# *Basrah Journal of Science*

*Section A*
Physics, Mathematics, Computer Science, Geology
Volume 17, Number 1, 1999

Basrah J. Science, A, Vol.17, No.1, 143-158, 1999

# A DATA FLOW ANALYSER TO DETECT PARALLELISM IN SEQUENTIAL PROGRAMS

Zainab S. Juma'a *  Nadia Y. Yousif
Computer Science Dept.,
College of Science, Basrah University,
Basrah - IRAQ


## Abstract

The detection of parallelism in sequential programs is very important aspect towards achieving parallel processing. In this paper, a method is presented to divide the sequential program into parts according to some rules. This includes a dependency analyser that follows the flow of data within the program to specify the locations of ***defining*** variables and those of ***using*** them in the whole program. This technique can analyse data dependency between statements and also within the iterations of the while loops. Thus, a suggestion can be drawn on how the sequential program can be implemneted in a multiprocessor system.



## Key words

Parallelism, data flow analysis, dependency relations, dependency analysis.


## Introduction

Detecting dependency between the statements of a sequential program involved partitioning the program into separated parts according to some formats or rules. Williams [1] has used a **Stanza** to represent a part of a sequential program written in Algol language. Her work was based on the Bernistien concepts [2] : the **fetch** from a memory location and the **store** in a memory location. The work in [1, 2] was based on finding the dependency relation between two adjacent stanzas (statements or parts) and depended highly on four Bernistien sets for using memory locations which are defined as follows :

1- W set : the set of variables fetched during execution of stanza,

2- X set : the set of variables stored during execution of stanza,

---

* **Current address : Dept. of Statistics, College of Administration & Economy, University of Basrah.**

## Entries

Allan and Oldehoeft [5] have proposed a data flow analysis procedure that transforms a high-level program into a data flow program suitable to be executed on a data flow machine. However, in this paper we follow the data analysis method of [5] but with the aim of finding the dependency relations between statements in order to explore parallelism in the sequential program.

A subset of Pascal language is chosen to write the resource programs for our automatic detector of parallelism , where each statement is represented by a data structure called *Entry* (defined below). The statements are classified into two parts :

(1) Simple: the assignment statement, and the condition part of a compound statement.

(2) Compound : the iterative statements (e.g., While ... Do and Repeat .. Until), and conditional statements (e.g., If ... Then, and If .. Then .. Else).

This partitioning of statements is vital for formulating different rules for each entry that represents a statement of any of the above types.

**Definition (1)**

An entry (E) is a record data structure composed of four fields :

1. Type field: to hold the type of statement such as while , or assignment .. etc.
2. In field : to hold the input set (I) for each statement (simple or compound).
3. Out field : to hold the output set (O) for each statement (simple or compound).
4. Tree field : to hold a pointer to the tree that represents the statement.

The entry of a simple statement is called a ***single entry*** , and the entry of a compound statement is called an ***interface entry*** through which the data flow information is passed between the body of the statement and the other statements in the program.

It is intended in this work to find the input-set (I) and output-set (O) for each statement of a Pascal program. Table (1) shows the Input and Output sets for the assignment statement and the condition part of a compound statement. Meanwhile Table (2) shows the input and output sets of the body of compound statements. However, the input and output sets of different types of statements in a Pascal program are introduced in Table (3). Notice that, we denote any current entry in Tables (1) and (3) by $E_0$.

Example 1:

```
While i <= n do
        begin
                i:= i+1 ;
                r:= n *b ;
                s:=r * 5 ;
        end;
```

By applying the rules of Tables (1) ,(2), and (3), the input set of the body of the while statement (I(E)), the input set of the condition (I(C)), and the output set of the while statement ($O(E_o)$) are given by :

$$
\begin{aligned}
I(C) &= \{i, n\} \\
I(E) &= \{i\} \cup \{\{\{n, b\} - \{i\}\} \cup \{\{r\} - \{i, r\}\}\} \\
&= \{i\} \cup \{n, b\} \cup \phi \\
&= \{i, n, b\} \\
O(E_0) &= \{i, r, s\}
\end{aligned}
$$

Therefore, the input set of the while statement is given by:

$$
\begin{aligned}
I(E_0) &= \{i, n\} \cup \{i, n, b\} \cup \{i, r, s\} \\
&= \{i, n, b, r, s\}
\end{aligned}
$$

Notice that, the inclusion of variable r in the input set $I(E_0)$, because of the rule of the while statement as suggested by Allan and Oldehoeft [5], will confuse the process of analysing the flow of data as it leads to accomplish an incorrect dependency relation between the while statement and the rest of statements in the program.

Now, if we modify Example 1 to be as follows :

```
begin
    x := 2 * t ;
    r := x * b + k ;
    while i <= n do
        begin
            i := i + 1 ;
            r := n * b ;
            s := r * 5
        end;
    :
end
```

**1- Consecutive Relation**

If a statement $i$ contains an output variable used by a statement $j$ such that $j \geq i+1$, which is represented formally as

$$O_i \cap I_j \neq \phi \ ,$$

then there is a consecutive relation between the statements $i$ and $j$.

**2- Prerequisite Relation**

This relation occurs between the statement $i$ and $j$ if statement $i$ does not contain output variables used by statement $j$ as input variables and vice versa. Also, there is no common output variables in statements $i$ and $j$ that can be used by statement $k$, where $k > j$.

That is,

$$O_i \cap I_j = \phi,$$
$$I_i \cap O_j = \phi,$$
$$O_i \cap O_j \cap I_k = \phi.$$

**4- Conservative Relation**

It means that the statements $i$ and $j$ contain output variables used by a statement $k$, where $k > j$. Formally,

$$O_i \cap O_j \cap I_k \neq \phi$$

## The Proposed Automatic Detector of Parallelism

Our proposed automatic detector of parallelism has been implemented in two stages : Data analysis stage, and the detection stage. These stages are explained below.

**A- Data Analysis Stage**

The process in this stage is divided into two parts. In the first part, a table of entries is built, whereas in the second part another table is built to maintain the definition / use of data variables. In what follows we describe these two parts.

*1- Building a Table for Entries*

In this part, the entries that correspond to the statements of the Source Program (SP) are constructed. First, a parse tree is built for each statement in SP by using the method of the recursive descent parser [8]. This is to ensure the syntactic correctness of the statement. After then, the tree is traversed to find the input and output variables of the statement in order to complete the contents of the entry that corresponds to this statement. This entry is added to a table called ***Entries Table***. Each entry will hold a

Input

| Type | Val | Use | Def | En_No |
|---|---|---|---|---|
| If | R | 1 | 3 | 0 |
| | | 2 | | |
| | | 4 | | |

*Figure (1) An Entry in the Definition-Use Table*

## B- The Detection Stage

In this stage the points in the program, in which parallelism is enhanced, are detected by finding the relations between the statements of the program. This process depends on the Definition-Use Table obtained from part two of the data analysis stage. The detection process, however, proceeds in a specific approach for each dependency relation, where such approach can lead to satisfying the case in which we can apply the condition of the dependency relation to recognize it.

In what follows, we present how the dependency relations can be recognized in this stage.

### *1- The Consecutive Relation*

To detect this relation, each variable in the Val field of an entry in the Definition-Use Table that corresponds to **output** variables is traced. That is to find whether this variable exists in the entries of the table that correspond to **input** variables. The tracing is started from the entry that has a serial number which is immediately greater than the serial number of the current one. Consider for example the following two entries : one corresponds to the output variable A, and the other corresponds to A as input variable.

Output

| Type | Val | Use | Def | En No |
|---|---|---|---|---|
| assn | A | 1 | 0 | 0 |
| | | 2 | 1 | |
| | | 3 | 4 | |
| | | 4 | | |
| | | 5 | | |

Input

| Type | Val | Use | Def | En No |
|---|---|---|---|---|
| If | A | 2 | 0 | 1 |
| | | 3 | 4 | |
| | | 4 | | |
| | | 5 | | |

In our work, we investigated the detection of dependency relations between the statements of the body of the While loop, in order to know how they can be executed in parallel if a parallel system is available. To illustrate this case, consider the piece of program in Example 2 below.

Example 2

```
T := 10 ;
R := T - 2 ;
While T <= R do
  begin
    S := T - 1 ;
    T := S / R ;
    K := T * R + 8
  end ;
```

The automatic detection of parallelism proceeds as follows:
First the Entries Table is constructed as shown in Table (5). In Table (5), we use T(3)(4) to indicate that the variable T of the condition part of the While statement is used in the entries numbered 3 and 4. However, entry 6 is excluded although it has the variable T in its input set and this is because the statement of entry 6 uses the value obtained from the output variable T of entry 5. The tree field which contains a pointer to the tree of each statement is not shows in Table (5) for simplicity.

*Table (5) Entries Table of Example 2*

| Type | in | out | Tree |
|---|---|---|---|
| 0. assn | null | T | |
| 1. assn | T | R | |
| 2. while | T(3)(4)<br>R | S<br>T<br>K | |
| 3. cond | T<br>R | null | |
| 4. assn | T | S | |
| 5. assn | S<br>R | T | |
| 6. assn | T<br>R | K | |

The Definition-Use Table is constructed afterwards as illustrated below:

*4- The Conservative Relation*

This relation is recognised if the variable in the Val field of an entry that corresponds to output variables is matched with the variable in the Val field of the other entries in the table where this variable is matched with the input variables of the consequent entries. For instance, if entries 1 and 2 have the variable b as an output variable, and entry 4 has b as an input variable, then by applying the condition of conservative relation :

$$O(E_i) \cap O(E_{i+1}) \cap I(E_j) \neq \phi, \text{ for } j > i+1$$

then we obtain the conservative relation between entries 1 and 2.

## Conclusion

Detecting parallelism in sequential programs requires a careful representative form and treatment. Using the table entries to represent the program, enabled us to implement the data dependency analysis through the whole program in a flexible and accurate manner. However, representing each statement by an entry in the table helped us to exclude any restriction that can be seen if the statements were represented by a construct such as the stanza which has the limitation of the size ( number of variables in one stanza is not more that 15 ).

In spite of the difference in these two representation methods, the concept of detecting the dependency relations between statements was the same with the use of the modification that relates to the data flow analysis which we implement. Therefore, we can conclude the followings: The flow of data is traced through the whole program by using the definition and use of variables in the input and output sets of each entry in the table. Meanwhile, the recursion implementation of the input and output sets has enabled us to include a compound statement or a mixture of simple and compound statements within a compound statement. For instance, If statement is included within another If statement or While statement.

Our approach is capable to analyse the dependency relations between any two statements, not just adjacent statements as was the case in [1]. However, the analysis also was within a compound statement to find the dependency relation between the iterations of the While loop for example.

In this work, we concentrated only on how to detect parallelism in sequential programs but not on how to enhance such parallelism which requires an allocation scheme of processors to the statements that can run in parallel. Besides, running the While loop in parallel needs a way to find synchronisation points between its iterations according to their detected dependency relations. Such cases require some extensive future investigations.

On the other hand, this paper does not concern the local and global variables, and procedures or functions. The investigation of such cases requires a careful study to the execution's environment before and after calling the procedure ( may be recursive ). This, obviously, requires new rules for constructing the input and output sets for procedures calls. This is the topic of our next research paper which is in preparation.