dependency graph (DDG). The means taken in this paper is able to derive maximally parallel versions of the programs and to minimize the execution time. The advantage of this means is not only to save the effort in implementation but also to possess generality and high-efficiency resulting from dynamically locating parallelism in program. The study shows that the maximally parallel program can run in considerably less time than that needed to run the original sequential Pascal-like program.

Keywords:
Statement parallelism dynamic detection, parallel execution, data dependency, Pascal-like program, multiprocessor

1. Introduction.

With the significant decrease in hardware cost, it is becoming economically feasible to design and build large multiprocessor system, and the interest in parallel processing is rapidly increasing. Some new problems arise in programming this kind of computer system:

1.The onus is on the programmer to detect and express all possible parallelism in his program.

2.Small changes in the program may mean that the programmer has to reorganize all the parallelism he has written in to the program.

3.Programs already in existence will have to be rewritten.

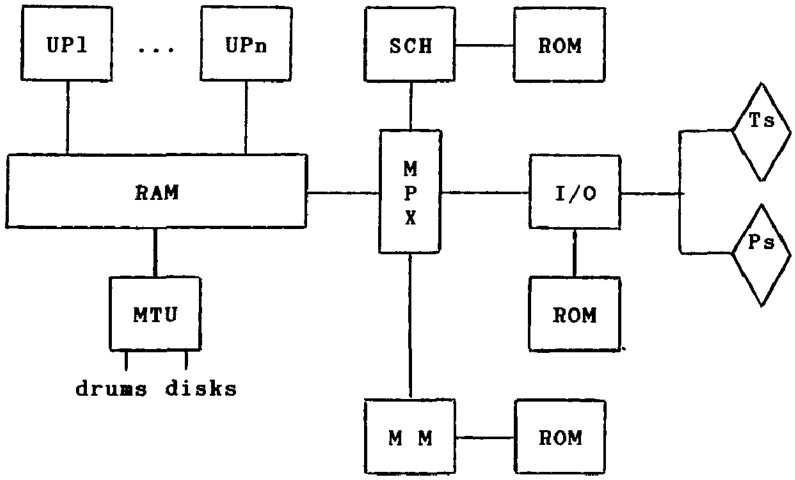
It would be beneficial to be able to examine automatically programs, indicate the relationship between parts of the code, and execute the program in parallel. A lot of efforts have been made in this aspect(e.g.[2-10]). Most of them handle with Fortran or Fortran-like programs, and statically consider the parallelism of program. In this paper, we shall develop a means of dynamically locating parallelism in runtime and executing in parallel in a multiprocessor system for Pascal-like language.

2. Machine Considerations.

In this section, A prototype of the architecture of multiprocessor system will be simply described. It, shown in Fig.1 consists of three special purpose dedicated processors for carrying out the system management task, and N processors for executing user processes.

All of the processors operate independently and communicate with each other via random access memory (RAM). Most of the data and programs used by the dedicated system processors are stored in dedicated read-only memories (ROMs) shown in Fig.1. The system processors access the RAM only to communicate with other processors or share some system information, and for this reason they can share a single port without unduly high memory contention problem.

The problem of size of the RAM is resolved by making use of swapping. Most of information resides on drums, and the memory transfer unit is kept continually busy by the memory manager, carrying out this swapping operation.



UP1,...,UPn : user processors
SCH : scheduler
I/O : I/O processor
MM : memory manager
MPX : multiplexor
MTU : memory transfer unit
Ts : terminals
Ps : peripherals

Fig.1.

3. Data Dependency Graph (DDG).

To discuss data dependency and control dependency, we define some relations between statements.

In sequential program, for statements Si and Sj, if Si is executed before Sj, we denote by Si < Sj.

Definition 1. For statements Si and Sj, Si and Sj can be executed in parallel, denoted by Si||Sj, if

$$(*)\quad In(Si) \cap Out(Sj) = \emptyset \ \& \\ Out(Si) \cap In(Sj) = \emptyset \ \& \\ Out(Si) \cap Out(Sj) = \emptyset$$

where In(Si) is all variables which are input of Si, and Out(Si) is all variables are changed by Si.

Definition 2. For statements Si and Sj, we say Si must be executed before Sj, denoted by Si <* Sj, if Si and Sj are not satisfy (*), and Si < Sj.

To parallelize linear program, we support a dynamic recent data dependency graph (DDG) of a program in runtime. DDG is a graph G = (V,E). (Ni,Nj) $\in$ E means Si <* Sj, where Si and Sj are statements corresponding to nodes Ni and Nj, respectively. Node set V corresponds to the statements which are concerned recently.

Here we need to interpret the means of recently concerned statements. In DDG, we do not cope with all statements equally. The complicated structure statement to which we do not pay many attentions is mapped to a node, and the structure statement to which we are paying a lot of attentions is mapped to many nodes, each of which corresponds to a sub-statements of this statement. The more complicated the statement to which a node corresponds is, the higher the abstract level of this node is, so the less attention is paid to this statement.

Node set V consists of several kinds of nodes. Their origin will be described below.

1. Assignment statement, or Read/Write statement is mapped to a <u>assignment node</u>.

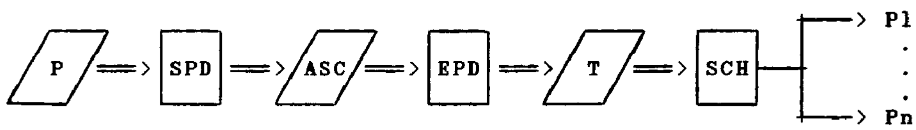
2. Procedure call statement is mapped to a <u>call node</u>.

3. IF statement 'if sb then S1 else S2' is rewritten to
'sc := sb; if sc then S1 else S2', where sc is a special auxiliary variable. Thus 'sc := sb' is mapped to a <u>selection condition node</u>, and 'if sc then S1 else S2' is mapped to a <u>selection node</u>.

4. While statement 'while lb do S1' is rewritten to 'lc := lb; while lc do S1', where lc is a special auxiliary variable. Thus 'lc := lb' is mapped to a <u>loop condition node</u>, and 'while lc do S1' is mapped to a <u>loop node</u>.

5. Compound statement is mapped to a <u>compound node</u>.

The assignment node, selection condition node, and loop condition node are atomic nodes, which can not be expanded. Other nodes are structure nodes, which can be expanded. Each node N is



P : program
ASC : assignment statement chains
T : tasks
P1,...,Pn : processors
SPD : statement parallelism detector
EPD : expression parallelism detector
SCH : scheduler

Fig.2.

associated with two fields In(N) and Out(N), where In(N) = In(S), and Out(N) = Out(S), S is the statement corresponding to N.

Let the program body is a compound statement S1;...;Sk, initially, DDG contains node N1,...,Nk, each of which corresponds to a Si (1 <= i <= k). With the running of program, the nodes of DDG are deleted or expanded, and the edges of DDG are modified correspondingly.

4. Parallel Execution of Program.

4.1. The Overview of System.

In a program there are two sources of parallelism which can be detected. The first consists of arithmetic or logic expressions, and this case has been studied by many authors (e.g.[4,10]). The second sources of parallelism consists of statements, especially loop statement.

By detecting the two sources of parallelism in a program, we transform the program into a set of tasks that can be executed simultanously. It is shown in Fig.2.

We give the definition of ASC below.

Definition 3. In recent DDG, G = (V,E), a sequence of atomic nodes, say N1,...,Nk, (k > 0) is called <u>chain</u>, if

1. $(N_i, N_{i+1}) \in E$ (1 <= i < k)

2. indegree(N1) = 0 & outdegree(N1) = 1
   & indegree(Nk) = 1
   & indegree(Ni) = outdegree(Ni) = 1
   (1 < i < k)

Definition 4. Statement sequence S1,...Sk, is called <u>assignment statement chain</u> (ASC), if N1,...,Nk is a chain in recent DDG, where Ni (1 <= i <= k) is the node corresponding to Si in recent DDG.

In the system, ASCs can be transformed by 'statement substitution' and other techniques to obtain a set of tasks that evaluate expressions simultanously. The transformation is accomplished by EPD.

In this paper we will mainly discuss the techniques of statement parallelism detection.

4.2. Statement Parallelism Detection.

IF and WHILE statements are main source of parallelism, especially WHILE statements. In DDG, IF (WHILE) statement is mapped to two nodes, one corresponds to evaluation of selection (loop) condition, another corresponds to modified IF (WHILE) statement. This make it possible to breakdown IF and WHILE statements to their sub-statements. They are shown in Fig.3 and Fig.4.

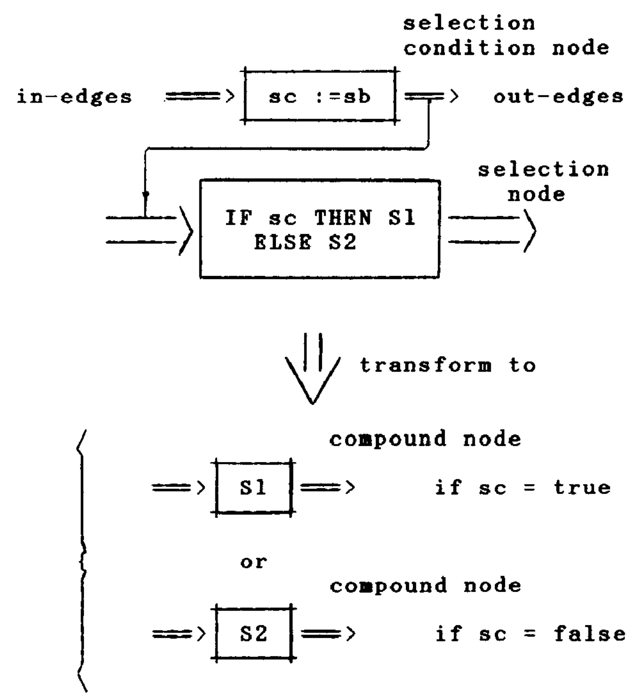


Fig.3.

Usually, loop condition depends only a few variables in loop body S. So, after these variables are evaluated, the loop condition can be evaluated again. Thus, we can constantly breakdown loop, and some

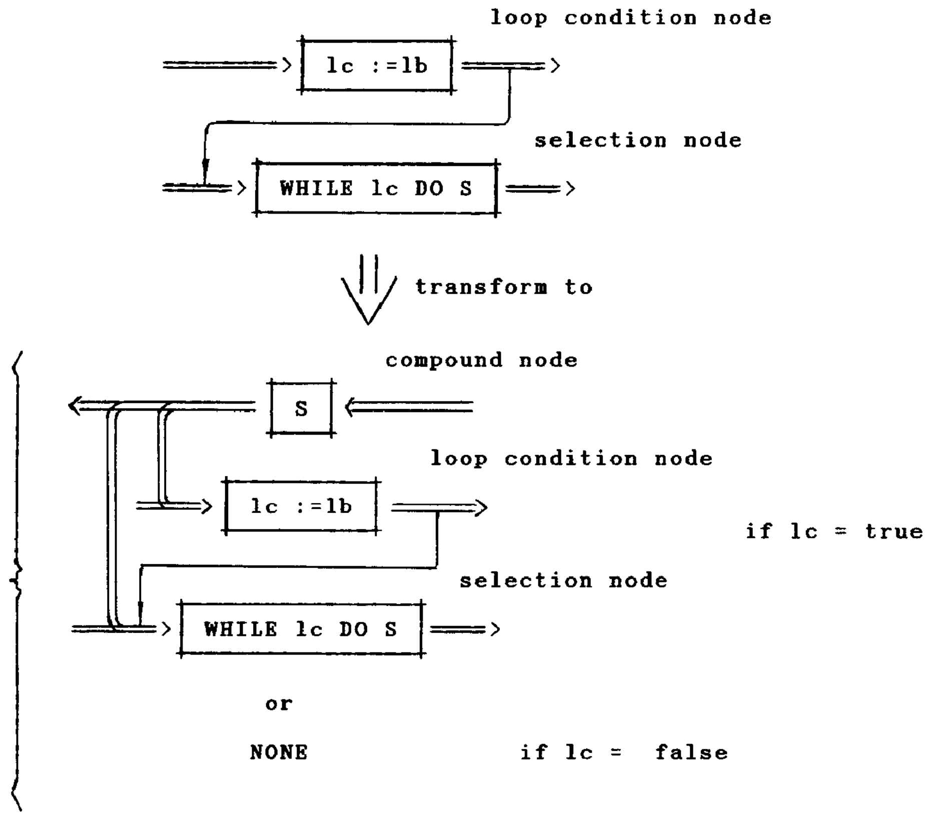


Fig.4.

parts of former iterations of loop and some parts of later iterations can be execution in parallel.

In the system, SPD will constantly find ASCs according to recent DDG, and output them to EPD. After a ASC is executed, SCH will send a message to SPD to acknowledge it. If the end of ASC is a statement to evaluate auxiliary condition variable, its boolean value will also be sent to SPD. SPD will modify dynamically recent DDG according to the messages sent by SCH.

SPD consists of two processes Find_Chain and DDG_Modify. Except first times, Find_Chain is wakeuped by the finish of DDG_Modify. DDG_Modify is wakeuped by the message sent by SCH.

We obtain ASCs by finding corresponding node chains in recent DDG. Formal description is shown in Fig.5.

In Fig.5, procedure EXPAND(N) replaces call node or compound node N with a sub-graph corresponding to the sub-statements of procedure body or compound statement, i.e. let procedure body or compound statement be 'S1;...;Sk', we replace N with N1,...,Nk, where Ni (1 <= i <= k) is the node corresponding to Si. After replacement of nodes, the edges of DDG is modified correspondingly. It is shown in Fig.6.

```
Process Find_Chain
  VAR  ANS, SNS : subset of nodes in
                       recent DDG.
        N : node in recent DDG.
        C : chain in recent DDG.
  BEGIN
    SNS := { N | is_structure_node(N)
                 & indegree(N) = 0 }
    INS := { N | is_atomic_node(N)
                 & indegree(N) = 0 }
    FOR  each N <- SNS  DO
      IF  is_call_node(N) or
            is_compound_node(N)
      THEN  EXPAND( N )
      ELSE  ERROR
      ENDIF
    ENDFOR
    FOR  each N <- INS  DO
      C := NIL | N
              /* add N to empty chain */
      N := Son(N)
      WHILE  (outdegree(N) = 1) &
        ( indegree(N) = 1) & ( OK(C) ) &
        ( is_atomic_node(N) )  DO
        C := C | N
            /* add N to the end of chain */
        N := Son(N)
      ENDWHILE
      CONDENSE(C)
      OUTPUT( C )
    ENDFOR
  END.
```

Fig.5.

```
Proc  EXPAND( Var N : node in recent DDG )
  VAR NS1,NS2 : nodes subset in recent DDG
       N1,N2 : node in recent DDG
       SG : the data dependency sub-graph
             of node N  /* SG = (SV,SE) */
  BEGIN
    IF  is_structure_node(N)
    THEN
      NS1 := { N1 | (N,N1) ∈ E }
      NS2 := { N2 | (N2,N) ∈ E }
      Delete the edges associated with N
      Replace(N,SG)
      IF  is_call_node(N)
      THEN
        FOR  each N1 ∈ SV  DO
          In(N1)  := In(N1)(F/A)
          Out(N1) := Out(N1)(F/A)
            /* F is formal parameter set
            and A is actual argument set.
            In(N1)(F/A) is a variable set
            obtained by replacing F with A
            in In(N1) */
        ENDFOR
      ENDIF

      FOR  each N1 ∈ SV  DO
        FOR  each N2 ∈ NS1 U NS2  DO
          IF (In(N1)∩Out(N2) ≠ Ø) or
             (Out(N1)∩In(N2) ≠ Ø) or
             (Out(N1)∩Out(N2) ≠ Ø)
          THEN
            IF N2 ∈ NS1
            THEN  E := E U {(N1,N2)}
            ELSE  E := E U {(N2,N1)}
            ENDIF
          ENDIF
        ENDFOR
      ENDFOR
    ENDIF
  END.
```

Fig.6.

In Fig.5, procedure CONDENSE(C) condense chain C into the end node of C in DDG. Thus, the end node will correspond to a assignment statement chain.

OK(C) is a boolean function. If chain C can add new element in the end, it is true, otherwise it is false. It is affected by many factors, such as the *number of processors*, *the number of* variables occuring in C, the occurence times of each variable in C, etc. e.g. in extreme situation

$$OK(C) = \begin{cases} true & C = NIL \\ false & otherwise \end{cases}$$

Thus, the chain can only contain one element.

OK(C) must be selected carefully for each system. We will not discuss it here.

Processes DDG_Modify receives the messages from SCH and modifies recent DDG. Formal description is shown in **Fig.7**.

```
Process DDG_Modify
  VAR  CS : chain set in recent DDG
       C  : chain in recent DDG
       N,N1,N2 : node in recent DDG
  BEGIN
    Receive CS from SCH
       /* CS is the chain set
          which have be executed */
    FOR  each C ∈ CS DO
      N := end( C )
          /* N is the end node of C */
      Delete C and all edges associated
        with N from recent DDG

      CASE the type of N DO
        Selection condition node :
          BEGIN
            N1 := the selection node
                    relating to N
            IF  Value(N)
            THEN  N2 := GETNODE( TB(N1) )
            ELSE  N2 := GETNODE( FB(N1) )
            ENDIF
              /* TB(N1), FB(N1) are true-
               branch and false-branch of
               IF statement corresponding
               to N1, respectively */
            Replace(N1,N2)
            EXPAND(N2)
          END

        loop condition node :
          BEGIN
            N1 := the loop node relating
                   to N
            IF Value(N)
            THEN  N2 := GETNODE( B(N1) )
                   /* B(N1) is the loop
                    body of loop statement
                    corresponding to N1 */
                  INSERT(N,N1)
                  INSERT(N2,N)
                  EXPAND(N2)
            ELSE  Delete N1 and all edges
                        associated with N1
            ENDIF
          END
      ENDCASE
    ENDFOR
    Wakeup( Find_chain )
  END.
```

Fig.7.

In **Fig.7**, function Value(N) is the result after executing selection or loop condition node N, which is obtained from received message.

GETNODE(S) is also a function, it returns a new node to which statement S is mapped.

Procedure *INSERT*(N1,N2), *in DDG*, inserts node N1 to node set and edges associated with N1 to edge set under the condition S1 < S2, where S1 and S2 are statements corresponding to N1 and N2, respectively.

5. Conclusion.

We has presented the techniques to dynamically detect the parallelism of Pascal-like program in runtime. Although the compilation for parallel machine is still a obstacle, the approach introduced in this work may put programs in their maximally parallel forms to minimize the execution time. Its advantage is not only to save effort in implementation, but also to possess generality and high-efficiency resulting from dynamically locating parallelism in program.

6. Reference.

[1]. Chattergy,R., & Pooch,U.N.
A distributed function computer with dedicated processors. Computer J. Vol.22, No.1 (1979) pp37-40

[2]. Evans,D.J., & Williams,S.A.
Analysis and detection of parallel processable code. Computer J. Vol.23, No.1 (1980) pp66-72

[3]. Foulk,P.W.,& Nassar,S.M.
Analysis of parallelism in nested DO loops. J. Syst. & Softw. Vol.5, No.1 (1985) pp73-80

[4]. Hellerman,H.
Parallel processing of algebraic expressions.
IEEE TC. Vol.15 (1966) pp82-91

[5]. Kuck,D.J.
Parallel processing of ordinary programs. Advances in Computers Vol.15 (1976) pp119-179

[6]. M.Di Manzo, A.L.Frisiani,& G.Olimpo.
Loop optimization for parallel processing. Computer J. Vol.22, No.3 (1979) pp234-239

[7]. Nagl,M.
Application of graph rewriting to optimization and parallelization of programs. Computing Suppl. 3. (1981) pp105-124

[8]. Ramamoorthy,C.V.,& Gonzalez,M.J.
A survey of techniques for recognizing parallel processable streams in computer programs.
Proc. AFIP. 1969 FJCC.

[9]. Stone,H.S.
One-pass compilation of arithmetic expressions for a parallel processor.
CACM Vol.10 (1967) pp220-223

[10]. Towel,R.A.
Control and data dependence for program transformations.
Report No. UIUCDCS-R-76-788. (1976). University of Illinois at Urbana-Champaign.