Bull. Nov. Comp. Center, Comp. Science, 1 (1993)

High level language STAR for associative parallel processors and its application to relational algebra

A.Sh. Nepomniaschaya

This paper describes implementation algorithms of relational algebra operations in associative parallel processors like Staran using the language STAR. For these operations we construct procedures which form a hierarchy.

1. Introduction

Fine-grain SIMD-architectures with bit-serial (vertical) processing and simplest processor elements (PEs) have been the subject of the intensive research in the recent years. Such machines allow to receive super performance due to simultaneous run of a large number of PEs [16]. We can cite Staran, Aspro, Lucas, MPP, DAP and CM which belong to the fine-grain SIMD-architecture. Most high level languages for such machines are directed towards a specific machine and take into account its architecture. The following high level languages were proposed: the language ASP for machines Staran, CM and Aspro [15], Parallel Pascal for MPP [17], DAP Fortran for DAP [8], Pascal/L for Lucas [4], *Lisp an extention of Common Lisp for CM [7].

In this paper we will consider associative parallel processors (APPs) like Staran [1] which can access data by contents and perform search operations in parallel. For such processors the assembly language Apple was proposed in [10]. To write different algorithms oriented to the mentioned processors, the language APL and some special meta-languages were considered in [3,6,11,14].

Recently some specialized parallel processors have been developed to accelerate the realization of certain non-numeric procedures including relational algebra operations [5,12,14]. Therefore it would be useful to have a high level language allowing to model the run of such processors, to define their acceleration and write down algorithms for associative architectures.

To this end high level language STAR was proposed in [13]. This language allows to model the complete process of associative processing. That is why it may be used both for the run simulation of new associative architectures and for the writing of algorithms oriented to APPs like Staran.

The main target of this paper is to write down the internal algorithms for relational algebra operations and to define their complexity using the language STAR. Relational algebra operations were selected due to the following reasons: (1) the hardware of APP is well suited for such operations [2,4]; (2) some algorithms of this paper can be used to compare them with similar algorithms written for specialized processors.

Note that representation of relational algebra operations on Lucas was considered in [4]. These operations have been implemented by microprograms. Using the language STAR we can write down the corresponding algorithms for such microprograms in an evident form and investigate them.

2. Model of the associative parallel machine

We consider the associative machine Staran as the base architecture. To describe our model we will use an abstract STAR-machine of SIMD-type with vertical data processing. It consists of the following components:

(1) a sequential common control unit (CU) where the programs and scalar constants are stored;

(2) k associative processors ($k \leq 32$), each consisting from m single-digit processor elemets ($m = 256$);

(3) k matrix memory modules where the i-th module is connected with the i-th processor ($1 \leq i \leq k$).

Each memory module consists of r blocks ($r \leq 16$) and each block consists of m words by m bits. In any block the rows are numbered from top to bottom and the columns from left to right. A row (word) or a column (slice) may be accessed equally easy. The data are viewed as a two-dimentional array written in binary code. Each array element occupies an individual row and all elements have the same length. The data array is divided into parts each of m rows. They are loaded into the module blocks so that each part is stored in a block and different parts are stored in different modules. In the CU a rendition table should be located allowing to associate with each array identifier the number of columns and parts in the partitioning.

Each associative processor can be represented as h vertical registers each of m bits. The bit columns of the data array are stored in the registers which perform the necessary Boolean operations, record search results and

ensure the word selection capability. The STAR-machine processor has a sufficient number of vertical registers ($h \geq 3$) to store intermediate results of data processing without using the module memory.

In this paper we consider the STAR-machine with one associative processor ($k = 1$). In this case all parts of the data array are loaded into module. The CU decodes program instructions and causes the processor to execute them. The processor performs vertical data processing for each block in turn.

3. Review of the language STAR

We will consider only those STAR constructions [13] which are necessary for the description of relational algebra operations. To simulate data processing in a module block the following data types are used: *integer*, *boolean*, *word*, *slice*, *table* and *array*. Data type will be introduced as in Pascal. Constants for the types *slice* and *word* are represented as an ordered sequence of symbols 'zero' and 'one' enclosed within single quotation marks (apostrophes). Note that the types *slice* and *word* are introduced for bit column access and bit row access respectively.

Let M be a variable of the type *array*. Then M is a structure of a fixed number of components, which all are of the same type *integer*. Let T be a variable of the type *table*. Then T is associated with the matrix T of k columns where $k \leq 256$.

By analogy with [4] for each matrix T there is a unique bit-slice (called workfield) TWF indicating by $'1'$ those rows which belong to T.

3.1. Operations, predicates and functions for slices

Any variable of the type *slice* consists of 256 components which belong to $\{0, 1\}$.

Let Y be a variable of the type *slice*, i be a variable of the type *integer*. Consider that ONE-component and ZERO-component of Y denote a component of Y with the value $'1'$ and $'0'$, respectively.

Define the following operations:

CLR(Y) sets all components of Y to ZERO;

SET(Y) sets all components of Y to ONE;

$Y(i)$ selects the i-th component of Y;

NUMB(Y) yields the number i of all ONE-components of $Y, i \geq 0$;

FND(Y) yields the ordinal number i of the first ONE-component of $Y, i \geq 0$;

STEP(Y) yields the same result as the operation FND(Y) and then resets the first ONE-component of Y. If the slice Y has no ONE-components it does not change.

Let X and Y be variables of the type *slice*. The following logical operations are executed simultaneously by all corresponding components of X and Y and are introduced in the obvious way:

$X \wedge Y$ is conjuction, $X \vee Y$ is disjunction, $\neg X$ is negation. Other logical operations are constructed from these ones by means of superposition.

Let us agree that $X \oplus Y$ denotes exclusive 'OR'.

There are the following three predicates for slices:

ONE(Y) yields *true* if and only if the slice Y consists completely of ONE-components;

ZERO(Y) yields *true* if and only if the slice Y consists completely of ZERO-components;

SOME(Y) yields *true* if and only if the slice Y has at least one component with value '1'.

We will use the following function Shift(Y, k) in which k is a variable of the type *integer*. This function moves the contents of Y placing the component from position N to position $N + k$ $(N \geq 1)$ and setting ZERO-components from the first through the k-th positions, inclusive. If $k = 0$ then the contents of Y does not change.

3.2. Operations and standard functions for words and matrices

Let w be a variable of the type *word*, i be a variable of the type *integer*. We will use the following two operations:

$\#w$ yields the length of w $(\#w \leq 256)$;

$w(i)$ yields the i-th component of w.

Let T be a variable of the type *table*, and i, j, k be variables of the type *integer*. We will use the following matrix operations :

$T(i)$ yields the i-th row in the matrix T;

Col(j, T) yields the j-th column in the matrix T;

$T[k]$ yields the k-th part of the matrix $T(1 \leq k \leq 16)$. This operation is used in the case when the matrix T has more than 256 rows.

The function Size(T) yields the number of columns in T.

The function Row(T) yields the cardinality (number of rows) of T.

Remark. It should be noted that statements of STAR resemble those of Pascal [9].

4. Algorithms for relational algebra operations

A relational database is defined as in [18]. Let D_i be a domain, $i = 1,2,\ldots,k$. The relation R is considered as a subset of the Cartesian product $D_1 \times D_2 \times \ldots \times D_k$. An element of R is called *tuple* and has the form $v = (v_1, v_2, \ldots, v_k)$, where $v_i \in D_i$. Let A_i be a name of the domain D_i which is called the *attribute*. Let $R(A_1, A_2, ..., A_k)$ denote a *scheme* of the relation R.

Any relation is represented as a matrix (table) in the memory module and each its tuple is allocated to one memory word. Therefore the values of attributes occupy the vertical fields in the matrix. Note that any relation consists of different tuples. If the relation R consists of several parts, then each part of R has its own workfield bit-slice RWF in the module.

Relational algebra operations can be divided into two groups:

(1) operations for which the result relation is a subset of one of the argument relations;

(2) operations which assemble a new relation which should be located in a new area of the memory module.

We introduce the following two notations being used for algorithm analysis. Let A be an algorithm applied to the matrix T. Denote by $N(A)$ the access number to the parallel memory during the execution of A and by k the row length of matrix T. Following [9], we assume that definition of existence of responder in the parallel memory needs no additional time.

We define two auxiliary functions to be used later. Let *drow(T)* be a function which counts the number of different rows in T. For a variable X of the type *slice* let *pos(X)* be a function which yields an array (vector) consisting of position numbers of ONE-components of X.

4.1. Relational algebra operations having bit-slice as a result

In this section we will consider the following relational algebra operations belonging to the first group: Intersection, Difference, Selection, Semi-join, Projection and Division. The resulting relation of any operation from this group is a subset of one of its argument relations. Therefore we will use a bit-slice to indicate the resulting relation tuples.

Notice that the basic function used by implementing the procedures of this group is search. That is why, we consider first of all the procedure MATCH which tests a word v for the membership in the relation D.

```
proc MATCH(D:table; DWF: slice; v: word; var M: slice );
label 1; var i,k: integer; X: slice;
begin k:= size(D); M:=DWF;
```

```
    for i:= 1 to k do
      begin X:= col(i, D); if v(i) =' 1' then M := M ∧ X
        else M := M ∧ ¬X; if ZERO(M) then goto 1
      end;
1:end
```

A detailed explanation may be appropriate here. Note that the relation D and its bit-slice DWF are loaded in parallel memory. Since the vertical processing is executed in the associative processor it is necessary to have at least two variables of the type *slice*. The variable X is used to store the current column of D which is operated on and the variable M is used as the resulting bit-slice. At the start M has the same contents as DWF since the seach will be executed among those rows of D which correspond to the positions with '1' in the slice DWF. Let us call such rows of D *selected* rows. At any j-th step of the algorithm ($1 \leq j \leq \#v$) the positions of those selected rows (tuples) of D will be marked by '1' in the slice M which have the first j symbols of v as their initial part. This algorithm terminates earlier if there exists such a step $h < k$ in which the slice M has only ZERO-components.

Thus the procedure MATCH yields the slice M in which $M(i) =' 1'$ if and only if $D(i)=v$. For this procedure we obtain $3 \leq N(MATCH) \leq 2k+1$, where the lower bound is reached when the relation D has no tuples (rows) beginning with the symbol $v(1)$.

The operation Intersection has two argument relations. The resulting relation consists of those tuples which belong to both argument relations. Consider this operation.

```
proc INTERS (T, R:table; TWF,RWF:slice; var Z:slice);
var i:integer; w:word; X, Y:slice;
begin CLR(Z); X:=TWF;
  while SOME(X) do
    begin i:=STEP(X); w := T(i); MATCH (R,RWF,w,Y);
      if SOME(Y) then Z(i) :=' 1'
    end;
end
```

Let us explain the procedure INTERS. At the start the resulting slice Z consists of ZERO-components and the slice X has the same contents as TWF. For each $i \in pos(X)$ we mark the tuple position $T(i)$ belonging to R by means of ONE-component in the slice Z.

For this procedure we have $N(INTERS) \leq 2+2\cdot(k+2)\cdot$row(T), where size(T)=size(R)=k. The upper bound will be smaller if row(T)$\leq$row(R).

Consider the operation Difference of relations T and R. The resulting relation consists of those tuples of T which do not belong to R.

The procedure header has the following form:

proc DIFFER (T, R:*table*; TWF,RWF:*slice*; *var* Z:*slice*);

Its body is obtained from the INTERS body by means of replacing the predicate SOME(Y) by ZERO(Y).

Let T be a relation having two attributes $T1, T2$ and R be a relation having one attribute. Let $T2$ and R be drawn from the same domain.

Consider the operation Semi-join of relations $T(T1, T2)$ and R. Its resulting relation consists of those tuples $T(i)$ for which there exists such j that $T2(i) = R(j)$. In the slice Z we mark positions of the result tuples.

The procedure header has the following form:

proc SJOIN($T(T1, T2)$, R:*table*; TWF,RWF:*slice*; *var* Z:*slice*);

Its body is obtained from the INTERS body by means of replacing the statement $w := T(i)$ by $w := T2(i)$.

Consider the operation Selection which tests a word w for the occurrence in the matrix T as its row substring.

```
proc SELECT(T:table; TWF:slice; j:integer; w:word; var Z:slice);
label 1; var i,k,m,n:integer; X:slice;
begin k := #w; n := k + j − 1; m := 0; Z:=TWF;
 for i := j to n do
  begin m := m + 1; X:=col(i,T); if w(m) =' 1'
   then Z := Z ∧ X else Z := Z ∧ ¬X;
    if ZERO(Z) then goto 1
  end;
1:end
```

Note that this procedure generalizes the procedure MATCH which is obtained when $j = 1$ and $\#w$=size(T).

Consider the operation Projection2 of the relation T having two attributes $T1$ and $T2$. Note that in general case $T1$ or $T2$ is not a relation.

```
proc PROJECT2(T(T1,T2): table; TWF: slice; var Z: slice);
var i:integer; X,Y: slice; w: word;
begin X:=TWF; CLR(Z);
 while SOME(X) do
  begin i:=FND(X); w := T2(i); Z(i) :=' 1';
   MATCH(T2,TWF,w,Y); X := X ⊕ Y
* Delete the tuples T(i) including the occurrence of w*
  end;
end
```

The procedure PROJECT2 selects the occurrence of the component $T2(i)$ in the current tuple $T(i)$ of the relation T. In the slice Z we mark the position of this tuple. From the further consideration we except all tuples of T, which include the occurrence of the component $T2(i)$.

For this procedure we have

$$N(PROJECT2) \leq 2 + (5 + 2size(T2)) \cdot drow(T2).$$

Before we consider the Division operation recall its definition. Let T be a dividend having two attributes $T1, T2$ and R be a divisor. Assume that values of $T2$ and R are drawn from the same domain. Then

$$T \div R = \{u \in T1/\forall v \in R\, uv \in T\}.$$

Note that the implementation of the Division operation is complicated enough [18]. However for the considered model we can write the following clear algorithm.

```
proc DIV(T(T1,T2),R: table; TWF,RWF: slice; var B: slice );
var X,Y,Q,M: slice; k: integer; w: word;
begin X:=RWF; Y:=TWF;
  while SOME(X) do
    begin k:=STEP(X); w := R(k); MATCH(T2,Y,w,Q);
      INTERS(T1,T1,Q,Y,M); Y := M
    end;
    B := M
end
```

To explain the procedure DIV, assume that $R = \{v_1, v_2, \ldots, v_k\}$. This procedure constructs the following sequence of embedded sets.

$$E_1 = \{\alpha \in T1/\alpha v_1 \in T\}$$

$$E_2 = \{\beta \in E_1/\beta v_2 \in T\}$$

$$\ldots\ldots\ldots\ldots$$

$$E_k = \{\delta \in E_{k-1}/\delta v_k \in T\}$$

It can be easily seen that $E_k = T \div R$ by construction.

4.2. Relational algebra operations having a relation as the result

In this section, we consider the operations Union, Product and Join from the second group. The resulting relation of any operation from this group is assembled from the argument relations. We will assume that the resulting relation has no more than 256 rows.

Firstly we consider a group of auxiliary procedures which will be used later. Some procedures will be considered in detail, but the other ones only informally explained. Note that we borrow some auxiliary procedure names from [12].

Consider the procedure PUSH which shifts the contents of the matrix D on r positions down.

```
proc PUSH(D:table; r:integer; var: H:table );
var i,k:integer; X,Y:slice;
begin k:=size(D); for i:= 1 to k do
   begin X:=col(i,H);Y:=col(i,D);Y:=shift(Y,r);
      col(i,H) := X ∨ Y
   end;
end
```

Let us explain the procedure PUSH. It performs disjunction between the i-th column of H and the i-th column of D after its shift on r positions. The resulting matrix H is obtained by means of a cycle on the variable i.

Consider the procedure TCOPY which constructs k copies of the matrix D.

```
proc TCOPY (D:table; DWF:slice; k:integer; var H:table);
var h,i,j,r,p:integer; B,X,Y:slice;
begin p:=size(D); r:=row(D);
 for j := 1 to p do
  begin CLR(B); X:=col(j,D); B := B ∨ X;
   for i:=1 to k − 1 do
    begin h := r · i;Y:=shift(X,h); B := B ∨ Y
    end;
 *k copies of the j-th column of D are created in the slice D*
    col(j,H) := B
   end;
 end
```

Let us explain this procedure. It constructs k copies for each j-th column of D. The resulting matrix H is obtained with the help of the external cycle on j.

Now we consider the following simple procedures:

proc CLEAR (*var D*: *table*);

proc COMPACT (*D*: *table*; *Y*: *slice*: *var H*: *table*);

proc WCOPY (*w*: *word*; *k*, *j*: *integer*; *var D*: *table*);

The procedure CLEAR sets ZERO-components in each column of D.

The procedure COMPACT constructs the matrix H consisting of those rows $D(i)$ of D, for which the i-th component of the slice Y is $'1'$, i.e., $Y(i) =' 1'$.

The procedure WCOPY writes k copies of the word w beginning with the j-th through the $(j + k - 1)$-th rows of the matrix D.

Consider the operation Union. Note that its argument relations have the equal number of attributes.

```
proc UNION (T, R:table; TWF,RWF:slice; var P:table; Z:slice);
var X,Y: slice; i,j: integer; w: word;
begin Z:=TWF; X:=RWF; CLEAR(P); PUSH(T,0,P);
* Relation T is copied into the matrix P*
 while SOME(X) do
  begin i:=STEP(X); w := R(i); MATCH(T,TWF,w,Y);
* The occurrence of R(i) in T is marked in the slice Y*
   if ZERO(Y) then
* The case when R(i) does not enter the relation T*
     begin Y := ¬Z; j:=FND(Y); P(j) := R(i); Z(j) :=' 1'
     end;
  end;
end
```

Let us explain the procedure UNION. The resulting relation P is assembled from the first relation T and those tuples of the second relation R which do not exist in the first one.

The operation PRODUCT has two argument relations. Its result relation is obtained as the concatenation of all combinations of its argument relations. Consider this operation.

```
proc PRODUCT (T, R:table; TWF,RWF:slice; var P(P1, P2):table);
var i,p,r,s:integer; X,Y:slice; G:table;
begin X:=TWF; Y:=RWF; p := 0; r:=NUMB(X); s:=NUMB(Y);
 while SOME(X) do
  begin i:=STEP(X); p := p + 1; WCOPY(T(i), s, 1 + (p − 1) · s, P1)
  end;
*s copies of any tuple of T are created in P1*
 COMPACT(R, Y, G); TCOPY(G, r, P2);
```

```
*r copies of the matrix G are created in P2*
end
```

Explain the procedure PRODUCT. Let r be a cardinality of T, s be a cardinality of R and G be a matrix obtained by compaction of R. The procedure PRODUCT constructs s copies of any tuple of T in $P1$ and r copies of the matrix G in $P2$. Note that the resulting relation P consists of $r \cdot s$ tuples.

Let us recall the definition of the operation Join. Assume that the first argument relation A has the attributes $A1, A2$ and the second one B has the attributes $B1, B2$. Let $A2$ and $B2$ be drawn from the same domain. The operation Join concatenates those tuples from its argument relations for which the corresponding values of $A2$ and $B2$ are equal. Now consider the operation Join.

```
proc JOIN (A(A1,A2),B(B1,B2):table; AWF,BWF:slice ;
var C(C1,C2):table);
var p,m,n,t:integer; w:word; M,N,Q:slice; E1,E2:table;
begin M:=AWF; t := 0; CLEAR(C1); CLEAR(C2);
 while SOME(M) do
  begin p:=FND(M); w := A2(p); MATCH(A2,AWF,w,N); M := M ⊕ N;
* All occurrences of w are deleted from M*
   MATCH(B2,BWF,w,Q); if SOME(Q) then
    begin PRODUCT(A1,B1,N,Q,E1,E2); PUSH(E1,t,C1);
     PUSH(E2,t,C2); m:=NUMB(N);
      n:=NUMB(Q); t := t + m · n
    end;
  end;
end
```

Let w be a tuple value belonging both to the domain of $A2$ and to the domain of $B2$. The occurrence positions of w are stored both in the slice N for the attribute $A2$ and in the slice Q for the attribute $B2$. Then the selected tuples from the relations A and B are concatenated by the Product operation. The obtained matrices $E1$ and $E2$ are stored into the result matrix (relation) C using the auxiliary procedure PUSH.

5. Conclusion

We have considered the abstract associative STAR-machine with one parallel memory module. Its run is described by means of the high level language STAR having statements which are resembling to the Pascal ones. However,

the STAR statements are applied to data structures oriented to description of the associative parallel machine with vertical processing. Therefore STAR allows to model the run of such architecture, to write down the algorithms which it executes, and to simulate the run of new associative architectures.

The main target of the paper is to write down in an evident form the internal algorithms of the relational algebra operations. For some operations we estimate the complexity of corresponding algorithms. These estimates can be used in the further research to define the acceleration of specialized processors which intended for execution of the same relational operations.

The constructed procedures form a hierarchy. To consider it we will use the following well-known definition.

Let P be a procedure. We say that P belongs to the O-th level of the hierarchy if it does not use other procedures. The procedure P belongs to the i-th level ($i \geq 1$) if all procedures used by P have a level which does not exceed i-1 and there exists at least one procedure belonging to the (i-1)-th level.

According to this definition we obtain the following classification of procedures.

The O-th level consists of all auxiliary procedures and the procedure for the Selection operation.

The first level consists of the procedures for the following operations: Intersection, Difference, Semi-join, Product, Projection and Union.

The second level consists of the procedures for the operations Division and Join.

So, the simplest relational algebra operation is Selection and the most complicated operations are Division and Join.

Acknowledgements. I would like to thank professor Ya.I. Fet for discussion the paper and editorial help.

References

[1] K.E. Batcher, STARAN parallel processor system hardware, Proc. of AFIPS Conf., Vol.43, 1974.

[2] B.P. Berra, E. Oliver, The role of associative array processors in data base machine architecture, IEEE Computer, Vol.12, No.3, 1979.

[3] A.D. Falkoff, Algorithms for parallel-search memories, J. of the ACM, Vol.9, No.10, 1962.

[4] C. Fernstrom, J. Kruzela, B. Svensson, LUCAS associative array processor. Design, programming and application studies, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vol.216, 1986.

[5] Ya.I. Fet, Parallel Processors in Control Systems, Energoizdat, Moscow, 1981 (in Russian).

[6] C.C. Foster, Content Addressable Parallel Processors, Van Nostrand Reinhold Company, New York, 1976.

[7] W.D. Hillis, The Connection Machine, MIT Press, Cambridge, Mass., 1985.

[8] R.W. Hockney, C.R. Jesshope, Parallel Computers: Architecture, Programming and Algorithms. Adam Hilger Ltd, Bristol, 1981.

[9] K. Jensen, N. Wirth, PASCAL User Manual and Report, Springer-Verlag, Berlin, 1978.

[10] R.G. Lange, High level language for associative and parallel computation with Staran, Proc. of Intl. Conf. on Parallel Processing, 1976.

[11] J. Miklosko, R. Klette, M. Vajtersic, J. Vrto, Fast algorithms and their implementation on specialized parallel computers, Special Topics in Supercomputing, North-Holand, Vol.5, 1989.

[12] M.R. Muraszkiewicz, Cellular array architecture for relational database implementation, Future Generations Computer Systems, Vol.4, No.1, 1988.

[13] A.Sh. Nepomniaschaya, Language STAR for associative and parallel computation with vertical data processing, Proc. of the Intern. Conf. "Parallel Computing Technologies", Novosibirsk, 1991 (in Russian).

[14] E. Ozkarahan, Database Machines and Database Management, Prentice- Hall, Inc. 1986.

[15] J.L. Potter, An associative model of computation, 2 – *nd* Int. Conf. Super-Computing, San Francisco, *CA*, May, 4-7, 1987.

[16] J.L. Potter, W.C. Meilander, Array processor supercomputers, Proceedings of the IEEE, Vol.77, No.12, 1989.

[17] A.P. Reeves, Parallel Pascal and massively parallel processor, The Massively Parallel Processor, J.L.Potter, Ed. Cambridge, *MA*: MIT Press, 1985.

[18] J.D. Ullman, Principles of Database Systems, Computer Science Press, 1980.