Associative Graph Processor and its Properties *

Anna Nepomniashaya
Institute of Computational Mathematics
and Mathematical Geophysics,
Pr. Lavrentieva 6, Novosibirsk-90, Russia
anep@ssd.sscc.ru

Zbigniew Kokosiński
Cracow University of Technology,
Faculty of Electrical and Computer Eng.
ul. Warszawska 24, 31-155 Kraków, Poland;
zk@pk.edu.pl

Abstract

In this paper a model of a versatile associative graph processor called AGP is proposed. The model can work both in bit-serial and in bit-parallel mode and enables simultaneous search for a set of comparands and selection of the search types. In addition it has some built-in operations designed for associative graph algorithms. The selected functions and basic procedures of this model are described and its possible architecture is discussed.

Keywords: associative parallel processor, bit-parallel processing, associative graph processing, multiple-search

1 Introduction

It is well known that associative (content addressable) parallel processors of the SIMD type are particularly well suited for performing fast parallel search operations in many search-intensive algorithms. Recent advances in associative processing include new applications in computational geometry, relational databases, and artificial intelligence [6], new selection of extreme search algorithms for fully parallel memories resulting from average-case analysis [15], multiassociative/hybrid search [4], an experimental implementation of multi-comparand multi-search associative processors and corresponding parallel algorithms for search problems from various complexity classes [7,8].

In fully parallel associative machines every cell has built-in single-bit comparator and therefore some searches (f.i. exact match, mismatch) can be performed faster than in bit-serial (vertical) machines where the word size is crucial. However, searches that require sequential (bit-serial) processing of words are asymptotically equivalent in both models. Significant speedup over sequential processors can be achieved in multi-comparand fully parallel processors, where the set of comparands is matched simultaneously against the data set [1].

Graph algorithms are specific class of algorithms that deserve particular interest of researchers and programmers [16]. Many graph algorithms were implemented on associative parallel processors. Such algorithms employ different graph representation forms given as two-dimensional tables.

In [2], a special case of Dijkstra’s algorithm for finding the shortest path between two vertices in unweighted undirected graphs and Floyd’s shortest path algorithm have been represented on the bit-serial word-parallel associative processor LUCAS.

In [5], simple algorithms for a group of graph problems (independent set, clique, chromatic number, graph isomorphism) have been proposed on the single comparand fully parallel associative model RAMRC. It is supported by dedicated generators of combinatorial objects for mask and pattern generation.

In [10–12], a natural straightforward implementation of a group of classical graph algorithms using simple and natural data structures has been proposed on the model of associative parallel systems with vertical processing (the STAR-machine). This group includes associative versions of the following algorithms: Warshall’s algorithm for finding transitive closure and Floyd’s shortest path algorithm, Dijkstra’s algorithm and the Bellman–Ford one for finding the single-source shortest paths.

However, all above-mentioned models of associative computation impose some bounds on the set of processor operations. Efficient implementation of associative algorithms for a new class of graph problems requires new operations and extensions to existing processor models.

In this paper, we introduce a new associative machine model called Associative Graph Processor (AGP) suitable for implementation of associative algorithms that require, in addition to search capabilities, a set of graph related operations. The model can perform bit-serial and fully parallel associative processing of matrices representing graphs that
are stored in memory as well as some basic set operations on matrices (sets of columns) like copy, addition, minimum, maximum and merging. We will describe these novel features in detail and discuss its possible hardware implementation.

2. Model of Associative Graph Processor

In this section, we propose a model of the SIMD type with simple single–bit PEs called Associative Graph Processor (AGP). It carries out both the bit–serial and the bit–parallel processing. To simulate the access data by contents, we use both the typical operations for associative systems first presented in Staran [3] and some new operations to perform bit–parallel processing.

The model consists of the following components:

- a sequential common control unit (CU), where programs and scalar constants are stored;
- an associative processing unit forming a two–dimensional array of single–bit PEs;
- a matrix memory for the associative processing unit.

The CU broadcasts each instruction to all PEs in unit time. All active PEs execute it simultaneously while inactive PEs do not perform it. Activation of a PE depends on the data employed.

Input binary data are loaded in the matrix memory in the form of two–dimensional tables, where each data item occupies an individual row and is updated by a dedicated row of PEs. In the matrix memory, the rows are numbered from top to bottom and the columns – from left to right. Both a row and a column can be easily accessed.

By analogy with other models of associative processing, we assume that input data are loaded in the matrix memory before updating them.

The associative processing unit is represented as a matrix of single–bit PEs that correspond to the matrix of input binary data. Each column in the matrix of PEs can be regarded as a vertical register that maintains the entire column of a table. Our model runs as follows. Bit columns of tabular data are stored in the registers which perform the necessary bitwise operations.

To simulate data processing in the matrix memory, we use data types slice and word for the bit column access and the bit row access, respectively, and the type table for defining and updating matrices. We assume that any variable of the type slice consists of \( n \) components. For simplicity, let us call “slice” any variable of the type slice.

For variables of the type slice, we employ the same operations as in the case of the STAR–machine along with a new operation FRST(Y).

The operation FRST(Y) saves the first (the uppermost) component \( \text{t} \) of \( Y \) and sets to \( \text{t} \) its other components.

For example, if \( Y = 0010110' \), then the operation FRST(Y) returns \( Y = 0010000' \).

For completeness, we recall some elementary operations for slices from [10] being used for representing graph algorithms.

\( \text{SET}(Y) \) sets all components of the slice \( Y \) to \( 1' \);
\( \text{CLR}(Y) \) sets all components of \( Y \) to \( 0' \);
\( \text{FND}(Y) \) returns the ordinal number \( i \) of the first (the uppermost) component \( 1' \) of \( Y \);
\( \text{STEP}(Y) \) returns the same result as \( \text{FND}(Y) \) and then resets the first \( 1' \) found to \( 0' \);
\( \text{NUM}(Y) \) returns the number of components \( 1' \) in the slice \( Y \).

In the usual way, we introduce predicates \( \text{ZERO}(Y) \) and \( \text{SOME}(Y) \) and the bitwise Boolean operations \( X \text{ or } Y \), \( X \text{ and } Y \), \( X \text{ xor } Y \).

The above–mentioned operations are also used for variables of the type word.

Let \( T \) be a variable of the type table. We use the following two operations:

\( \text{ROW}(i, T) \) returns the \( i \)-th row of the matrix \( T \);
\( \text{COL}(i, T) \) returns the \( i \)-th column of \( T \).

Moreover, we use two groups of new operations. One group of such operations is applied to a single matrix, while the other one is used for two matrices of the same size. All new operations are implemented in hardware.

Now, we present the first group of new operations.

The operation \( \text{SCOPY}(T, X, v) \) simultaneously writes the given slice \( X \) in those columns of the given matrix \( T \) which are marked by ones in the given comparand \( v \).

The operation \( \text{not}(A, v) \) simultaneously replaces the columns of the given matrix \( A \), marked by ones in the comparand \( v \), with their negation. It will be used as the right part of the assignment statement.

Remark 1. It should be noted that the above presented two operations use the column parallelism, while the next two operations of this group will use the row parallelism.

The operation \( \text{FRST}(A) \) simultaneously replaces each \( i \)-th row of the given matrix \( A \) with \( \text{FRST}(\text{ROW}(i, A)) \).

The operation \( \text{op}(A, B, v) \) simultaneously performs disjunction in every row of the given matrix \( A \) and saves the result in the slice \( Y \), that is, \( \forall i \ Y(i) = 0' \) if and only if \( \text{ROW}(i, A) \) consists of zeros.

Now, we determine the second group of new operations.

The operation \( \text{SMERGE}(A, B, v) \) simultaneously writes the columns of the given matrix \( B \), that are marked by ones in the comparand \( v \), in the corresponding columns of the result matrix \( A \). If the comparand \( v \) consists of ones, this operation copies the matrix \( B \) into the matrix \( A \).

The operation \( \text{op}(A, B, v) \), where \( \text{op} \in \{\text{or, and, xor}\} \), is simultaneously performed between those columns of the given matrices \( A \) and \( B \) that are marked by ones in the
3. A Group of Basic Procedures

In this section, we propose a bit-parallel implementation of a group of basic procedures on the AGP model. In particular, we employ these procedures to represent different algorithms on associative parallel systems.

The procedure WCOPY(w, X, F) writes the binary word w in the rows of the result matrix F that correspond to ones in the given slice X. Other rows of F will consist of zeros. On the AGP model, it is implemented as follows.

```
procedure WCOPY(w:word; X:slice;
var F:table);
var Y:slice;
v:word;
begin CLR(Y); v:= not w;
  SCOPY(F, X, w);
  SCOPY(F, Y, v);
end;
```

The procedure WCOPY runs as follows. The slice X is simultaneously written in the columns of F that correspond to ones in the given w, while the slice Y is simultaneously written in other columns of F.

The procedure WMERGE(w, X, F) writes the given string w in the rows of the given matrix F that correspond to ones in the given slice X. Other rows of F do not change.

```
procedure WMERGE(w:word; X:slice;
var F:table);
var A,B:table;
Y:slice;
v:word;
begin Y:= not X; SET(v);
  WCOPY(w, X, A);
  F:= or (A, B, v);
end;
```

The procedure WMERGE runs as follows. We first save the given string w in the matrix A rows marked by ‘1’ in the slice X. Then by means of the matrix B, we save the matrix F rows that correspond to ‘1’ in the slice Y (that is, negation of X). Finally, after performing $F := or(A, B, v)$, we obtain the result of this operation.

The procedure TMERGE(T,X,F) writes the matrix T rows, marked by ones in the given slice X, in the corresponding rows of the result matrix F. The rows of F that correspond to zeros in the slice X do not change.

```
procedure TMERGE(T:table; X:slice;
var F:table);
var A,B:table;
Y:slice;
v:word;
begi
  Y:= not X; SET(v);
  SCOPY(A, X, v);
  A:= and (A, T, v);
  SCOPY(B, Y, v);
  B:= and (B, F, v);
  F:= or (A, B, v);
end;
```

Let us explain the main idea of this procedure. In the matrix A, we simultaneously save the rows of given matrix $T$ that are marked by ones in the slice X. Then in the matrix B, we simultaneously save the rows of the matrix $F$ that correspond to zeros in the slice X. These rows of $F$ will not change. The result matrix $F$ is obtained after fulfilling the statement $F := or(A, B, v)$.

The procedure HIT(T, F, X, Y) defines positions of the corresponding identical rows in the given matrices $T$ and $F$ using the slice $X$. It returns the slice $Y$, where $Y(i) = '1'$ if and only if $ROW(i, T) = ROW(i, F)$ and $X(i) = '1'$.

```
procedure HIT(T,F: table; X: slice;
var Y: slice);
var A: table;
v: word;
begi
  Y:=X and ( not Y);
end;
```

This procedure runs as follows. First, we simultaneously compare the corresponding columns in the given matrices $T$ and $F$ and save the result in the matrix $A$. Then we simultaneously perform the disjunction of all rows of the matrix $A$ and save the result in the slice $Y$. To define the result slice $Y$, we take into account only the matrices $T$ and $F$ rows that are marked by ‘1’ in $X$.

The procedure MATCH(T, X, w, Y) defines positions of rows in the given matrix $T$ that coincide with the given pattern $w$. It returns the slice $Y$, where $Y(i) = '1'$ if and only if $ROW(i, T) = w$ and $X(i) = '1'$.
by means of a matrix analogy with SETMIN. It is implemented on the AGP, there is at most a unique bit/4.

The procedure SETMIN(T, F, X, Y) defines positions of the corresponding rows in the given matrices T and F, where ROW(i, T) < ROW(i, F). It returns the slice Y, where Y(i) = ‘1’ if and only if ROW(i, T) < ROW(i, F) and X(i) = ‘1’.

procedure SETMIN(T, F, X, Y);
var A: table;
begin

This procedure runs as follows. First, the given pattern w is written in the matrix A rows marked by ‘1’ in the given slice X. Then the procedure HIT is applied to matrices T and A.

The procedure SETMIN(T, F, X, Y) defines positions of the corresponding rows in the given matrices T and F, where ROW(i, T) < ROW(i, F). It returns the slice Y, where Y(i) = ‘1’ if and only if ROW(i, T) < ROW(i, F) and X(i) = ‘1’.

procedure MATCH(T: table; X: slice;
w: word; var Y: slice);
var A: table;
begin

The procedure MATCH(T, A, X, Y) is determined by analogy with SETMIN. It is implemented on the AGP model as SETMIN(F, T, X, Y).

Correctness of procedures WCOPY and SETMAX is evident. Correctness of other procedures is proved by contradiction.

Now, we evaluate time complexity of the presented procedures. On the AGP model, the basic procedures WCOPY, TMERGE, and HIT take O(1) time each because they use only the elementary operations of the associative processing. The procedures WMERGE, MATCH, SETMIN and SETMAX take O(1) time each because they use basic procedures that run in O(1) time.

On the STAR-machine, all these procedures take O(k) time each [9, 11], where k is the number of bit columns in the corresponding matrix.

It should be noted that other basic procedures for non-numerical processing can be also implemented on the AGP model. Therefore a group of associative parallel algorithms that employ such basic procedures can be implemented on this model. In particular, this group includes associative versions of Kruskal’s algorithm and the Prim–Dijkstra one for finding the minimal spanning tree [13], the associative parallel algorithm for finding a fundamental set of circuits [14], and associative parallel algorithms for dynamic edge update of minimum spanning trees [9].

4. An Implementation of the AGP Model

The Associative Graph Processor architecture that is shown in Fig.1 consists of the following components:

- Data Array (DA) – n × p binary data matrix;
- Comparand Array (CA) – m × p binary comparand matrix;
- Tag Array (TA) – n × m binary tag matrix for processing and storing current comparison results;
- Search Function Generator (SFG) for TA;
- DM, CM, SM1, SM2 – binary mask vectors selecting proper submatrices of DA, CA and TA;
- Bit Position Counter (BPC) for generating consecutive bit slices selected by SM1 for bit–serial processing;
- C2 – binary comparand vector for TA;
- C2 – binary tag vector for TA;
- Logical Tag Resolver (LTR) for computing binary functions SOME/NONE and ALL/NOT ALL for T2.

Data Array contains two input matrices A and B (in some cases also C matrix). This means that a single cell of DA contains 2D or 3D flip–flops. It is possible to perform four basic logical operations \{\sim, \lor, \land, \oplus\} on elements of A and B in selected rows and columns of DA.

The result can be loaded either to A or B (or C) and any of these matrices can be then selected and processed as
Figure 1. An architecture of the AGP model

In the Tag Array, each cell TA[i,j] is a programmable processing element that performs comparison of subsequent pairs of bits \{DA[i,l], CA[j,l]\}, where l is the coordinate of processed bit slice, according to a search function loaded into TA. The number of basic logic searches generated by SFG is 6, but in general this set can be extended to 16 according to a given application [1].

In the AGP model, data processing is organized in bit–serial word–parallel mode. The typical single–comparand register is replaced by a Comparand Array, and the tag memory is extended to the size of the Carthesian product of data and comparand sets. Associative processing is performed exclusively in two–dimentional Tag Array. In each consecutive step of processing states of all PEs (TA cells) are updated on the basis of its previous state, the corresponding data/comparand values, and the search type selected by SFG. The type of the search determines the initial state of TA. Results of comparisons are stored in the Tag Array and must be further processed in order to extract global search output. Therefore, the Tag Array has built–in capability of performing fully parallel (bit–parallel word–parallel) exact match/mismatch search with a single comparand C2 and search mask SM2 – results are stored in T2.

Depending on a particular application it is necessary to add to the basic architecture of AGP a number of optional architecture components like counter of responders, select first circuit, generators of combinatorial configurations, controllers for sequencing processor operations in TA and extra memory registers for specific purposes.

The model is a parametrized generalization of many other simpler models proposed earlier and can substitute them functionally, i.e. process all algorithms derived for those models. For instance, models of single-comparand processors are equivalent to our model with the parameter m=1. In many cases, where multi-comparand searching is crucial, the AGP model provides a significant improvement of algorithms’ performance.

In [7], the FPGA implementation details of a very similar associative processor with extensive search capabilities is reported. The design was performed with Xilinx Foundation Series Software using low cost Xess XS40 demo board with a single Xilinx XC4005XL FPGA device. The device contains 196 CLBs and 112 IOBs. Only six logic searches were implemented, i.e. \{ =, \neq, <, >, \leq, \geq \}. The maximum size of the prototype obtained was 6x6x4. In that case 36 simultaneous comparisons were performed with the clock rate 68 ns per bit (max frequency 14.7 MHz). Asymptotic hardware complexity of the FPGA device is \(O(n^2)\) (it is dominated by the size of the Tag Array).

We hope that the implementation data of this device closely approximate an AGP implementation. It should be noted that the practical size of the associative graph processor depends strongly on the graph problem at hand.

5. Conclusions

In this paper, a model of an associative graph processor has been proposed which enables highly parallel search operations. We have also presented a group of basic procedures being used for implementing associative parallel algorithms. The AGP model is versatile and flexible enough to meet a wide range of requirements and to solve combinatorial problems from various complexity classes.

In contrast to other models of associative processors, the AGP model provides faster and more versatile parallel search operations and a number of additional operations on two–dimentional matrices that represent input data. It can emulate many simpler associative models used so far.

We intend to continue work on the development of the processor and to study new features useful in associative graph processing. We are also planning to extend its application domain and to design new efficient associative algorithms using multi–comparand search operations.

References


