. Indian Inst. Sci., Mar.-Apr. 1991, 71, 125–157
© Indian Institute of Science.

A parallelizing compiler for Pascal

MAULIK A. DAVE AND Y. N. SRIKANT
Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India.

Received on September 24, 1990; Revised on March 18, 1991

Abstract

A parallelizing compiler takes as its input a program in a sequential language such as FORTRAN or PASCAL and after extracting parallelism which is implicit in it, generates code which is suitable for execution on a parallel processor. We have used Pascal for our compiler as it has extra features such as recursion, pointers, record structures and nesting of procedures. The aim is to extract the maximum parallelism in a reasonable time. Instead of flowgraphs, the concept of boxgraphs has been developed and implemented. An algorithm has been implemented to carry out simple and interprocedural dataflow analysis, array subscript analysis, to detect parallelism in boxgraphs and to convert them into boxgraphs depicting parallelism explicitly. The compiler generates code for the ORG Supermax machine, a shared memory multiprocessor with two 68020 processors running on UNIX operating system V.3.

Key words: Compiler, parallel processing, parallelization.

1. Introduction

1.1. *Multiprocessors*

The demand for high-speed computers is increasing rapidly in structural engineering, weather forecasting, petroleum exploration, fusion energy research, and other areas which are extremely important for the advancement of human civilization. Modern computer architectures are centered on the concept of parallel processing. Parallel computer systems can be classified into five groups: pipelined computers, array processors, multiprocessor systems, dataflow computers, and VLSI algorithmic processors. We shall be concerned in this paper solely with multiprocessor systems and their programming aspects and hence shall not discuss further pipelined and array processors, dataflow computers, and VLSI processors.

1.1.1. *Overview of multiprocessor architecture*

A basic multiprocessor organization is conceptually depicted in Fig. 1. The system contains two or more processors of approximately comparable capabilities. All processors share

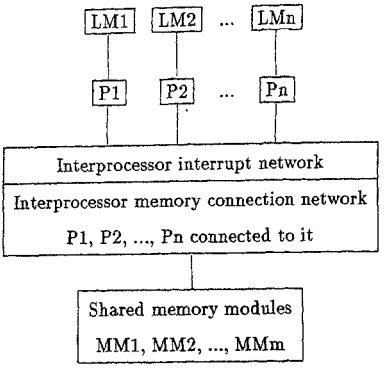

FIG. 1. Basic multiprocessor organization. LM = Local memory, P = Processor, MM = Memory module.

access to common sets of memory modules, I/O channels, and peripheral devices. The entire system is controlled by a single integrated operating system providing interactions between processors and their programs at various levels. Apart from the shared memory and I/O devices, each processor has its own local memory and private devices. Interprocessor communication can be performed either through shared memory (shared memory multiprocessor) or through a message-passing mechanism (message-based multiprocessor). Multiprocessor systems are also known as multiple instruction stream-multiple data stream (MIMD) machines.

The commercial multiprocessor systems available are Sequent Balance 21000, Intel iPSC Concurrent Computer d5, Apollo DN 10000, 4 Processor VAX stations, Multiple transputer systems (from several vendors), Alliant FX/8, BBN Butterfly Parallel Processor, CRAY X-MP Model 22, FPS T Series Parallel Processor, IBM 3090, Loral Dataflo LDF 100, the Tandem multiprocessor, etc.

1.1.2. *Programming multiprocessors*

Programming a multiprocessor is different from programming a uniprocessor in two ways—architectural attributes and a new programming style peculiar to parallel applications.

An architectural attribute that may affect programming in a multiprocessor system is nonhomogeneity. If the central processors are nonhomogeneous, that is, functionally different, they must be treated differently by software.

The basic unit of a program in execution on a multiprocessor is a process, an independent

schedulable entity (a sequential program) that runs on a processor and uses system resources. It may also execute concurrently with other processes, delayed only when it needs to wait to interact with other processes or for resources. Hence, a parallel program can be said to consist of two or more processes.

1.2. *Parallelizing compilers*

There is another approach for exploiting parallelism on parallel machines other than writing parallel programs. This is to write such compilers, which will take sequential programs and determine the parallelism implicit in them. Such compilers which convert sequential programs into parallel programs are known as parallelizing compilers. Parallelizing compilers for multiprocessors are sometimes also called as concurrentizing compilers. Concurrentizing compilers try to create a parallel program consisting of parallely executable processes out of a sequential program. We shall be concerned in this paper with concurrentizing compilers only but we shall use both the terms interchangeably.

1.2.1. *Why or why not parallelizing compilers?*

The use of parallelizing compilers offers the following advantages over that of writing parallel programs.

(1) *Use of existing software packages:* Most of the software in existence is in sequential languages. Moreover, a lot of money has been spent on producing the existing software packages. We believe that rewriting all of them in parallel languages is much more expensive than restructuring them by parallelizing compilers.

(2) *Program portability*: For different kinds of architectures, we need different language features to be able to exploit the available parallelism explicitly[1]. To avoid writing a separate parallel program for each kind of parallel machine, one can write a single sequential program and use a parallelizing compiler to restructure them. This is more advantageous because a major part of a parallelizing compiler is machine-independent (only the code generator is different just like in an ordinary compiler), and hence can be used for various kinds of machines with only a little one-time extra effort. Thus parallelizing compilers improve program portability.

(3) *Training facilities not needed:* For parallel programming, programmers need additional training. Moreover, it is difficult to train non-computer science programmers in parallel algorithms and languages. Parallelizing compilers avoid all such extra strain on the programming community.

Unfortunately, parallelizing compilers have the following disadvantages also.

(i) The compilation time to detect fine-grain parallelism in sequential programs is very large.
(ii) Full parallelism is not detected by parallelizing compilers, specially in the presence of arrays.
(iii) Efficient parallel algorithms have been found for some problems but present-day parallelizing compilers are not able to convert the corresponding sequential programs into efficient parallel ones.

In spite of these disadvantages, the advantages, specially (1) and (3) above, seem to be overwhelming and hence more and more parallelizing compilers are being built all over the world.

1.2.2. *An introduction to parallelization*

When a sequential program is executed, one operation is performed at a time on the processor. The output of such a program is the output of the statements executed according to their textual sequencing. However, for a program, the same output can sometimes be achieved by different ordering of the statements also. The total ordering imposed by a sequential language is more restrictive than is necessary to guarantee a program's output. Only portions of the original total (sequential) ordering are absolutely essential to maintain the results. The required ordering is a partial ordering as opposed to the total ordering of the sequential execution. This phenomenon is exploited during the determination of parallelism in sequential programs.

Dependence is a relation among the statements of a program. A statement s_2 is dependent on statement s_1, if s_1 must be executed before s_2 to preserve the semantics of the original program. Under this definition, dependence represents the essential orderings within a program. Any execution order that preserves a program's dependences also preserves its output.

Now we explain two main types of dependences, *viz.*, data and control dependence. Consider the following two statements:

s_1: $a := b + c$;

s_2: $d := a + e$;

Since s_2 uses the value of a, which is changed by s_1, these two cannot be executed in parallel. s_2 is dependent on s_1 due to data consideration. Now, consider the following two statements:

s_3: if $(a <> 0)$ then

s_4: $b := c + d$;

s_4 and s_3 cannot be executed in parallel because execution of s_3 controls whether s_4 has to be executed or not. This is control dependence.

To determine data dependence in the presence of array references and pointers is very difficult and sometimes impossible. For example, consider the following statements:

s_5: $a[i + 1] := b + c$;

s_6: $d := a[j + 1]$;

To determine the dependence of s_5 and s_6 completely at compile time is impossible because we do not know which element of a will be changed in s_5 and which will be used in s_6. The subscript analysis methods yield information regarding *possible* dependence

only and not complete dependence as in statements s_1 and s_2 above. More details of dependence analysis and other details of a parallelizing compiler are explained in the forthcoming sections.

1.3. *Related work*

The area of parallelizing compilers is relatively new when compared with other areas of compilers. The foundation for work in this area was laid by Kuck and his group. Padua *et al* present some basic techniques of parallelization[2]. Banerjee has given an excellent array subscript test[3]. Wolfe has analyzed extensively the problems involved in vectorizing FORTRAN programs, and has given techniques such as loop fission, loop fusion, loop scalarization, and loop interchanging[4]. Apart from these, also described are symbolic dependence testing and sectioning[5] and PFC (Parallel Fortran Converter—a vectorizer for FORTRAN) developed at Rice University[6].

Loop concurrentization, trade off between vectorization and concurrentization, and other issues are discussed by Padua and Wolfe[7]. Wolfe throws some light on the difference between vectorization and concurrentization[8]. Wolfe and Banerjee review advanced techniques for data dependence testing and their applications in concurrentizing and vectorizing compilers[9]. PTRAN (Parallel TRANslator) is a system for automatically restructuring sequential FORTRAN programs for execution on parallel architectures. It is being developed at the IBM's T. J. Watson Research Center. Allen *et al*[10] give some techniques of interprocedural dataflow analysis used in PTRAN, and also an overview of it.

Application of interprocedural dataflow analysis to vectorization and concurrentization results in detection of more parallelism. Though Triolet *et al* claim good results from their method[11], unfortunately it involves complex algorithms which use symbolic computations and linear programming whose cost in terms of CPU time and memory space is heavy. Burke and Cytron have given a hierarchical dependence test[12] which is an improvement over previous methods. Callahan and Kennedy from Rice University[13] have described a new technique called regular section analysis. The idea is to preserve common array substructures as regular sections and use them for interprocedural dataflow analysis. Li and Yew[14] have introduced a data structure called atom images to propagate subscript information through call statements.

While implementing concurrentizing compilers on real multiprocessors, synchronization delay between concurrent processes and assignment of processes to processors poses a very difficult challenge[15-17]. Techniques have been developed for run-time detection of parallelism also[15,18-20].

A novel technique of parallelization is presented by Ferrante *et al*[21]. The so-called program dependence graph (PDG) proposed as an intermediate program representation makes explicit both the data and control dependences for each statement in a program.

Another area of research is pointer dataflow analysis. Modern-day programs written in languages like Pascal, and C use pointers extensively. Attempts are also being made to parallelize programs containing pointers[22-29].

1.4. *Highlights and organization of the paper*

We have implemented a parallelizing compiler which is based on certain new concepts and also some older but well-proven ones.

Most of the parallelizing compilers have been written for FORTRAN. We have chosen Pascal because it has extra features such as recursion, pointers, record structures, and nesting of procedures.

The aim of our compiler is to extract as much parallelism as possible in a reasonable amount of time. We observe that shared memory multiprocessors do not have a large number of processors. We have not detected fine-grain parallelism as it is not useful on shared memory multiprocessors (because of small number of processors and scheduling overheads) and the compilation time may become very large. For the latter reason, symbolic computation is also not used.

Instead of flowgraphs, we have developed and implemented the concept of boxgraphs in our compiler. Boxgraphs have the following advantages:

- As they do not contain cycles, analysis becomes easier.
- As the boxes are sufficiently large, scheduling overheads are not considerable.
- As each boxgraph can be analysed separately, parallel algorithms can be developed easily (future work).
- They can be constructed easily for block-structured languages.
- There is provision to display parallelism explicitly.

An algorithm has been implemented to carry out simple and interprocedural dataflow analysis and array subscript analysis, and hence to detect parallelism in boxgraphs and to convert them into boxgraphs depicting parallelism explicitly.

We have implemented our compiler on the ORG Supermax machine, a shared memory multiprocessor with two 68020 processors running on UNIX operating system V.3. Dynamic scheduling of processes and parallelism have been achieved using the features available in the system. We do not claim our implementation to be efficient but feel that it is a good test bed to try out experiments with parallelizing compilers.

Section 2 contains the details of intermediate representation. Section 3 describes dependence analysis. Section 4 contains the details of detection of parallelism, and Section 5 a description of our implementation of parallelism. Finally, Section 6 concludes with some results of parallelization and future directions.

2. Intermediate representation

In this section, the main features of the intermediate representation used in our compiler are described. Only a subset of Pascal, stripped off some of its features such as goto, case and with statements, file and set data type, variant records, has been chosen. However,

we believe that our principles, when applied to these features also, will work correctly. We have removed them to simplify the implementation.

2.1. *A brief overview of a parallelizing compiler*

The task of a parallelizing compiler can be divided into two parts: (1) Transforming the input program into intermediate representation (with parallelism explicit), and (2) Transforming intermediate representation to machine code.

The first part depends on the type of architecture of the machine for which the compiler will generate the code. However, it does not depend on specific machine characteristics. For example, the first part of a parallelizing compiler will be the same for all kinds of multiprocessors with shared memory, but will differ for multiprocessors and vector processors. The second part will need machine details such as the number and types of processors, and services provided by the operating system. It is because of this reason that portability of the first part is higher than that of the second part and can be dealt with separately.

The first part can be further divided into three parts:

(1) Generation of intermediate code with only sequential features. This is the same as in any usual compiler.
(2) Dependence analysis of the above intermediate code using the techniques of dataflow and control flow analysis. The aim of such analysis is to detect parallelism in programs.
(3) Conversion of the intermediate code in (1) above into one having parallel constructs also. Here, we use the dependence information computed in (2) above.

The intermediate representation which we have used contains a set each of quadruples and boxgraphs. This representation was chosen because it can be implemented on multiprocessors with any number of processors. Each procedure or function in the source program is recognized as a boxgraph and a set of quadruples. The set of quadruples will be the same as in any ordinary compiler and it will be generated during parsing. We shall now give a few basic definitions which help in explaining our intermediate representation.

2.2. *Some basic definitions*

Boxes are defined in such a way that when the input program is divided into a number of boxes, control flow analysis becomes simple.

(1) *Modified basic block*: A modified basic block is a basic block without any 'goto' quadruple at its end.

(2) *Box*: A box is a set each of whose elements can either be a box or a modified basic block and it can be one of the following:

  (i) *SIMPLE box*: It is a modified basic block.
  (ii) *COMPOUND box*: It is a set of boxes which can be executed in parallel.
  (iii) *FOR box*: It contains the box corresponding to the body of a for-loop, the loop index variable and the quadruples corresponding to the expressions of a for-loop (Fig. 2).

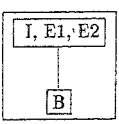

FIG. 2. For box for the statement for I: = E1 to E2 do B;

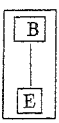

FIG. 3. Loop box.

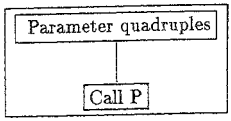

FIG. 4. Call box.

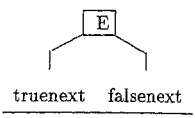

FIG. 5. Condition box.

(iv) *LOOP box*: It contains the box corresponding to the body of a repeat-until-loop and a modified basic block containing quadruples for the condition expression (Fig. 3). We convert while-loop into repeat-until-loop for convenience.

(v) *CALL box*: It contains quadruples for all the parameters and a call to a procedure (Fig. 4).

(vi) *CONDITION box*: It consists of expression quadruples formed due to if-then-else or if-then statement with special properties that it is connected to *two* boxes by the relation next (defined later). (Fig. 5).

(vii) *NULL box*: It does not contain anything.

(viii) *PARFOR box*: It is the same as a FOR box except that while executing a PARFOR box each iteration of a for-loop can be executed in parallel.

It is easy to see that boxes may contain other boxes also.

(3) *Next*: Let B be a box and let a and b be any elements of B. Then the relation next (denoted by N) can be defined as

$$N: B \rightarrow B,$$

$N = \{(a, B) |$ box b can immediately follow box a during execution of the program.$\}$ and having the following properties:

- If aNb then $a <> b$.
- If aNb then bNa is false.
- If aNb and bNc then cNa is false.
- For any a in B, there exist at most two distinct b_1, b_2 in B such that aNb_1 and aNb_2.
- There exists one and only one element x in B such that if yNx then y is not in B. Such an element is defined as the *starter* of B.
- There exists one and only one element x in B such that if xNy then y is not in B. Such an element is defined as the *ender* of B.

(4) *Exit*: Exit of a box b in B is a in B iff bNa.

(5) *Truenext and Falsenext*: Of the two exits of a CONDITION box b, the box which is to be executed if the condition of b is true is called the truenext of b, and the other, the falsenext of b.

(6) *Endif*: Every CONDITION box has a corresponding endif box. A box b is an endif box of c, if

- there are at least two paths in the boxgraph from c to b, and
- all the paths from c to b do not have any box in common except c and b.

(7) *Boxgraph*: We define a boxgraph as a directed graph with nodes representing boxes and with an arc from box b_1 to box b_2 iff $b_1 N b_2$. The properties of the relation *next* reflect the nature of any boxgraph corresponding to a Pascal subroutine. For the program segment given below, the corresponding boxgraph is shown in Fig. 6.

We divide a program into boxgraphs during parsing and quadruple generation itself. Since this uses standard syntax-directed translation methods possible with YACC on UNIX systems[30,31], we do not elaborate it here. For details see Dave[23].

Given program segment

```
procedure pr (a, b, c, d: integer; var e, f, g: integer);
  var i: integer;
  begin
    repeat
          a: = b + c;
          b: = c * d;
          c: = e + f;
          if ((c * a) < > e) then
          begin
            pr1 (a, b, c, d);
            e: = e + f;
            c: = b + g
  end else
            pr1 (a, e, f, g);
          for i: = a + 1 to a + 9 do c: = c + 1;
    until (a < > b)
  end;
```

3. Dependence analysis

In this section, we define data dependences, and describe simple and interprocedural dataflow analysis and array subscript analysis which are useful in determining the data dependences.

3.1. *Data dependences*

We have already explained in Section 2.2 the concept of data dependence intuitively. Now, we shall define the following sets which are used in explaining the parallelisability conditions.

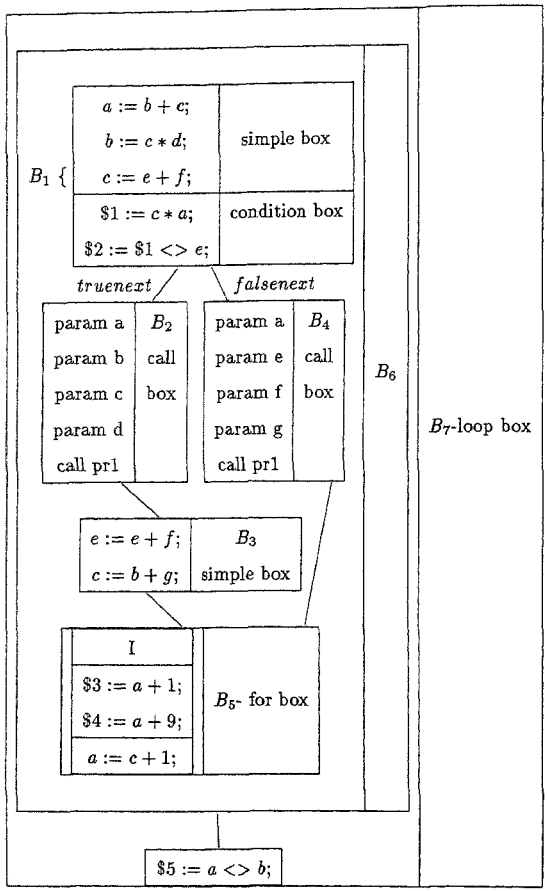

FIG. 6. Boxgraph for the program in Section 2.2.
● B_1 and B_5 are the starter and the ender, respectively, of B_6; ● B_5 is the endif box of B_1; ● B_3 is the exit box of B_2; ● B_5 is the exit box of B_3 and B_4.

a) *Changeset:* Changeset of a box can be defined as the set of memory locations (variables) which are *changed* when the box is executed.

b) *Useset:* Useset of a box can be defined as the set of memory locations (variables) which are *used* (read only) when the box is executed.

Consider S: $a := b + c$;

T: for $i := j * k$ to l do W;

Changeset of $S = \{a\}$

Useset of $S = \{b, c\}$

Changeset of $T = \{i\} \cup$ changeset of W

Useset of $T = \{j, k, l\} \cup$ useset of W

Now various types of data dependences between the two boxes can be defined.

(1) *Antidependence:* A non-empty intersection of u_1 and c_2 implies antidependence between a box with useset u_1 and a box with changeset c_2.

(2) *True dependence:* A non-empty intersection of c_1 and u_2 implies true dependence between a box with useset u_2 and a box with changeset c_1.

(3) *Output dependence:* If the intersection of c_1 and c_2 is not empty then it implies output dependence between boxes with changesets c_1 and c_2.

When two boxes have any one of anti, true, or output dependence between them, then they cannot be executed in parallel. For example, consider

Box 1: $S1: v1 := v2; S2: v3 := v4;$

Box 2: $S4: v2 := v5; S5: v5 := v1; S6: v3 = v6;$

$U1 = \{v2, v4\}$

$U2 = \{v5, v1, v6\}$

$C1 = \{v1, v3\}$

$C2 = \{v5, v2, v3\}$

Since, $U1 \cap C2 = \{v2\}$,

$C1 \cap C2 = \{v1, v3\}$, and

$C1 \cap C2 = \{v3\}$.

Box1 and Box2 have all three dependences. Hence, they cannot be executed in parallel. The ill effects of executing them in parallel are: If $S4$ executes before $S1$, then $v1$ gets a wrong value $v5$ instead of old values of $v2$. If $S5$ executes before $S1$, then $v5$ gets a wrong value (old $v1$) instead of new $v1$. If $S6$ executes before $S2$, then $v3$ gets a wrong value ($v4$), instead of $v6$.

3.2. *Simple dataflow analysis*

3.2.1. *Storing sets for dataflow analysis*

The compiler has been written in *C* language which does not provide set-type dat structures[32]. But *C* permits powerful bit operations to be performed on unsigned integer We took advantage of this fact and stored the sets in arrays of unsigned integers. Wheneve a changeset or useset is to be formed, the required number of unsigned integers are allocate using the Calloc function in *C*.

The required number of bits is nothing but the total number of variables which may b possible members of the set. For each procedure, this number is different and is stored a a field called datasize in procedure node in the symbol table. Datasize of a procedure i the sum of number of variables declared as local, number of parameters and the datasiz of the parent (the one which is immediately nesting the procedure) procedure.

Every variable declared in the visible scope of a procedure is allocated one bit. If a particular variable is used or changed in the procedure then the corresponding bit is on; otherwise it is off. The assignment of bits to variables is done by ordering the variables. The ordering is guided by the declarations in the procedure. For every procedure, the ordering is: local variables, value parameters, reference parameters, and global variables (in the same order).

In any declaration, the order in which the variables are declared is assumed for the bits also. To illustrate this, we present the following example.

```
program p(input, output);
var a, b, c:real;
  procedure pr(d, e:real; var f:real);
    var g, h:real;
```

Shown above is the declaration of a procedure. For the procedure *pr*, the ordering of variables is

$g, h, d, e, f, a, b, c.$

The set $\{b, c, g\}$ will be represented by bits as 10000011.

3.2.2. *Computation of sets for dataflow analysis*

We now show how to compute change and use sets for different types of boxes. For most of the cases, the sets are computed at parsing time itself. For an assignment statement, the changeset is the left hand side variable or an array name and the useset consists of all the variables involved on the right hand side. For an array reference whether on the right or the left hand side (of an assignment statement), the variables involved in its subscripts are included in the useset of the statement.

The useset (changeset) of a SIMPLE box is nothing but the union of usesets (changests)

of the assignment statements which constitute it. For a CONDITION box, the variables involved in the condition of the corresponding if-then-else statement are also included in the useset. For a FOR box, the useset is the union of usesets of all the boxes which constitute the body of the FOR box. The variables involved in the expressions of the limits of the corresponding for-loop are also included in the useset. The changeset is the union of changesets of all the boxes which constitute the body of the FOR box. The changeset (useset) of a LOOP box is found by computing the union of changesets (usesets) of all the boxes which constitute the body of the box. The changeset and useset of a COMPOUND box are also computed in similar manner. The computation of the sets of a CALL box is discussed later.

3.3. *Interprocedural dataflow analysis (IPDFA)*

In this subsection, we discuss interprocedural dataflow analysis which is crucial for any dataflow analysis. IPDFA in the presence of array references will be discussed in Section 3.2. Our method is based on the algorithm for IPDFA[30].

3.3.1. *Need for IPDFA*

IPDFA is carried out in the presence of call statements. While parsing, the changeset and the useset of a CALL box are computed as follows.

All the parameters and reference parameters listed in the procedure call are included in the useset and changeset, respectively.

For example, let the declaration for a procedure be

procedure *pp* (*i*1, *i*2, *i*3: real; var *v*1, *v*2: integer);

Let it be called as

pp (*j*1 ∗ *j*2, *j*3, *j*4, *j*5, *j*6);

The CALL box corresponding to the call of *pp*, during parsing, will have

useset = {*j*1, *j*2, *j*3, *j*4, *j*5, *j*6},

changeset = {*j*5, *j*6}.

Since Pascal allows access to global variables in a procedure, the calculated sets for the CALL boxes during parsing are only subsets of the actual sets. In the above example, if *pp* uses a variable *v* declared in any of the procedures which nest *pp*, then *v* should be included in the useset. This explains the need for IPDFA. The function of IPDFA is to update the sets by inspecting all the calls in the program in a separate pass after parsing is completed.

3.3.2. *Shifting*

Before putting forth an IPDFA algorithm, the concept of shifting is presented. Since for every procedure a boxgraph is created, the useset and the changeset of a procedure can

be easily found out by computing the union of the corresponding sets of all the boxes in the boxgraph (of the procedure). These sets are stored as bit arrays as explained earlier. The data size of a procedure has two parts: one consists of its variables including local variables, value and reference parameters and the other only global variables. The process of shifting updates the sets in such a way that only global variables are retained in the sets.

The shifting algorithm for a procedure call is shown below:

Algorithm: SHIFT.

Input: A set S, allocated as unsigned integers (as explained earlier).

(2) A procedure D which is called.

(3) A procedure R which calls D.

Output: A set T with proper shifting carried out.

Method: If a procedure P is immediately nesting a procedure Q, then P can be called as parent of Q.

Let Datasize $(D) = d$,

Datasize (Parent of D) $= e$.

Step 1. $U =$ set obtained by left shifting of S by $d - e$ bits.

Step 2. If R is the parent of D then $T = U$.

If R and D are at the same level, then

$T =$ set obtained by right shifting of U by $d - e$ bits.

3.3.3. *IPDFA Algorithm*

The IPDFA algorithm is as follows:

```
Var changed:boolean;
        changed: = TRUE;
(* This while-loop is to traverse though all the calls in the program repeatedly until changed
becomes false (i.e., The usesets and changesets have not changed w.r.t. the previous
iteration.).*)
While (changed) do begin
        changed: = FALSE;
(* This for-loop takes all the procedures in the input program one by one. *)
For each procedure P in the input program do begin
(* This for-loop takes all the procedure calls made in P one by one. *)
        For each called procedure q with corresponding box B in P do begin
(* Finding out the global variables which are used in the called procedure by SHIFT. *)
        S1 = SHIFT (useset (Q));
```

```
(* Updating the useset of the corresponding CALL box. *)
        useset (B) = useset (B) ⋃ S1;
(* Computing S2 which represents the new useset formed due to the effects of the call. *)
        S2 = useset (P) ⋃ S1;
(* If the new useset is different from the old one, then update and report change by making
changed as TRUE. *)
        If (S2 < > useset (P)) then begin
        changed: = TRUE;
        useset (P): = S2;
        end
(* Finding out the global variables which are changed in the called procedure by SHIFT. *)
        S1 = SHIFT (changeset (Q));
(* Updating the changeset of the corresponding CALL box. *)
        changeset (B) = changeset (B) ⋃ S1;
(* Computing S2 which represents the new changeset formed due to the effects of the
call. *)
        S2 = changeset (P) ⋃ S1;
(* If the new changeset is different from the old one, then update and report change by
making changed as TRUE. *)
        if (S2 < > changeset (P), then begin
          changed: = TRUE;
          changeset (P): = S2;
        end
    end
end.
```

The IPDFA algorithm repeatedly traverses through all calls in the input program. While traversing, it keeps on updating various data details of CALL boxes and procedures. For each traversal, it records whether any change occurs or not with respect to the previous traversal through the boolean *changed*. When no change occurs w.r.t. the previous traversal, it stops.

IPDFA computes sets for CALL boxes only. Hence, after IPDFA is over, another pass over the boxgraph is required to compute the sets for LOOP and FOR boxes which incorporate CALL boxes in them.

3.4. *Array-subscript analysis*

In this Section, we shall discuss how to carry out dataflow analysis in the presence of subscripted variables. Array subscript analysis plays a very important role in parallelizing compilers because programs use arrays extensively and a lot of parallelism is lost if we do not execute two boxes in parallel whenever they use a common array variable, even though the two boxes may be operating on disjoint parts of the array.

3.4.1. *Need for subscript analysis*

Consider the following statements, S and T:

$S: a[2] := b + c;$

$T: d := a[3];$

The changeset of S is $\{a\}$, and the useset of S is $\{b, c\}$.

The changeset of T is $\{d\}$, and the useset of T is $\{a\}$.

Since the intersection of the changeset of S and the useset of T is nonempty, the compiler will not declare them as parallelizable. However, since $a[2]$ and $a[3]$ do not represent the same memory location, S and T can be declared as parallelizable. This suggests that the conventional dataflow analysis based on sets is not sufficient to extract parallelism in the presence of subscripted variables.

3.4.2. *Complexity of the problem*

Essentially our aim is to find out whether a common location in an arrary is being referred to or not by two array references. At compile time, it is difficult to find out this information. Consider two array references such as $a[i]$ and $a[j]$. Unless symbolic computation is carried out, nothing about i and j, which can be useful in determining whether $a[i]$ and $a[j]$ refer to a common memory location or not, is known. Symbolic computation is very expensive and not very practical to implement. Therefore, less-expensive approximate tests have been developed which can determine whether two array references access a common memory location or not. These tests give assurance that two array references *will not* access a common memory location but fail to assure that both will definitely access a common memory location. This is a conservative but safe approach towards the problem. We have implemented those tests which are less expensive but give fairly good results.

3.4.3. *GCD Test*

Let us consider two array references $r[a_0 + a_1 * x_1 + a_2 * x_2 + \ldots + a_n * x_n]$ and $r[b_0 + b_1 * y_1 + b_2 * y_2 + \ldots + b_m * y_m]$. The subscripts are linear functions† of the *integer* variables $x_1, x_2, \ldots, x_n$, and $y_1, y_2, \ldots, y_m$, respectively. $a_0, a_1, a_2, \ldots, a_n\, b_0, b_1, b_2, \ldots, b_m$ are all integer constants.

Both will refer to the same memory location in the array r when

$$a_0 + a_1 * x_1 + \ldots + a_n * x_n - b_0 - b_1 * y_1 - \ldots - b_m * y_m = 0;$$

$$\text{i.e.,} \quad a_1 * x_1 + \ldots + a_n * x_n - b_1 * y_1 - \ldots - b_m * y_m = b_0 - a_0. \tag{1}$$

The above is nothing but a linear Diophantine equation. From the theory of Diophantine

†The assumption that subscripts are linear functions of integer variables is generally valid in scientific programs. This is based on observations of mathematical software packages.

equations[33], we have, for an equation of the form $c_1 * v_1 + c_2 * v_2 + \ldots + c_k * v_k = c$, where $k >= 2$, and $c_1, c_2, \ldots, c_k$ are integer constants and $v_1, v_2, \ldots, v_k$ are integer variables.

a solution exists iff

gcd $(c_1, c_2, \ldots, c_k)$ divides c.

Therefore, for equation (1), a solution exists iff

$$\gcd(a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m) \text{ divides } (b_0 - a_0). \tag{2}$$

This means that if condition (2) is satisfied, then there exist integer values for the variables $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m$ such that (1) is satisfied and hence the two array references *may* access common locations in array r. If (2) is not satisfied then it is definite that (1) cannot be satisfied for any integer value of the variables and hence the array references will not access any common memory location in array r. Of course, even if (2) is satisfied, there are chances that the integer variables never take value at run time such that (1) is satisfied and hence the two array references never access common memory locations in array r. Satisfaction of (2) only implies *existence* of integer values for variables which satisfy (1). Condition (2) is the well-known GCD test.

Although the GCD test is comparatively weak, it is very easy and fast and hence is always applied first[4,5].

3.4.4. *Banerjee's test*

If it is known at compile time that the values the variables take have upper and lower bounds then a more powerful test can be conducted.

Let us consider two array references $r[a_0 + a_1 * x_1 + a_2 * x_2 + \ldots + a_n * x_n]$ and $r[b_0 + b_1 * y_1 + b_2 * y_2 + \ldots + b_m * y_m]$. Let all xs and ys have some lower and upper bounds for the values they can take. For these array references to access a common memory location in array r,

$$a_0 + a_1 * x_1 + \ldots + a_n * x_n - b_0 - b_1 * y_1 - \ldots - b_m * y_m = 0. \tag{3}$$

We can without loss of generality, assume that $x_1 = y_1, x_2 = y_2, \ldots, x_k = y_k$, for some k less than or equal to m and n (k can be zero also).

Let

$$v_i = x_i = y_i \text{ and } c_i = b_i - a_i, \text{ for } i = 1, 2, \ldots, k.$$

Let

$$v_i = x_i \text{ and } c_i = a_i \text{ for } i = k+1, \ldots, n.$$

Let

$$v_j = y_i, c_j = b_i \text{ and } j = n + i - k, \text{ for } i = k+1, \ldots, m.$$

Let

$$l = m + n - k.$$

We have reduced equation (3) to

$$c_1 * v_1 + c_2 * v_2 + \ldots + c_l * v_l = a_0 - b_0. \tag{4}$$

Let $L_1, L_2, \ldots, L_l$ be the lower bounds and $U_1, U_2, \ldots, U_l$ be the upper bounds to the variables $v_1, v_2, \ldots, v_l$.

Before explaining Banerjee's test, we shall define positive and negative parts of a number.

Positive part

Positive part of a real number r can be defined as

$$\text{pos}(r) = 0 \text{ if } r < 0.$$
$$= r, \text{ otherwise.}$$

Negative part

Negative part of a real number r can be defined as

$$\text{neg}(r) = 0 \text{ of } r > 0$$
$$= r, \text{ otherwise.}$$

For any integer variable v, with upper bound U and lower bound L,

i.e., if $L <= v <= U$, then,

$$\text{neg}(c)*(U - L) <= c*(U - L) <= \text{pos}(c)*(U - L)$$

which in turn yields,

$$\text{neg}(c)*(U - L) + c*L$$
$$<= c*v$$
$$<= \text{pos}(c)*(U - L) + c*L. \tag{5}$$

Let

$$LB_i = \text{neg}(c_i)*(U_i - L_i) + c_i*L_i,$$
$$UB_i = \text{pos}(c_i)*(U_i - L_i) + c_i*L_i.$$

From (5), for $i = 1, 2, \ldots, l$, we get the set of inequalities:

$$LB_i <= c_i*v_i <= UB_i.$$

Adding all the inequalities, for $i = 1, 2, \ldots, l$, we get

$$LB_1 + LB_2 + \ldots + LB_l$$
$$<= c_1*v_1 + c_2*v_2 + \ldots + c_l*v_l$$
$$<= UB_1 + UB_2 + \ldots + UB_l.$$

Applying (4), we get

$$LB_1 + LB_2 + \ldots + LB_l$$
$$<= a_0 - b_0$$
$$<= UB_1 + UB_2 + \ldots + UB_l. \tag{6}$$

All the *LB*s and *UB*s can be calculated by knowing *cs*, *Ls*, and *Us* at compile time. (6) is known as Banerjee's test. Proofs of the above properties and other details can be found elsewhere[3,5,9].

The Banerjee's test, like the GCD test, assures us that two array references will not access a common memory location but fails to assure that two array references will definitely access a common memory location. This approach is conservative but on the safer side. The Banerjee's test takes more time than that of GCD test, but gives better results. We first apply the GCD test and then proceed to Banerjee's test. There are examples where GCD test fails and Banerjee's test succeeds, but the *vice versa* also is not uncommon. Hence these two tests are complementary in nature and should be used together to obtain better results.

3.4.5. *Multidimensional arrays*

Arrays having more than one dimension need special treatment. Here we shall discuss two approaches toward the problem. We have implemented both of them.

Subscript by subscript analysis: Let there be two array references $a[r_1, r_2, \ldots, r_n]$ and $a[s_1, s_2, \ldots, s_n]$ where r_i and s_i, $i = 1, 2, \ldots, n$ are linear functions of variables indicating subscript expressions. In subscript by subscript analysis, all the subscripts are checked one by one. If all of them show failure (*i.e.*, the tests in Section 3.4 indicates access to common memory locations), then access conflict is assumed, otherwise the two references are assumed to be access conflict free. That is, for the above two array references, both will access a common memory location if for *all* values of i in the range 1 to n, r_i and s_i fail tests in Sections 3.4.3 and 3.4.4.

Linearization: Burke and Cytron[1,2] showed that linearization of multidimensional arrays is as important as subscript by subscript analysis. It showed some examples in which independence (*i.e.*, no access conflict) was not detected by subscript by subscript analysis but was detected by linearization.

We observe the following facts.

- Memory can be viewed as a one-dimensional array MEM.
- The function that maps array references with multiple subscripts into their locations in MEM is linear with respect to the subscript (irrespective of whether row- or column-major arrangement is assumed).
- Due to this, if all the subscripts of an array reference are linear functions then the corresponding subscript of MEM will also be a linear function.

Let the declaration of an array a be

var $a[l1..u1, l2..u2, \ldots, ln..un]$.

Let the target array reference be $a[r1, r2, \ldots, rn]$ and $r1, r2, \ldots, rn$ be all linear functions.

Let $a[l1, l2, \ldots, ln]$ be represented as $a[0]$. By assuming row-major arrangement

$a[r1, r2, \ldots, rn]$ can be represented as $a[f]$, (from Horwitz and Sahni[34]) where

$$f = (r1 - l1)(u2 - l2)(u3 - l3)\ldots(un - ln) + (r2 - l2)(u3 - l3)(u4 - l4)\ldots(un - ln) + \ldots + (rn - ln).$$

All the array references can be represented by their corresponding single-dimension subscripts as shown above and then any of the tests in Sections 3.4.3 and 3.4.4. can be applied on them.

3.4.6. *Implementation details*

For each box, the use-array and change-array references are stored in a linked list with each node pointing to a reference. A reference is also stored as a linked list with each node pointing to a subscript. The subscript stores the corresponding linear function with the bounds of the variables, if available.

A variable may have bounds because of two reasons:

- for a loop index, the index may be bounded on lower and upper sides, or
- by declaration of subrange data types, *e.g.*, the declaration var i: 1 .. 5; indicates that i is bounded by 1 and 5.

During parsing, special routines which have been provided are used to detect whether an expression is a linear function or not. It is very likely that some subscripts are not linear functions. For such array references, dependence (*i.e.*, access conflict) is assumed.

3.4.7. *Interprocedural dataflow analysis*

All the array references present in the body of a procedure are collected. When a procedure X is called, for the CALL box which contains the call to X, IPDFA transforms the array references of the procedure in the following manner.

- If the array is locally declared then that reference is discarded.
- If the array is declared as a value or reference parameter then its name is changed to the corresponding one in the call statement.
- All the variables of the subscripts declared in value and reference parameters are changed as per the call.

With the above transformation, IPDFA is carried out as explained earlier.

The above actions are carried out to record the impact of the reference inside the procedure on outside environment. For example,

```
program pp;
  var i,j: integer; a:array;
  procedure pr(x:array);
```

```
begin
...
...:= x[5*i + 4];
end;
begin
pr(a); (* Box b1. *)
a[5*i + 3]:= ...; (* Box b2. *)
end.
```

By conventional analysis and IPDFA, boxes $b1$ and $b2$ will be declared as non-parallelizable because of a. But our IPDFA will convert the reference $x[5*i+4]$ in pr to $a[5*i+4]$ in box $b1$. This will make $b1$ and $b2$ as parallelizable. This shows the high importance of carrying out IPDFA with subscripted variables in the manner shown above.

Usage of GCD and Banerjee's test in actual detection of parallelism in programs is described in Section 4.

4. Detection of parallelism

In this section, we shall describe the main algorithm for parallelization. For detection of parallelism, there are two considerations: data and control.

4.1. *Data considerations*

A routine called *coexecutable* takes two boxes and finds out whether they are suitable for parallel execution (in which case they are said to be parallelizable) or not based on data considerations.

Let u_1, and u_2 be the usesets and c_1, and c_2 the changesets of boxes B_1 and B_2, respectively. It is assumed that B_1 N B_2 in the corresponding boxgraph.

Let

set_1 = intersection of u_1 and c_2;

set_2 = intersection of c_1 and u_2;

set_3 = intersection of c_1 and c_2.

If set_1, set_2, and set_3 are empty (which means none of the three dependences exist) then the boxes B_1 and B_2 are declared as parallelizable by the routine *coexecutable* because it is obvious that parallel execution of B_1 and B_2 does not affect each other. A compound box B having elements B_1 and B_2, useset $u = u_1 \cup u_2$, changeset $c = c_1 \cup c_2$ is formed. When references to the same array are present in any two sets to be intersected, the GCD and

Banerjee's tests (see Section 3.2) are applied to all possible pairs of references to check if dependences exist.

4.2. *Control considerations*

4.2.1. *Lemma*

Consider a boxgraph such as $b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b_n$.

If b_1 is not parallelizable with b_2, b_2 is not parallelizable with $b_3, \ldots, b_{n-1}$ is not parallelizable with b_n (all due to data considerations), then b_1 is not parallelizable with b_n. For proof see Dave[23].

4.2.2. *Control dependence*

We say that a box b is control dependent on box c iff

- there is a path from c to b in the boxgraph
- there exists a condition box d not equal to b, in the path from c to b in the boxgraph such that there is no path from the endif box of d to b in the boxgraph.

If b is control dependent on c then b and c cannot be declared as parallelizable. For example, consider the following statements:

S_1: if($a < > b$) then

S_2:$c := d$;

Whether S_2 will be executed or not depends on the result of S_1. Hence, S_1 and S_2 cannot be executed in parallel.

4.2.3. *Path analysis algorithm*

From the lemma given above, for a boxgraph $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow \ldots b_n$, it is sufficient to analyse b_1 with b_2, b_2 with $b_3, \ldots$ and so on.

The control dependences and the lemma of the previous section lead to the following path analysis algorithm

The path analysis algorithm takes the starter and the ender of a boxgraph and modifies the boxgraph so that it shows parallelism explicitly.

procedure *pathanalysis* (*blfrom, blto*):

(* *blfrom* is the starter of the boxgraph and *blto* is the ender of the boxgraph. *)
begin

$b1 = blfrom$;

while ($b1 < > blto$) do begin

```
    if (b1 is a CONDITION box) then begin
        pathanalysis (truenext of b1, endif of b1);
        pathanalysis (falsenext of b1, endif of b1);
    end
    else begin
        b2 = next of b1;
        if (b2 = NULL or b2 = NULLBOX) then goto label;
        if (b2 is not CONDITION box) then
          b1 = datatest (b1, b2);
        else b1 = b2;
    end;
    if (b1 = NULL or b1 = NULLBOX) then goto label;
  end
  label:;
end;
```

The datatest routine carries out the following tasks:

It takes two boxes b_1 and b_2 as parameters, and

- by data consideration, it finds out whether they can be executed in parallel using the routine *coexecutable* (Section 4.1),
- if they cannot be executed in parallel then b_2 is returned, otherwise,

– a compound box b is formed from b_1, and b_2. Box b has as its changeset the union of the changesets of b_1 and b_2 and as its useset the union of the usesets of b_1 and b_2. The nexts of b will be exactly according to the nexts of b_2.

– If b_2 is the endif box of some box c then next of b_2 is made the endif box of c. Also duplicates of b_2 are made and are attached to all the boxes having next as b_2. If next of b_2 is NULL and b_2 is an endif box then a NULLBOX is created and this is taken as next of b and as nexts of all the duplicates made. For example, for the program segment given below, Fig. 7 shows the corresponding boxgraphs before and after the path analysis.

```
if (a < > b) then begin
  c: = d + e;
end
```

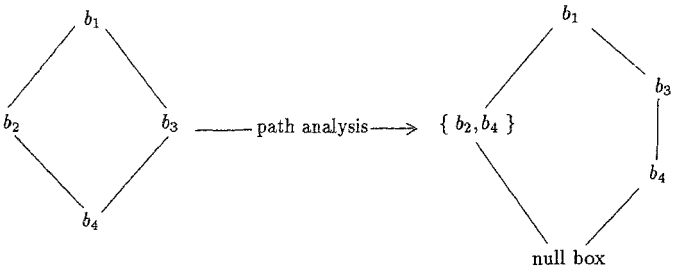

FIG. 7. Example for path analysis.

else

$f := 2$;

$g := f + 1$;

box b_1: $a <> b$;

box b_2: $c := d + e$;

box b_3: $f := 2$;

box b_4: $g := f + 1$;

Notice that b_2 and b_4 can be executed in parallel.

– Next of b_2 is returned.

For each subroutine the corresponding boxgraph is modified by the major analysis routine. Major analysis routine calls the path analysis routine by passing the starter and the ender of boxgraphs. All the LOOP boxes and the FOR boxes which have their own boxgraphs are also modified by calling the path analysis algorithm. All the FOR boxes whose corresponding for-loops can be unrolled are identified as PARFOR boxes.

4.2.4. *Loop unrolling*

Loop unrolling is very important in parallelization. In a for-loop, if different iterations of the for-loop are data- independent (*i.e.*, data computed in one iteration is not used in any other iteration) then these iterations can be executed in parallel. For example,

```
for i := 1 to 9 do begin
        a[i] := i + c;
        b[i] := i*c;
        d[i] := c - i;
    end;
```

All the iterations of this loop can be unrolled and executed in parallel with different values (1 to 9) of *i* supplied to them. Loop unrolling can be carried out by comparing the boxgraph attached to the corresponding for-loop with istelf under data considerations. The boxgraph is supplied to the routine *coexecutable* which determines whether the loop can be unrolled or not. If the loop can be unrolled then the FOR box corresponding to it is converted to a PARFOR box. The meaning of a PARFOR box is that the boxgraph attached to it (body of the loop) is to be executed in parallel for different values of its index starting from lower to upper limit.

4.2.5. *Parallelization of CALL statements involving recursion*

Each CALL statement in the program is converted to a CALL box as shown earlier. Our method can declare a CALL box as parallelizable with any other type of boxes. This automatically takes care of recursion. For example,

```
procedure pr;
begin
  ...
  pr;
  ⟨rest of the body of the procedure which does not depend on the data computed before⟩
  {say this is B}
end
```

In the given program, body *B* can be declared as parallelizable with the call *pr*. This means that the next recursive call is executed in parallel with the present execution of the body of the procedure. One classic example is the fibonacci sequence computation algorithm. The procedure calls itself twice while executing. Both the calls can be executed in parallel as detected by our compiler.

5. Implementation of parallelism

In the previous sections, we described how to generate the boxgraphs and convert them to show parallelism explicitly. Conversion of these boxgraphs to assembly code depends on specific machine details. We have implemented our compiler on the ORG Supermax machine which is a shared memory multiprocessor with two 68020 processors running on UNIX operating system V.3 at a speed of 25 MHz. We generate 68020 assembly code.

The ORG Supermax has two processors, each with 4 MB of local memory. But the processors can access the other processor's local memory by declaring it as a shared memory. We have designed our scheme to suit this machine. The aim of our scheme is to use both the processors as efficiently as possible during the execution of the program. To achieve this purpose, we have chosen to implement dynamic scheduling of processes.

Dynamic scheduling has the following advantages over static scheduling.

- Dynamically *created* processes cannot be implemented using static scheduling. We need to create processes dynamically in the case of recursive procedure calls.
- If the number of asynchronous parallel processes are more than the number of processors available, then dynamic scheduling gives much better performance.
- In the case of heavy nesting of processes and limited number of processors, dynamic scheduling is a good choice.

Dynamic scheduling has the following disadvantages over static scheduling.

- The scheduling time overhead is more in dynamic scheduling than in static scheduling. If process execution times are not sufficiently large then this can affect the program execution speed in a serious manner.
- Static scheduling is more efficient in the case of synchronous or almost synchronous parallel processes.
- In static scheduling, the processes are assigned to the processors statically. Due to this, the local data used by the processes can be put into local memories of the corresponding processors. If accessing local memory is faster than accessing shared memory then static scheduling may improve the performance considerably.

In spite of these disadvantages, we preferred to implement dynamic scheduling because the advantages overweigh the disadvantages.

We create two programs from the imput Pascal program. One is the master and the other is the slave. All the data are stored in the shared memory so that both the programs can access it at any time. The programs also share a common message queue. The common message queue contains messages to start a new process, execute it completely, or halt it.

In the beginning, the master calls the main procedure and starts execution. The slave waits for a message to be put into the common message queue. As soon as the common message queue obtains a message to execute a process, the slave removes the message from the queue and starts executing the process indicated by it.

Whenever parallel processes are to be executed by either the master or the slave program, all the parallely executable processes are put into the common message queue. A common identifier is assigned to all these parallel processes to identify them as children of the same process. This is essential because, due to nesting of processes, they can create other children processes. Further, the master and the slave programs also keep track of the number of child processes with a given identifier which have not yet completed execution. This is required to resume the execution of the parent process which created these child processes.

After putting messages to execute these processes into the common message queue, the program (slave or master) which created them takes a process from the message queue and starts executing it. In the meantime, if the other program becomes free then that will also take a process from the queue and start executing it. Each time a program completes execution of a process, it carries out the following tasks:

- It puts a message into the queue saying that the process has been completed.
- If it has created parallel processes, then it checks whether execution of all the child processes it created has been completed or not. This can be done by using the identifier supplied when the parent process puts the child processes into the queue.
- If all the processes created by it have been executed then it resumes execution of the parent process which created these child processes.
- If all the child processes have not been executed completely then it waits for a message to execute a process to be put into the queue (if not already available).

The first implementation was not efficient because it generated C programs instead of assembly code and used UNIX system calls for management of message queues. The current implementation (ongoing) produces assembly code and incorporates autoscheduling techniques and uses sophisticated guided self-scheduling strategies for loop scheduling[15]. Code generation in detail is beyond the scope of this paper and will be reported after carrying out a thorough performance test of the autoscheduling technique.

6. Results, conclusions, and future directions

6.1. *Summary*

A highly ambitious compiler writer will feel dissatisfied after writing a parallelizing compiler because:

(1) Some of the sequential algorithms which can be easily parallelized by hand cannot be parallelized even by using the most advanced parallelizing techniques developed to date. Quicksort is a good example.
(2) Programs written in languages like Pascal, Ada and C use pointers extensively. Because the area of pointer dataflow analysis is still not well developed, we have a none-to-happy situation regarding parallelization of non-numerical programs.

However, an optimistic compiler writer will feel satisfied because:

(1) For numerical programs, the performance of parallelizing compilers is quite good.
(2) Parallelizing compilers are cost effective when compared to human beings doing a similar job of parallelizing existing large programs.
(3) Programs parallelized using parallelizing compilers have higher reliability than programs parallelized by hand.

Our approach to the problem has been to extract the maximum parallelism possible in a reasonable amount of compilation time. Moreover, our intermediate representation *viz.*, boxgraph is specially designed to suit shared memory multiprocessors, and has the following advantages over other representations:

(1) For well-written programs, our box-choosing algorithm chooses the boxes in such a way that each box will be sufficiently large. This reduces the scheduling overheads considerably in shared memory-multiprocessors, which in turn increases the speedup of the parallelized program.

(2) Analysing boxgraphs is much easier and hence compilation time is less.
(3) Sequential algorithms for analysing boxgraphs are such that in future, parallel algorithms can be developed without too much effort so that compilation time can be brought down even further.
(4) For any block-structured language, the method to construct boxgraphs is quite simple.

We have implemented various techniques of array subscript analysis. Our implementation and interprocedural dataflow analysis in the presence of subscripted variables have given fairly good results.

The back end of the parallelizing compiler to generate assembly code is about to be complete. We have used autoscheduling techniques combined with guided self-scheduling of loops to implemented parallel programs.

6.2. *Results*

The compiler is about 12000 lines of C-code and it took nearly 12 months to finish. We have tested it on a large number of programs and have found satisfactory results. We include a list of some of the programs (Table I) that we have tested together with some comments on the performance of our compiler with respect to the programs. The actual listing of the programs and the boxgraphs with parallelism explicit have been included in Dave's thesis[23]. We have taken these programs from well-known books[35–38].

The above results indicate that we have been able to achieve our target, namely, detection of a reasonable amount of parallelism in a reasonable amount of time. The time for parallelization is quite small. *The average speed of our parallelizing compiler is approximately 100 lines of source code per second.*

6.3. *Conclusions*

The high speed of our parallelizing compiler is basically due to the set structure which has been extensively used in our compiler. The following factors may improve the amount of parallelism detected by our parallelizing compiler: (i) use of U-D, D-U, and D-D chains, (ii) constant propagation, (iii) induction variable elimination, (iv) loop interchanging, and (v) more sophisticated array subscript analysis.

However, these factors will also increase the time required for parallelization. We intend to implement these techniques in a future version and study their effect on actual programs.

6.4. *Future research directions*

(1) A large amount of work needs to be done in the area of pointer dataflow analysis on the lines proposed[23].
(2) Parallel algorithms need to be developed for the existing techniques of detecting parallelism, thus decreasing the compilation time.

Table 1
Performance statistics of the parallelizing compiler

Sl no.	Program name	Function*	Reference	No. of lines of source	Time taken for parallelization	Total no. of loops	No of loops parallelized	No. of procedure calls	No of calls parallelized	No. of cobegin-coend blocks
1	ex	1	35	165	6.3 s	19	3	9	2	5
2	am	2	35	129	3.6 s	15	8	11	6	8
3	cel	3	35	78	0.7 s	0	0	7	1	1
4	de	4	35	75	0.8 s	9	5	4	1	2
5	ff	5	36	48	0.5 s	6	4	3	0	0
6	fibo	6	37	16	0.1 s	0	0	3	2	1
7	smr	7	36	17	0.1 s	2	2	0	0	0
8	mm	8	35	48	0.4 s	4	1	3	1	3
9	trasales	9	38	85	1.7 s	9	5	5	1	3
10	fi	10	35	96	1.1 s	6	0	8	3	3

**Functions of the programs*

1. adi: This is an implementation of a relaxation method called alternate direction implicit.
2. amoeba: It is a procedure for solving the problem of multidimensional minimization by the Downhill-Simplex method proposed by Nelder and Mead.
3. cel: It is a procedure to compute complete elliptical integral.
4. des: It is an implementation of data encryption standard integral.
5. dff2: It is a procedure to compute 2-dimensional fast Fourier transform assuming that the one-dimensional fast Fourier transformation procedure is supplied.
6. fib: It computes the famous fibonacci numbers.
7. It is a part of an implementation of a simple multigrid relaxation algorithm.
8. mmid: Implementation of the modified midpoint method.
9. This is an implementation of the famous travelling salesman problem.
10. fit: A program for fitting data to a straight line.

(*Contd*)

Table I (*Contd*)

Comments on the programs

1. A very large program chosen mainly to demonstrate some of the features of the parallelizing compiler. Nesting of parallelism is observed. A reasonable degree of parallelism was detected.
2. In this program, a very high degree of parallelism was detected. With efficient scheduling mechanisms the parallel program can achieve substantial speedup.
3. It is a small numerical program. Due to dependences, the parallelism detected is almost nil.
4. Although this program is small, a very high degree of parallelism was detected.
5. Here is an example where 4 out of 6 loops were detected to be parallelizable. This highly parallelizable procedure is used in numerical programs very frequently, and hence explains the importance of parallelizing compilers.
6. Two recursive procedure calls were parallelized. Hence, the degree of parallelism detected is very high. Fibonacci numbers are used repeatedly in sorting and other programs.
7. This partial program is given to detect how the injection (first loop) and prologation (second loop) operations in a simple multigrid relaxation algorithm are unrolled by the parallelizing compiler.
8. High degree of parallelism was detected in this small program. The variable swap was the bottleneck for not parallelizing loops 12 and 13.
9. No comments.
10. The only important parallelization is of two procedure calls of sqrt. The degree of parallelism detected is fair.

(3) New techniques using parallel algorithms need to be developed to extract more parallelism which is being discarded only because their sequential versions are very expensive.
(4) Symbolic computation should perhaps be used in a limited fashion to extract parallelism.
(5) An efficient implementation of the back end of our parallelizing compiler with self-scheduling techniques incorporated into it is underway. This will be used to measure the performance of our parallelizer with respect to real-life application programs.
(6) Data-partitioning techniques suitable for parallelizing programs to be run on message-based multiprocessors need to be developed.

References

1. HWANG, K. AND BRIGGS, F. A. *Computer architecture and parallel processing*, 1984, McGraw-Hill.
2. PADUA, D. A., KUCK, D. J. AND LAWRIE, D. H. High-speed multiprocessors and compilation techniques, *IEEE Trans.*, 1980, **C-29**, 763–776.
3. BANERJEE, U. *Speed up of ordinary programs*, Ph.D. Thesis, 1979, University of Illinois at Urbana-Champaign.
4. WOLFE, M. J. *Optimizing supercompilers for supercomputers*, Ph.D. Thesis, 1982, University of Illinois at Urbana-Champaign.
5. ALLEN, J. R. *Dependence analysis for subscripted variables and its application to program transformation*, Ph.D. Thesis, 1983, Rice University, Houston, Texas.
6. ALLEN, J. R. AND KENNEDY, K. *Automatic translation of Fortran programs to vector form*, Rice University Reports, July 1984.
7. PADUA, D. A. AND WOLFE, M. J. Advanced compiler optimizations for supercomputers, *Commun. ACM*, 1986, **29**, 1184–1201.
8. WOLFE, M. Vector optimizations *vs* vectorization, *J. Parallel Distributed Computing*, 1988, **5**, 551–567.
9. WOLFE, M. AND BANERJEE, U. Data dependence and its application to parallel processing, *Inter. J. Parallel Programming*, 1987, **16**, 137–178.
10. ALLEN, F., BURKE, M., CHARLES, P., CYTRON, R. AND FERRANTE, J. An overview of the PTRAN analysis system for multiprocessing, *J. Parallel Distributed Computing*, 1988, **5**, 617–640.
11. TRIOLET, R., IRIGOIN, F. AND FEAUTRIER, P. Direct parallelization of call statements, *ACM SIGPLAN'86 Symp. Compiler Construction*, 1986.
12. BURKE, M. AND CYTRON, R. Interprocedural dependence analysis and parallelization, *ACM SIGPLAN'86 Symp. Compiler Construction*, 1986.
13. CALLAHAN, D. AND KENNEDY, K. Analysis of interprocedural side effects in a parallel programming environment, *J. Parallel Distributed Computing*, 1988, **5**, 517–550.
14. LI, Z. AND YEW, P. Efficient interprocedural analysis for program parallelization and restructuring, *ACM PPEALS*, 1988.
15. POLYCHRONOPOULOS, C. D. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers, *IEEE Trans.*, 1987, **C-36**, 1425–1439.

16. MIDKIFF, S. P. AND PADUA, D. A. Compiler algorithms for synchronization, *IEEE Trans.*, 1987, **C-36**, 1485–1495.

17. AIKEN, A. AND NICOLAU, A. Optimal loop parallelization, *ACM Proc. SIGPLAN'88 Conf. Programming Language Design and Implementation*, 1988.

18. POLYCHRONOPOULOS, C. D. *Advanced loop optimizations for parallel computers*, CSRD Report No. 664, University of Illinois at Urbana-Champaign, 1987.

19. POLYCHRONOPOULOS, C. D. *More on advanced loop optimizations*, CSRD Report No. 667, University of Illinois at Urbana-Champaign, 1987.

20. POLYCHRONOPOULOS, C. D. Compiler optimizations for enhancing parallelism and their impact on architecture design, *IEEE Trans.*, 1988, **C-37**, 991–1004.

21. FERRANTE, J., OTTENSTEIN, K. J. AND WARREN, J. D. The program dependence graph and its use in optimization, *ACM Trans. Programming Languages Systems*, 1987, **9**, 319–349.

22. HENDREN, L. J. AND NICOLAU, A. Parallelizing programs with recursive data structures, *IEEE Trans.*, 1990, **PDS-1**, 35–47.

23. DAVE, M. A. *A parallelizing compiler for Pascal*, M.Sc. (Engng) Thesis, 1989, Department of Computer Science and Automation, Indian Institute of Science, Bangalore.

24. HORWITZ, S., PFEIFFER, P. AND REPS, T. Dependence analysis for pointer variables, *Proc. ACM SIGPLAN'89 Symp. on Programming Language Design and Implementation*, June 1989.

25. GROB, L. S. Automatic exploitation of concurrency in C: Is it really so hard?, *Ultracomputer Note-140*, Ultracomputer Research Laboratory, New York.

26. ALLEN, R. AND JOHNSON, S. Compiling C for vectorization, parallelization, and inline expansion, *Proc. ACM SIGPLAN'88 Conf. on Programming Language Design and Implementation*, June 1988.

27. GUARNA, V. A. *Analysis of C programs for parallelization in the presence of pointers*, CSRD Report No. 695, University of Illinois at Urbana-Champaign, 1987.

28. COUTANT, D. Retargetable high-level allias analysis, *ACM Symp. on Principles of Programming Languages*, 1986, pp 110–118.

29. WEIHL, E. W. Interprocedural dataflow analysis in the presence of pointers, procedure variables, and label variables, *Seventh Annual ACM Symp. on Principles of Programming Languages*, 1980, pp 83–94.

30. AHO, A. V., SETHI, R. AND ULLMAN, J. D. *Compilers – Principles, techniques and tools*, 1986, Addison-Wesley.

31. JOHNSON, S. C. *YACC – Yet another compiler compiler*, 1975, CSTR 32, Bell Labs, Murray Hills, N. J.

32. KERNIGHAN, B. W. AND RITCHIE, D. M. *The C programming language*, 1977, Prentice-Hall.

33. NIVEN, I. AND ZUCKERMAN, H. S. *An introduction to the theory of numbers*, 1972, Wiley.

34. HOROWITZ, E. AND SAHNI, S. *Fundamentals of data structures in Pascal*, 1984, Galgotia Book Source, New Delhi.

35. PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A. AND VETTERLING, W. T. *Numerical recipes: The art of scientific computing*, 1988, Cambridge University Press.

36. TSENG, P. *A parallelizing compiler for distributed memory parallel computers*, Ph.D. Thesis, 1989, Carnegie Mellon University, CMU-CS-89-148.

37. KNUTH, D. E. *Fundamental algorithms*, 1983, Addison-Wesley/Narosa.

38. BRAWER, S *Introduction to parallel programming*, 1989, Academic Press.