STAR – A LANGUAGE FOR ASSOCIATIVE AND PARALLEL COMPUTATION WITH VERTICAL PROCESSING

A. Sh. Nepomnyashchaya

STAR is a language for coding and analysis of algorithms for associative parallel processors with vertical processing (STARAN type processors). A description of the language is followed by examples of STAR procedures that are used in a model implementation of the superhigh-level language PARIS.

Data-processing and logic problems play an important role in various applications. The associative parallel processor provides one of the architectures for the solution of such problems [1-3]. Most high-level languages for such processors (Glypnir, Pascal/L, Parallel Pascal) exploit their architecture and are designed for a particular type of associative processor. STAR is a language designed for associative parallel processors with vertical sequential bitwise processing (STARAN type processors).

Our model of a parallel associative computer consists of a shared control unit and up to 32 associative processor modules. The control unit decodes the instructions and submits them for concurrent execution by all processor modules. An associative processor module consists of one or several 256 × 256 blocks (each block contains 256 256-bit words). We assume in this paper that a module contains no more than 16 blocks. The memory consists of 256-bit words (which may be extended to 9 kbits). Besides the parallel memory, each associative processor module has three 256-bit processing registers. In each active module, vertical processing may be conducted only in one block. Modules communicate by exchanging data.

The parallel operation of the system is described below using Pascal/L tools [4].

STAR is a language for coding and analysis of algorithms designed for execution on a parallel associative computer. On the one hand, STAR includes a set of data types and operations that are sufficient for the description of associative vertical data processing algorithms; on the other hand, the operations, the functions, and the predicates used in STAR have been simplified to the maximum extent compared with the instruction set of an actual associative computer. Therefore, unlike other high-level languages for associative processors, STAR does not hide the features of these computers from the user. STAR is used in a model implementation of the superhigh-level language PARIS [5], with SETL as its kernel. Finally, note that STAR is useful for teaching programming on associative parallel processors with vertical processing.

THE STAR KERNEL

In this section, we give the STAR constructs used for describing the operation of a single processor module.

The set of types in STAR is determined by its orientation toward processors with sequential bitwise (vertical) processing. We describe the operation of a single processor module using the following types: integer, boolean, word, slice, array, table.

A data type in STAR is defined, as in Pascal, either directly with the description of the variable or by a named reference.

Constants of types integer, boolean, word, slice are integers, the logical constants true and false, and sequences of ones and zeros enclosed in quotes. Each variable of type table is a structure of k components of equal length of type slice; each variable of type array is a structure of a fixed number of components of type integer. A few remarks concerning the types table and slice. Let T be a variable of type table. To this variable we associate a matrix T of k columns and s rows. Vertical associa-
tive processing of data is performed in a 256 × 256 block. Therefore \( k < 256 \) and one block can process either the entire matrix \( T \) (if \( s < 256 \)) or only part of the matrix. If \( s < 256 \), the matrix is "padded" with 256 − \( s \) rows of length \( k \) filled with the symbol \( \omega \). Informally, the symbol \( \omega \) indicates that so far the corresponding position does not contain a record. Thus, prior to processing, the matrix \( T \) or any part of the matrix consists of 256 rows.

Variables of type \textit{slice} used in vertical data processing therefore consists of 256 components, each an element of the set \{0, 1, \omega\}.

Operations on Slices

STAR is constructed using the associative processor instructions described in [2-4, 6]. Let \( Y \) be a variable of type \textit{slice}. The following operations are defined:

- \( \text{CLR}(Y) \) writes all zeros into the slice \( Y \);
- \( \text{SET}(Y) \) writes all ones into the slice \( Y \);
- \( \text{MASK}(Y, i..j) \) writes ones into the slice \( Y \) from position \( i \) to \( j \) inclusive (\( i < j \)); all other positions in \( Y \) are zeros;
- \( \text{MASKI}(Y, k) \) writes a combination of \( 2^k \) zeros and ones into the slice \( Y \) (\( 1 \leq i \leq 7 \)); the constant \( k \) is a binary number consisting of seven 0s and one 1 and \( i \) is the position of the 1-bit in this representation of \( k \) (for example, \( \text{MASKI}(Y, '00000100') \) defines a combination of the form '00001111' in the slice \( Y \));
- \( Y(i) \) extracts the \( i \)-th component from the slice \( Y \) (\( 1 < i < 256 \));
- \( \text{STEP}(Y) \) substitutes 0 for the first 1 in the slice \( Y \) (clears 1 to 0); if there are no 1s in \( Y \), the slice remains unchanged by this operation;
- \( \text{NUMB}(Y) \) produces the number of 1s \( k \) in the slice \( Y \) (\( k \geq 0 \));
- \( \text{FND}(Y) \) gets the serial number \( k \) of the first 1 in the slice \( Y \) (\( k \geq 0 \)).

Let \( X \) and \( Y \) be variables of type \textit{slice}. We define the following operation:

- \( \text{PRESS}(X, Y) \) removes from the slice \( X \) the components that correspond to zeros in the slice \( Y \) and compresses the contents of \( X \); if there are no zeros in \( Y \), the contents of \( X \) is unchanged.

Let us elucidate the semantics of this operation. Suppose that the slice \( Y \) contains \( k \) zeros (\( k \geq 1 \)) and the slice \( X \) contains \( l \) symbols \( \omega \) (\( l \geq 0 \)). Then the operation \( \text{PRESS}(X, Y) \) produces a slice \( X \) in which the last \( k + l \) symbols are \( \omega \).

The following bitwise operations have obvious definitions: conjunction \( X \land Y \), disjunction \( X \lor Y \), and negation \( \neg X \). Other bitwise operations for the slices \( X \) and \( Y \) are formed from these three operations by superposition: implication \( X \Rightarrow Y \), equivalence \( X \Leftrightarrow Y \), \( \neg(X \land Y) = (\neg X) \lor (\neg Y) \), \( (X \lor Y) = X \land (\neg Y) \), \( \neg(X \Rightarrow Y) = (\neg X) \land (\neg Y) \), \( \neg(X \Leftrightarrow Y) = (\neg X) \lor (X \land \neg Y) \) (exclusive "or," which we also denote \( X \oplus Y \)), and also \( X \lor (\neg Y), (\neg X) \land Y \).

Predicates for Slices

We define three predicates.

The predicate \( \text{ONE}(Y) \) takes the value \text{true} if and only if there are no zeros in the slice \( Y \).

The predicate \( \text{ZERO}(Y) \) takes the value \text{true} if and only if there are no ones in the slice \( Y \).

The predicate \( \text{SOME}(Y) \) takes the value \text{true} if and only if the slice \( Y \) contains at least one 1.

Standard Functions for Slices

Let \( X \) be a variable of type \textit{slice}. We define the following functions:

- \( \text{Shift}(X, k) \) shifts the contents of the slice \( X \) \( k \) positions; as a result of this shift, each component of \( X \) is moved from position \( N \) to position \( N + k \), the first \( k \) components are cleared to zeros, and the components pushed out of the slice are lost;
- \( \text{ShiftI}(X, k, i) \) shifts the contents of the slice \( X \) \( k \) positions starting with position \( i \) (\( i > 1 \)); as a result of this shift, each component of \( X \) starting with position \( N \) (\( N \geq i + 1 \)) is moved to position \( N + k \); the components \( i + 1, i + 2, \ldots, i + k \) are cleared to zero and the components pushed out of the slice are lost.

Below we define four functions in which the argument \( X \) by assumption does not contain the components \( \omega \).