

# Associative Algorithms in Pascal/A

S SANYAL

Computer Systems and Communications Group,  
Tata Institute of Fundamental Research, Bombay 400 005, India

AND

N KRISHNAKUMAR

Department of Computer Science and Engineering, Indian Institute of Technology,  
Madras 600 036, India

**This paper deals with how associative algorithms can be easily formulated and expressed in a flexible associative language, Pascal/A, which is an extension of standard Pascal. The power of associative processing is delineated with the help of a few examples. Further, the simulation of an associative machine using Pascal/A is also discussed, with a broad overview of the issues involved in the efficiency of such an implementation.**

**P**RESENT day computers are limited to location-addressable memories. Here the access philosophy is the selection of memory words by indexes in a storage space viewed as a one-dimensional array of these words. A significant departure from this concept but moving closer to the human memory mechanism is the idea of associative memories or content-addressable memories (CAMs)[1,2]. In this case, the stored data items can be retrieved using their content or part of their content.

It is readily obvious that CAMs allow us to work with actual values as in mathematics, rather than with the locations of those values. The better performance of CAMs is due to the fact that searches for data items can be done in parallel.

The application of associative concepts to database management systems will result in a significant speedup in query processing. Another point of note is that associative algorithms to various problems can be easily formulated, as shall be seen later, making use of database concepts. However, the data

manipulation languages (DMLs) that are in existence are somewhat removed from the general structure of higher level languages (HLLs) and are devoid of the control structures that the latter provide. Attempts to remove this drawback and to make the manipulation of databases available from HLLs, while incorporating parallelism, have resulted in the development of associative extensions to these HLLs. This parallelism will, of course, have to be reflected in the hardware of the CAMs and the appropriate mapping from software to hardware will have to be effectively done for good performance.

It should be mentioned at this stage that much headway has not been made until now in terms of associative memories due to their prohibitive costs with semiconductor technology. However, with recent research discoveries such as the usage of holographic techniques and optical phase conjugation [3] for implementing CAMs, interest has been rekindled in this area. The relevance of associative processing features in a HLL background is now established so that a marriage of the power of content-addressability and the HLLs may be done.

Attention is drawn in this paper to an extended version of standard Pascal, Pascal/A [4],

which provides flexible associative programming language features. The discussion is carried into the ease of implementing associative algorithms in Pascal/A and how a testbed for algorithms, simulating the content addressability, may be constructed. The various types of implementations possible are then relatively evaluated.

### PASCAL/A AND ITS FEATURES

Essentially, this language provides features that may be used to devise new algorithms and techniques that exploit content addressability and the latent parallelism in problems. An overview of the constructs of Pascal/A is given below.

#### The table

The table is the only data structure adopted for the extension. This is a dynamic collection of an arbitrary number of identically structured Pascal records, each having the values of the attributes of the table. Thus a table is nothing but a relation, however without the set property of unique elements. This variable size data definition feature is a significant step ahead with respect to both LEAP [5] and PFOR [6], two other prominent associative languages. PFOR provides only static parallel variables whereas in LEAP, the entries are constructs named 'associations' which are restricted to ordered triples of (attribute, object, value). How the table facilitates the implementation of associative algorithms is seen later. Parallel operations on the different rows of a table are possible through generic instructions.

The syntax for the declaration of a table in BNF is

```
<table_type> ::= table of <type>
<type> ::= <standard_Pascal_type> |
<table_type>
```

Therefore a typical table declaration for a student file would look like

```
type student_typ = table of record
name : packed array [1..20] of char;
age : integer;
grade : real;
end;
```

```
var student : student_typ;
```

Name, age and grade are the attributes of the table student.

Similar to files, tables may also exist outside the program and this bolsters the concept of the manipulation of external databases. Such external tables will have to appear as a program parameter in the program heading as in

```
program grading (input, output, student);
```

#### Interfacing associative data structures

Every declaration of a table variable implicitly defines a buffer for it and rows can be transferred into and out of the table through the buffer. The I/O semantics of table variables is thus similar to that of sequential files.

In general, the definition of a statement in Pascal expands now, so that we will take the precaution of expressing it in BNF—

```
<statement> ::= <standard_Pascal_
stmt> | <extension_stmt>
<extension_stmt> ::= <insert_stmt> |
<retrieve_stmt> |
<readout_stmt> | <delete_stmt> |
<qualif_stmt> | <where_stmt> |
<foreach_stmt>
<table_id> ::= <identifier>
<field_id> ::= <identifier>.
```

Four primary functions allow query or manipulation of the table—

- (i) *Insert*: Transfers the content of <table\_id>^(the buffer for the table <table\_id> as in file notation) into the table, the buffer being unmodified by the operation.

```
<insert_stmt> ::= insert(<table—id>);
```

- (ii) *Retrieve* : Reads a specified row non-destructively from the table. Appropriate semantics may be defined in case multiple matches for the specification occur.

```

<retrieve_stmt> ::= retrieve
(<selection>);
<selection> ::= <table_id> [ <extension_bool_expr> ]
<extension_bool_expr> ::= <e_b_expr>
|
<e_b_expr> <logical_op> <extension_bool_expr>
| not <extension_bool_expr>
<e_b_expr> ::= <field_id> <rel_op> <identifier>
<logical_op> ::= and | or
<rel_op> ::= = | < | <= | > | >=

```

- (iii) *Readout* : Reads the data and also deletes the row from which reading has taken place.

```

<readout_stmt> ::= readout
(<selection>);

```

In both retrieve and readout, the reading takes place into the buffer.

- (iv) *Delete* : deletes specified rows from the table, ie delete (student [grade=0.0]) will delete the records of all the students with grade=0.0.

```

<delete_stmt> ::= delete (<selection>);

```

### Generic instructions for associative processing

Here a content-addressing feature selects the rows which may be manipulated with these instructions. The influence of PFOR is evident in these control structures. Basically there are three ways of addressing by content—

- (i) *Qualified statement*

```

<qualif_stmt> ::= <selection>_1.<field_id>
id> ::= <selection>_1.<field_id>
<operator> <expr>;

```

Here one may give a qualification expression for selection of rows in the table, as for example—

```

student [grade < 4.0]. grade :=
student [grade < 4.0]. grade + 0.05;

```

The qualification expression on either side of the assignment should be the same.

- (ii) *Where statement*

```

<where_stmt> ::= where <selection> do
<statement> [otherwise <statement>]

```

This combines the effects of a standard with statement and the content addressing feature—

```

where student [grade < 4.0] do
grade := grade + 0.05;

```

It is noteworthy that during the execution of the where statement, attributes that are part of the qualification expression can be changed. Relevant to this point of observation is the presence of an associative if-then-else construct—

```

where student [grade < 4.0]
do grade := grade + 0.05
otherwise grade := grade + 0.025;

```

- (iii) *Ordered selection statement*

```

foreach_stmt> ::= foreach <selection>
[<ordering>]
[<limitation>] do <statement>
<ordering> ::= (desc | asc) <field_id>
<limitation> ::= atmost <integer-expression>

```

The selection is a qualified table. The ordering determines the ascending or descending ordering on a particular field. The limitation is the number of iterations. During each iteration exactly one row is set to active and at the same time copied to the table buffer.

### Procedures and functions

Associative variables may be used as procedure or function parameters just as any other

type. However, tables will have to be passed as variable rather than value parameters to eschew excessive copying. Furthermore, functions may not be of type table.

The execution of parallel functions is possible by including the function call within a where statement or within a qualifying statement. Obviously, global variables may be accessed but not changed by parallel procedures or functions.

### Predefined functions

In addition to the interfacing functions, four more functions are defined.

- (i) The boolean function *match* determines whether the set of active rows (selected by a qualification) is empty and evaluates to false if empty, otherwise to true.
- (ii) The boolean function *empty* indicates whether the table variable has any rows at all.
- (iii) The integer function *counts* the number of active rows satisfying a qualification.
- (iv) *Size* returns the size of the table (the number of rows).

### APPLICATIONS AND EXAMPLES

The likely applications of associative programming languages are in problems of multi-key information retrieval, studies on the semantic associations of words, text searching and so on, where essentially we detect the requirement of quasi-simultaneous operations on large data sets. However, we can also utilise content-addressability to solve problems of a much more rigorous nature as in parallel compilation, numerical analysis and graph theory.

We will highlight the ease of formulating associative algorithms with a few examples and how we may express them in Pascal/A.

Consider the well known shortest path problem. Here we are required to find the shortest path between two vertices of a directed

graph. Various applications of this problem instance, such as in telephone networks or as in semantic networks of Artificial Intelligence, render this problem important. Numerous algorithms have been put forward to solve this in a sequential fashion but the elegance of the associative solution, along with its efficiency, is readily evident.

The solution to the shortest weighted path (length) problem is as follows. The idea is to traverse edges in the graph in parallel at the same speed. Thus starting at the source, we proceed along the edges emanating from it till we reach a node (the nearest to the source). At the other edges, we have still not reached a node so that the effective distance to cover now is the remaining length. We set the distance between the source and the other nodes as this length. Now we resume the traversal from two points, the source and the node that we had reached. This goes on till the specified destination is obtained and the path that leads to it in this fashion is the shortest path. Thus at any stage, we take one daughter source at a time and traverse its adjacent edges (coming from it) in parallel. The entire algorithm has been implemented in Pascal/A in the Appendix. The crux lies in the traversal of all the edges, in parallel, with a particular source and this is seen in content-addressability as the retrieval of all edges with source—a particular node. It should be noted that this extremely naive approach is efficient in the associative case.

Yet another example is in finding out whether an undirected graph is bipartite or not. A graph is bipartite if all its vertices can be divided into two disjoint sets, such that there is no edge between two members of the same set. We take an arbitrary vertex as the starting point and give it a label 0. Then all edges starting from this vertex are traversed and the vertices at the other end labelled with a 1. Starting from these vertices with a label 1, we traverse the adjacent edges (for one label-1 vertex at a time) and label the vertices at the other end with a 0. Thus we proceed alternately labelling vertices with 0s

and 1s. If at any stage, there is an already labelled vertex and it does not match the label which it would have got now, then the graph is not bipartite. The order of time complexity is linear in the number of vertices in the graph.

The absence of any complex algorithmic tricks in the above algorithms reveals the power and efficiency of an associative solution and enhances the need for easy expressibility in a flexible language such as Pascal/A. As seen in the Appendix, the presence of the named table as a dynamic structure imparts the flexibility to the language, for it directly corresponds to the concept of large sets of data with the same attributes being put under a superset. Database systems can thus be efficiently implemented using this facility.

### IMPLEMENTATION BY SIMULATION

Now that we have seen the relatively simple nature of associative algorithms, we would like to verify them as well. In the sequential environment, one would have to simulate the parallelism in the associativity. Considering the fact that the extension is minor compared to the extensive standard Pascal, it is better that a translator from Pascal/A to sequential Pascal be constructed, to avail of the full power of a good Pascal compiler. This translator would insert code for manipulation of an efficient data structure implementing the table.

It is true that the performance in storing, modifying and retrieving information depends on the representation in, and the topological structure of, the associative memory. In general, we should realise that if the problem is to find entries in a database by given identifiers, then hash coding is preferred. But if the searching conditions are specified by giving the values or ranges of certain attributes, then traditional sorting and tree searching methods may be used. The reader is referred to [7] for an excellent overview of such methods.

The concept of inverted file systems, where the roles of records and attributes are reversed,

is prominent. Here, we list the records with a given attribute, instead of the attributes of a given record. The implementation of LEAP [5] uses inverted list structures to store its triples of (attribute [A], object [O], value [V]). However, since a-triple must be retrievable on the basis of one, two or all three of the elements in the triple, the hash coding is somewhat complex and six hash tables are organized for the combinations of (A,O), (A,V), (O,V), A alone, O alone and V alone. In our case though, we are not restricted to such triples so that the complexity of our implementation increases manifold. It is to be noted that the efficiency of any of the methods used in multi-key retrieval is extremely dependent on the type of queries expected.

Here, we put forward a naive implementation of the preprocessor, used to do the translation, for the sake of completeness and to act as an introduction to the manner of implementation expected.

The preprocessor did a proper syntax-analysis of the extension constructs, as well as a partial semantic check. The vital part however is in the actual translation to standard Pascal. This was accomplished in three phases-

#### Processing of type declarations

All lines in the Pascal/A program with type declarations were held temporarily in a linked list of lines, before being written on to the new file (which serves as input to the standard Pascal compiler). This was done to keep track of the various type identifiers and their types, along with associated information, such as the field names of the record and the like. When a type declaration of table—as described earlier in the introduction to Pascal/A—was met with, the record was traced and a new record type created. This new record type had two more fields—a 'prev' pointer and a 'next' pointer. The simple implementation of the table as a doubly-linked list was adopted, hence the two extra fields. Thus the declaration of the student

table would now look like—

```

type
  student_typ=record
    name : packed array [1..20]
    char; of
    age : integer;
    grade : real;
  end;
  student_typptr= ^ student_typextn;
                    {pointer}
  student_typextn=record
    rec : student_type ;
    prev, next : student_typptr;
  end;

```

### Processing of variable declarations

Here the external tables were separated from the external files—note that both appear in the program heading—by looking up the symbol table for type declarations. All references in the variable declarations to type declarations of tables have to be appended with the suffix extn. The restriction is of course that a variable name starting with a table name must not be used to denote some other type, to avoid confusion. If there is a direct table declaration among the variable (var) declarations, the declaration is transferred to the type declarations and appropriate adjustments made.

Thus at the end of the first two phases,

- (i) All table declarations occur in the type declarations and variable declarations refer to these types.
- (ii) The external tables have been declared as files of the same name.
- (iii) A symbol table of external and internal table names has been constructed to aid in syntax and semantic checks on the extension statements.

### Processing of Pascal/A program block

In this phase, statements need to be introduced to read in the external tables (from files

of the same name), since the external tables have to be ready before the program starts executing. The associative constructs are then to be recognized, their syntax checked and corresponding sequential statements substituted for them. At this point, it is noteworthy that within constructs with a selection such as the 'where', the selection need not be repeated for each field name used. For the case of the implementation of such details, the Pascal 'with' statement comes in handy, for the entire block need only be enclosed within such a 'with' statement.

A point of consideration is the implementation of the ordering in the foreach statement. Here it was decided to use insertion sort since by implementation, the entire table would be sorted on that field and if another sort were required on the same field in the same ordering (quite often the case in loop programs), then it would be in best case complexity. Other schemes such as quicksort would have had more chances of worst case complexity table sorting. That some other schemes may be much better is entirely true, which poses another area of improvement in the overall sequential execution time.

Getting back to the actual translation, the external tables will have to be written back to the files of the same name at the end of execution and statements are to be introduced to accomplish this.

The preprocessor above has been implemented on an IBM-PC in the C language in around 2000 lines of code. The scope for expansion and attaining better access times with different data structures is currently being researched.

### CONCLUSION

The attempt, in this paper, has been to discuss associative algorithms and the language Pascal/A in an integrated environment. The examples have been given with an idea to show the elegance and death of complexity in

associative algorithms, and to give their representation in Pascal/A. The translator to sequential Pascal is seen as a useful tool to verify such algorithms as may be formulated. Thus we have seen that Pascal/A provides a means to write elegant associative programs to solve problems for which sequential solutions are highly complex, and the scope for development of many more algorithms is very large.

#### REFERENCES

1. C C Foster, *Content Addressable Parallel Processors*, Van Nostrand Reinhold Company, 1976.
2. T Kohonen, *Content Addressable Memories*, Springer Verlag, 1980.
3. D M Pepper, Application of optical phase conjugation, *Scientific American*, vol 253, Jan 1986.
4. H J Stuttgen, *A Hierarchical Associative Processing System. Lecture Notes in Computer Science*, vol 195, Springer Verlag, 1985.
5. J A Feldman & P D Rovener, An algol-based associative language, *Communications of the ACM*, vol 12, pp 439-449, Aug 1969.
6. D E Wilson, PEPE Support Software, *IEEE Comp-Con*, pp 61-64, 1972.
7. D E, Knuth, *The Art of computer Programming-* (Vol 3), *Sorting and Searching*, Addition Wesley nc., 1973.

#### APPENDIX

program weighted-path (input, output, nodefile);

{This program finds the shortest weighted path between two vertices in a digraph.}

const  
 n=100;  
 plusinf=10000;

type  
 node=1..n;  
 steps=0..n;  
 edge=record  
 src: node; {Start pt. of the edge.}  
 dest: node; {End pt. of the edge.}  
 step: steps;  
 length: integer; {Length of edge.}  
end;

```

var
  graph : table of edge;
  nodefile : file of node;
  source : node; {Source node of path.}
  destination : node; {Destination node of path.}
  i : steps;
  j : integer;
  min : integer; {Minimum length edge at each step.}
  current : steps; {Current step indicator}
  progress : boolean; {Search still progressing}
  found : boolean {Search successful?}

begin
  {Initialize everything.}
  current := 0;
  found := false;
  progress := true;

  {Read in the source and destination nodes.}
  writeln (' Input the source and destination nodes. ');
  readln (source, destination);

  {Input of graph edges}
  writeln (' Input the number of edges. ');
  readln (i);

  writeln (' Input the start',
    ' and end points of the edges and the lengths. ');
  graph^.step := current; {Initialise step of each edge to 0.}
  for j := 1 to i do
  begin
    readln (graph ^. src, graph ^. dest, graph ^. length);
    insert (graph); {Construct the table.}
  end;

  {Prepare first step.}
  current := 1;

  {Mark all edges originating in source in parallel.}
  where graph [src=source] do step := current;

  {Now do the searching.}
  while (progress) and [not found] do
  begin
    {Store all nodes reached in the last step through
    edge of minimum length and nodes which have been
    reached until now with untraversed edges. }

    min := plusinf;
    foreach graph [step = current] do
      if (graph ^. length < min)
      then min := graph ^. length; {Find minimum
      edge length.}
  end;
end;

```

```

rewrite (nodefile);
foreach graph [step = current] do
  begin
  if (graph ^. length > min)
  then
    begin
      nodefile ^ := graph ^. src;
      graph ^. step := 0; {The edge is still to be
        traversed.}
    end
  else
    nodefile ^ := graph ^. dest;
    graph ^. length := graph ^. length - min; {Update
    length.} put (nodefile);
  end;
  if (match) graph [ (step = current) and
    (dest = destination)]
  then found := true;
  if not found
  then
    begin
      current := current + 1; {for next step}
      reset (nodefile);
      while not eof (nodefile) do
        begin
          {All edges originating in the same node
          are}
          {marked in parallel in O(1) time. }
          where graph [ (step = 0 ) and
            (src = nodefile) ^. ] do
            step := current;
            if not eof (nodefile) then get (nodefile);
          end;
          if not (match (graph [step = current])
            then progress := false;
            {No edge marked on previous step.}
          end
        end; {while}
      {Output section}
      if found
      then
        begin
          writeln ( ' The path in reverse is as follows:');
          writeln;
          write (destination);
          i := current;
          while (destination <> source) do
            begin
              retrieve (graph [ [(step = i) and
                (dest = destination) ] );
              write (graph ^. src);
              destination := graph ^. src;
              i := i - 1;
            end;
          end
        else writeln ( 'There is no such path in this graph. ');
      end. {main}
    end
  end
end.

```

---