PAPER

Orbital Systolic Algorithms and Array Processors for Solution of the Algebraic Path Problem

Stanislav G. SEDUKHIN[†a)], Toshiaki MIYAZAKI[†], *and* Kenichi KURODA[†], *Members*

SUMMARY The algebraic path problem (APP) is a general framework which unifies several solution procedures for a number of well-known matrix and graph problems. In this paper, we present a new 3-dimensional (3-D) orbital algebraic path algorithm and corresponding 2-D toroidal array processors which solve the $n \times n$ APP in the theoretically minimal number of $3n$ time-steps. The coordinated time-space scheduling of the computing and data movement in this 3-D algorithm is based on the modular function which preserves the main technological advantages of systolic processing: simplicity, regularity, locality of communications, pipelining, etc. Our design of the 2-D systolic array processors is based on a classical 3-D→2-D space transformation. We have also shown how a data manipulation (copying and alignment) can be effectively implemented in these array processors in a massively-parallel fashion by using a matrix-matrix multiply-add operation.

key words: *algebraic path problem, orbital systolic algorithms, array processors, data manipulation*

1. Introduction

The algebraic path problem (APP) is a general framework which unifies several solution procedures for a number of well-known matrix and graph problems such as matrix inversion, transitive closure, all-pairs shortest paths, maximum reliability paths, minimum spanning tree, etc. The APP has been studied extensively [1]–[8] because matrix computations and graph problems are among the most important computational problems in the computer science and engineering area. A summary of some results and historical notes on the APP may be found in [7], and a wide variety of APP applications are described in [5].

Due to the importance of the problem, there has been considerable research on implementing the APP (or particular instances) on 2-D and 1-D systolic and SIMD array processors, see extensive review in [8]–[10]. All 2-D implementations take $O(n)$ time-steps on $O(n^2)$ processing elements (PEs). It is well known [11] that the theoretically minimal time of implementing n^3 operations of the APP on an $n \times n$ array processor is $3n$, but with technologically undesirable global communication between processing elements. The localization of communication required for a systolic implementation has, to date, imposed a slowdown from $3n$ to $4n$ and more.

Actually, the traditional systolic implementation of the APP requires $5n - 4$ time-steps which is the length of the longest path in a *localized 3-D data dependence graph* of the APP algorithm [6], [12]. Deeper localization approaches that reduced the running time from $5n$ to $4n$ [10], [11], [13], [14] and the number of processing elements from n^2 to $n^2/3$ [9], [15], [16] have been proposed independently by a number of authors. However, all these optimized array processors sacrifice VLSI requirements by allowing irregular connections, non planar implementation, unbounded or bounded data broadcast, complicated control, etc.

Moreover, the main problem with these and other 2-D systolic algorithms is that all initial data must be properly prepared and stored in advance in a special (time-space skewed) form by using preprocessing in some additional and nontrivial parallel memory which, finally, has to stream this initial 2-D data into array processor through the boundary processing elements. The existence of this buffering memory and the required time of preprocessing for a 2-D data alignment are usually ignored in the well established traditional design and complexity analysis of the systolic algorithms [17]. As a result, the existing 2-D systolic array processors cannot be easily implemented in practice. These 2-D array processors, however, were very attractive solutions for a planar VLSI with the limited number of input/output (I/O) pins. Nowadays, the advances in the 3-D VLSI technology [18], stacked memory [19]–[21], and Giga-size sensor arrays [22] with a parallel read-out and embedded logic allow realization of massively parallel array processors with one I/O channel per PE. Moreover, the cost of floating-point computing is extremely cheap now[*] and will definitely be even cheaper in the future which makes array processors with billions of cores technically implementable and economically justified. However, this opening opportunity [24] requires a design of the new fine-grained massively-parallel algorithms and array processors which are based on the concurrent access to all initial data items while preserving the main technological advantages of the systolic processing, i.e., simplicity, regularity, locality of communications, pipelining, and balancing computations with I/O data. We would like to use this opportunity to design the massively-parallel algorithms and array processors for solving the APP.

Our main contributions are as follows:

1. We formally introduce a new 3-D algorithm for the APP with a modular time-step scheduling function by revising the well-known 2-D Guibas-Kung-Thompson

Manuscript received June 5, 2009.
Manuscript revised October 6, 2009.

[†]The authors are with the University of Aizu, Aizu-Wakamatsu-shi, 965–8580 Japan.
a) E-mail: sedukhin@u-aizu.ac.jp

DOI: 10.1587/transinf.E93.D.534

[*]The cost of computing has dropped from the about $\$10^{12}$/Gflop (1961) to \$0.42/Gflop (2007) [23].

algorithm [25].

2. We systematically design the new 2-D toroidal systolic array processors with a frontal plane I/O for solving the APP in the theoretically minimal time.
3. We propose a new technique for massively-parallel data manipulation such as data copying and alignment which is based on a standard matrix multiplication algorithm.

The rest of the paper is organized as follows. In Sect. 2 the Algebraic Path Problem and its applications are discussed in some detail. The 3-dimensional orbital APP algorithm which is based on the modular scheduling of computations and local data movements is introduced in Sect. 3. Based on this 3-D orbital APP algorithm, a few time-step optimal orbital systolic array processors are proposed in Sect. 4. Finally, Sect. 5 concludes the paper.

2. The Algebraic Path Problem

Let $G = (V, E, w)$ be a weighted graph, where $V = \{1, 2, \ldots, n\}$ is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $w : E \rightarrow S$ is an edge-weight function that assigns a weight from a closed semiring $(S, \oplus, \otimes, *, \bar{0}, \bar{1})$. A closed semiring $(S, \oplus, \otimes, *, \bar{0}, \bar{1})$ is an algebraic structure in a set S with two binary operations, addition $\oplus : S \times S \rightarrow S$ and multiplication $\otimes : S \times S \rightarrow S$, a unary operation called *closure* $* : S \rightarrow S$, and the constants $\bar{0}$ and $\bar{1}$ in S.

A path p is an arbitrary sequence of connected vertices $(i, k_1, k_2, \ldots, k_m, j)$ that begins with i and ends with j. The weight of a path p is defined as the product of all edges of the path:

$$w(p) = w(i, k_1) \otimes w(k_1, k_2) \otimes \ldots \otimes w(k_m, j).$$

The weight of a path of length 0, which begins and ends in the same vertex and does not contain edges, is defined to be $\bar{1}$.

Let $P(i, j)$ be the set of all paths from i to j. The APP is the problem to compute the sum of all paths from i to j for all pairs (i, j):

$$d_{i,j} = \bigoplus_{p \in P(i,j)} w(p). \tag{1}$$

The APP can also be formulated in matrix form. Like in [1], we introduce a matrix semiring $(S^{n\times n}, \boldsymbol{\oplus}, \boldsymbol{\otimes}, *, \bar{O}, \bar{I})$, a set of $n \times n$ matrices $S^{n\times n}$ over a closed scalar semiring $(S, \oplus, \otimes, *, \bar{0}, \bar{1})$ with two binary operations, matrix addition $\boldsymbol{\oplus} : S^{n\times n} \times S^{n\times n} \rightarrow S^{n\times n}$ and matrix multiplication $\boldsymbol{\otimes} : S^{n\times n} \times S^{n\times n} \rightarrow S^{n\times n}$, a unary operation called *closure of a matrix* $* : S^{n\times n} \rightarrow S^{n\times n}$, the zero $n \times n$ matrix $\bar{O}$ whose all elements equal to $\bar{0}$, and the $n \times n$ identity matrix $\bar{I}$ whose all main diagonal elements equal to $\bar{1}$ and $\bar{0}$ otherwise. Matrix addition and multiplication are defined as usual in linear algebra.

We associate a matrix $A = [a_{i,j}]$ in $S^{n\times n}$ with the weighted graph such that

$$a_{i,j} = \begin{cases} w(i, j) & \text{if } (i, j) \in E \\ \bar{0} & \text{if } (i, j) \notin E. \end{cases}$$

If we define a matrix $D = [d_{i,j}]$ in $S^{n\times n}$ whose entries are values of $d_{i,j}$ for all pairs (i, j), the matrix D can be expressed in terms of the matrix A as follows (see [2] for details):

$$D = A^* = \bigoplus_{m \geq 0} A^m = \bar{I} \boldsymbol{\oplus} A \boldsymbol{\oplus} A^2 \boldsymbol{\oplus} A^3 \boldsymbol{\oplus} \cdots . \tag{2}$$

Equation (2) is the matrix formulation of the APP. Thus, the APP in matrix form can be stated as the problem to compute A^*, the closure of a matrix A defined in (2). In addition, A^* in (2) can be rewritten as follows:

$$\begin{aligned} A^* &= \bar{I} \boldsymbol{\oplus} A \boldsymbol{\oplus} A^2 \boldsymbol{\oplus} A^3 \boldsymbol{\oplus} \cdots \\ &= \bar{I} \boldsymbol{\oplus} A \boldsymbol{\otimes} (\bar{I} \boldsymbol{\oplus} A \boldsymbol{\oplus} A^2 \boldsymbol{\oplus} A^3 \boldsymbol{\oplus} \cdots) \\ &= \bar{I} \boldsymbol{\oplus} A \boldsymbol{\otimes} A^*. \end{aligned} \tag{3}$$

The APP in matrix form can also be stated as the problem to solve (3) for A^*.

Applications of the APP are obtained by specializing a scalar closed semiring. Some of them are detailed below:

- *Transitive and reflexive closure*: the weights a_{ij} are taken from $S = \{0, 1\}$. $\oplus = \vee$, $\otimes = \wedge$, $a^* = 1$ for all a in S, $\bar{0} = 0$, and $\bar{1} = 1$. The closure of a matrix gives the transitive and reflexive closure.
- *All-pairs shortest paths*: the weights a_{ij} are taken from $S = R_+ \cup \{\infty\}$, where R_+ is the set of positive real numbers. $\oplus = \min$, $\otimes = +$, $a^* = 0$ for all a in S, $\bar{0} = \infty$, and $\bar{1} = 0$. The closure of a matrix gives the all-pairs shortest paths.
- *Maximum cost paths*: the weights a_{ij} are taken from $S = R_+ \cup \{+\infty, -\infty\}$, where R_+ is the set of positive real numbers. $\oplus = \max$, $\otimes = +$, $a^* = 0$ for all a in S, $\bar{0} = -\infty$, and $\bar{1} = 0$. The closure of a matrix gives the maximum cost (critical) paths, or $+\infty$ if there are paths of unbounded cost [1].
- *Maximum capacity paths* (also called *tunnel problem* or *network capacity problem* [7]): the weights a_{ij} are taken from $S = R_+ \cup \{\infty\}$, where R_+ is the set of positive real numbers. $\oplus = \max$, $\otimes = \min$, $a^* = \infty$ for all a in S, $\bar{0} = 0$, and $\bar{1} = \infty$. The closure of a matrix gives the maximum capacity paths.
- *Maximum reliability paths*: the weights a_{ij} are taken from $S = [0, 1]$ and can be considered as reliabilities of the information transport between the two connected vertices. $\oplus = \max$, $\otimes = \times$, $a^* = 1$ for all a in S, $\bar{0} = 0$, and $\bar{1} = 1$. The closure of a matrix gives the most reliable paths, i.e., the path with the highest probability between any two vertices in a graph.
- *Minimum cost spanning tree*: the weights $a_{ij} = a_{ji}$ are taken from $S = R_+ \cup \{\infty\}$, where R_+ is the set of positive real numbers. $\oplus = \min$, $\otimes = \max$, $a^* = 0$ for all a in S, $\bar{0} = \infty$, and $\bar{1} = 0$. Let $D = (d_{ij})$ is the closure of a matrix, then its entries $d_{ij} = a_{ij}^{(0)}$ are the edges of the minimum cost spanning tree [4].

- *Inverse of a real non-singular matrix*: the weights a_{ij} are taken from $S = R$. $\oplus$ and $\otimes$ are the conventional arithmetic $+$ and $\times$ operations on R respectively, $a^* = (1-a)^{-1}$ for all a in R with $a \neq 1$ (a^* is undefined for $a = 1$), $\bar{0} = 0$, and $\bar{1} = 1$. The closure of a matrix gives $(I - A)^{-1}$ (see [2] for details).

3. 3-Dimensional Orbital APP Algorithm

3.1 The Generic APP Algorithm

After initialization, $a_{i,j}^{(0)} = a_{i,j}$, $1 \le i,j \le n$, the generic algorithm for solution of the APP can be represented in the following compact form:

Algorithm 1

```
for $k = 1 : n$ do
    for all $1 \le i, j \le n$ do
        $a_{i,j}^{(k)} \leftarrow a_{i,j}^{(k-1)} \oplus a_{i,k}^{(k-1)} \otimes (a_{k,k}^{(k-1)})^* \otimes a_{k,j}^{(k-1)}$;
    end for all
end for
```

However, to resolve data dependency, the Algorithm 1 can be rewritten as follows [11], [13].

Algorithm 2

```
for $k = 1 : n$ do
    for all $1 \le i, j \le n$ do
        $a_{i,j}^{(k)} \leftarrow \begin{cases} (a_{i,j}^{(k-1)})^* & : i = j = k; \\ a_{i,k}^{(k)} \otimes a_{i,j}^{(k-1)} & : i = k \neq j; \\ a_{i,j}^{(k-1)} \otimes a_{k,j}^{(k)} & : j = k \neq i; \\ a_{i,j}^{(k-1)} \oplus a_{i,k}^{(k)} \otimes a_{k,j}^{(k-1)} & : (i \neq k)\&(j \neq k); \end{cases}$
    end for all
end for
```

As it can be seen from the Algorithm 2, there are four different types of computations, depending on the index conditions. It was proven [11] that the time-minimal scheduling function for the Algorithm 2 is

$$\mathbf{step}(a_{i,j}^{(k)}) = \begin{cases} 3k-2 & : \ i = j = k; \\ 3k-1 & : \ i = k \neq j; \\ 3k-1 & : \ j = k \neq i; \\ 3k & : \ (i \neq k)\&(j \neq k), \end{cases} \tag{4}$$

where $\mathbf{step}(a_{i,j}^{(k)})$ assigns a time-step of computing the element $a_{i,j}$ on k-th iteration, assuming that each elementary computation in the Algorithm 2 is performed in one step. From (4) we can conclude that the running time for *any* parallelization of this algorithm is at least $3n$. This schedule assumes an unbounded number of $O(n^2)$ processing elements with a global communication: in particular, broadcast of the pivot $a_{k,k}^{(k)} = (a_{k,k}^{(k-1)})^*$, the pivot column $\{a_{i,k}^{(k)} | 1 \le i \le n\}$, and the pivot row $\{a_{k,j}^{(k)} | 1 \le j \le n\}$ on each k-th iteration are necessary. Possible localization and regularization of a global communication leads to increasing the running time from $3n$ to $5n$ [6], [12].

3.2 The Algorithm for the Idempotent APP

For the APP with an *idempotent* semiring, where $a = a \oplus a$, the closure operation is $(a_{k,k}^{(k-1)})^* = \bar{1}$. Moreover, because $a_{k,k}^{(k-1)} = \bar{0}$ for any $k \in \{1, 2, \ldots, n\}$ then for $i = k$, $a_{k,j}^{(k)} = a_{k,j}^{(k-1)} \oplus a_{k,k}^{(k-1)} \otimes a_{k,j}^{(k-1)} \equiv a_{k,j}^{(k-1)}$, and for $j = k$, $a_{i,k}^{(k)} = a_{i,k}^{(k-1)} \oplus a_{i,k}^{(k-1)} \otimes a_{k,k}^{(k-1)} \equiv a_{i,k}^{(k-1)}$. Therefore, for an idempotent semiring the Algorithm 1 can be simplified as follows:

Algorithm 3

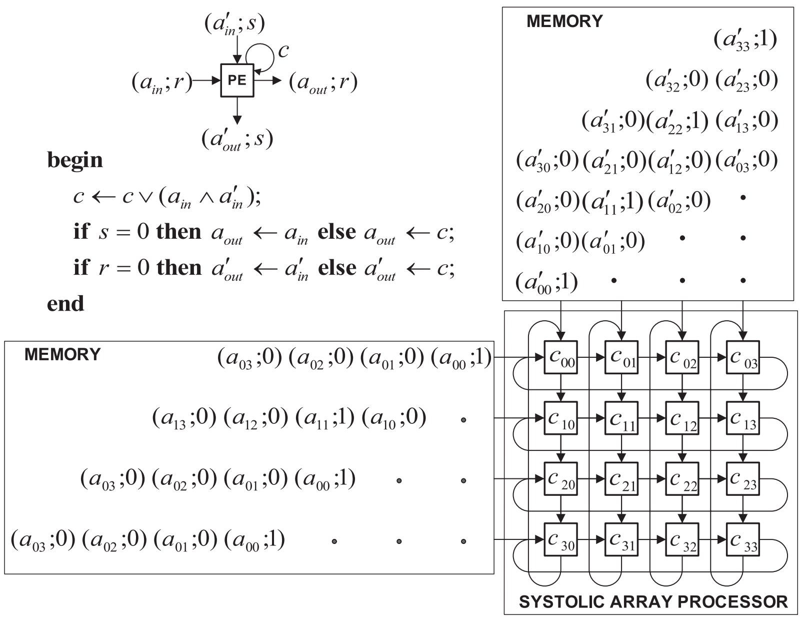
```
for $k = 1 : n$ do
    for all $(1 \le i \neq k \le n)\&(1 \le j \neq k \le n)$ do
        $a_{i,j}^{(k)} \leftarrow a_{i,j}^{(k-1)} \oplus a_{i,k}^{(k-1)} \otimes a_{k,j}^{(k-1)}$;
    end for all
end for
```

The modified Algorithm 3 requires only $n(n-1)^2$ "multiply-add" operations which is less than n^3 operations in the classical Floyd-Warshall algorithm for the transitive closure of a directed graph. In the Algorithm 3 there exist only one type of computing $a_{i,j}^{(k)}$, which depends on the previously computed elements $a_{i,j}^{(k-1)}$, $a_{i,k}^{(k-1)}$, and $a_{k,j}^{(k-1)}$.

It is well known that there is the straightforward relation between paths problem and matrix-matrix multiplication (MMM) [26]. In spite of that, in the MMM algorithm, any nested order of the i, j, k loops is permissible [27] while in the APP algorithm, an iteration loop k has to be the outermost one. Therefore, the APP Algorithm 3 has more strict data dependencies than the MMM algorithm and these dependencies should be resolved to update the entire $n \times n$ matrix $A^{(k)} = [a_{i,j}^{(k)}]$ in the Algorithm 3 before moving on to the next $(k + 1)$ iteration [28].

3.3 The Guibas-Kung-Thompson Algorithm

Among all existing 2-D systolic array processors for the APP there exist a not trivial systolic design for the reflexive transitive closure of an $n \times n$ boolean matrix A due to Guibas, Kung, and Thompson (GKT) [25]. A systolic system for GKT algorithm is shown in Fig. 1. The GKT algorithm is based on three-pass matrix-matrix multiplication $C \leftarrow C + A \times A'$, and, therefore, requires $3n^3$ "multiply-add" operations, i.e., three times more than the classical Floyd-Warshall algorithm. However, only $5n - 2$ time-steps are needed for GKT algorithm to implement all these operations and obtain the solution matrix $A^* \equiv C = [c_{i,j}]$ on an $n \times n$ toroidal array processor. The side-around (toroidal) connections are needed to return an $n \times n$ matrix $A + I = [a_{i,k}]$, where I is the identity matrix, and its copy $A' = [a'_{k,j}]$ for the second and third passes. After completion, the resulting elements of a matrix $C = [c_{i,j}]$, which is a zero matrix initially, are resided in PE$\langle i, j \rangle$, $0 \le i, j < n$, (see Fig. 1). Note that here, in contrast to a matrix multiplication, all elements $a_{i,k}$, $a'_{k,j}$, and $c_{i,j}$ are updated at every pass.



```
begin
    $c \leftarrow c \vee (a_{in} \wedge a'_{in})$;
    if $s = 0$ then $a_{out} \leftarrow a_{in}$ else $a_{out} \leftarrow c$;
    if $r = 0$ then $a'_{out} \leftarrow a'_{in}$ else $a'_{out} \leftarrow c$;
end
```

$(a'_{in}; s)$
c
$(a_{in}; r)$
PE
$(a_{out}; r)$
$(a'_{out}; s)$
MEMORY
$(a'_{33};1)$
$(a'_{32};0)\ (a'_{23};0)$
$(a'_{31};0)(a'_{22};1)\ (a'_{13};0)$
$(a'_{30};0)\ (a'_{21};0)(a'_{12};0)\ (a'_{03};0)$
$(a'_{20};0)(a'_{11};1)\ (a'_{02};0)$
$(a'_{10};0)(a'_{01};0)$
$(a'_{00};1)$
MEMORY
$(a_{03};0)\ (a_{02};0)\ (a_{01};0)\ (a_{00};1)$
$(a_{13};0)\ (a_{12};0)\ (a_{11};1)\ (a_{10};0)$
$(a_{03};0)\ (a_{02};0)\ (a_{01};0)\ (a_{00};1)$
$(a_{03};0)\ (a_{02};0)\ (a_{01};0)\ (a_{00};1)$
c_{00} c_{01} c_{02} c_{03}
c_{10} c_{11} c_{12} c_{13}
c_{20} c_{21} c_{22} c_{23}
c_{30} c_{31} c_{32} c_{33}
SYSTOLIC ARRAY PROCESSOR

Fig. 1 A systolic system architecture for the GKT algorithm and PE function on each time-step.

To control the actions of the PE$\langle i, j\rangle$, the GKT algorithm uses the boolean control signals r and s such that $r = 1$ and $s = 1$ for the diagonal elements of the matrices A and A', respectively, and $r = s = 0$ otherwise. The detailed proof of the validity of this algorithm can be found in [29]. The GKT algorithm for the transitive closure can be extended to solve the other APP applications with an idempotent semiring. However, as it was mentioned before, the problem with this and other 2-D systolic algorithms is that all initial data should be properly organized in advance and stored in the parallel memory with a not trivial behavior (see Fig. 1).

3.4 A New 3-D Orbital APP Algorithm

Because the GKT algorithm is based on the three-pass matrix multiplication in the corresponding semiring:

$$C \leftarrow C \oplus A \otimes A' \Longleftrightarrow c_{i,j} = c_{i,j} \oplus \sum_{k=0}^{n-1} a_{i,k} \otimes a'_{k,j},$$

we can use our previously proposed [30] 3-D orbital matrix-matrix multiply-add algorithms to solve the APP with idempotent semirings. Like the GKT algorithm we use as input an $(n \times n)$ matrix $A = [a_{i,k}] = A \bar{\oplus} \bar{I}$, its copy $A' = [a'_{k,j}] = A$, and a matrix $C = [c_{i,j}]$ all elements of which are initially equal to $\bar{0}$. The boolean control signals r and s are also added to the elements of matrices A and A', respectively. We can choose one of the allowable for the 3-D orbital MMA algorithm modular time-step scheduling functions, for example,

$$\mathbf{step}(c^{(k)}_{i,j}) = (\alpha^T \cdot p) \bmod n = (-i - j + k) \bmod n, \quad (5)$$

where $p = (i, j, k)^T, \alpha^T = (-1, -1, 1)$, T is a transpose sign, and start processing from the $\mathbf{step}(c^{(k)}_{i,j}) = 0$. Note that here, $i, j, k \in [0, n-1]$ and, in general, a scheduling vector $\alpha^T \in (\pm 1, \pm 1, \pm 1)$ (see [30] for details). Moreover, in (5) we assume that elementary computation $c^{(k)}_{i,j} \leftarrow c^{(k-1)}_{i,j} \oplus a_{i,k} \otimes a'_{k,j}$ is performed within one time-step.

The 3-D parallel algorithm for the idempotent APP and given scheduling function (5) is shown as Algorithm 4, where a statement $points(i, j, k) = [0 \le i, j, k < n] \& [(k - j - i) \bmod n]$ and "||" means that the statements connected by || are independent on each other and can be performed in parallel. The main part of this algorithm (lines 2–17) is a simple matrix-matrix multiply-add in the 3-D toroidal index space $\mathcal{I} = \{(i, j, k)^T | 0 \le i, j, k < n\}$, but with different to the matrix multiplication reassignments (lines 6–14) which are depend on the index conditions. We realistically assume that an elementary unit of processing (lines 4–15), which includes both computing and data rolling, is implemented in one time-step, which is independent on the problem size.

The initial distribution of matrices A, A', and C in a 3-D $n \times n \times n$ toroidal index space $\mathcal{I} = \{(i, j, k)^T | 0 \le i, j, k < n\}$ is predefined by the selected time-step function at the initial step, i.e., each active by the time-step function $\mathbf{step}(c^{(k)}_{i,j}) = 0$ an index point $p = (i, j, k)^T \in \mathcal{I}$ has to have initially the elements $a_{i,k} = a_{in}, a'_{k,j} = a'_{in}$, and $c_{i,j} = c_{in} = \bar{0}$ which are input data for the first pass. The distribution of active computations on each time-step in the 3-D toroidal index space is shown in Fig. 2. The toroidal side-around connections are not shown for simplicity and directions of data rolling are symbolically represented by an orbital sign above the picture.

On each time-step at each of n^2 active index point $p = (i, j, k)^T \in \mathcal{I}$ we have to implement a "multiply-add" operation (line 5 in the Algorithm 4) and, after corresponding assignments (lines 6–15), cyclically shift (roll) output data to the neighbor index points depending on the

Algorithm 4

```
1: for $pass = 1 : 3$ do
2:   for $step = 0 : n-1$ do
3:     for all $points(i, j, k) = step$ do
4:       begin
5:       $c_{out} \leftarrow c_{in} \oplus a_{in} \otimes a'_{in}$;
6:       case($rs$) :
7:         % black element update: $i = j = k$ (●)
8:         (11): $a_{out} \leftarrow c_{out}$ || $a'_{out} \leftarrow c_{out}$;
9:         % red element update: $j = k \neq i$ (◐)
10:        (01): $a_{out} \leftarrow a_{in}$ || $a'_{out} \leftarrow c_{out}$;
11:        % white element update: $(i \neq k)\&(j \neq k)$ (○)
12:        (00): $a_{out} \leftarrow a_{in}$ || $a'_{out} \leftarrow a'_{in}$;
13:        % blue element update: $i = k \neq j$ (◑)
14:        (10): $a_{out} \leftarrow c_{out}$ || $a'_{out} \leftarrow a'_{in}$;
15:      end
16:    end for all
17:  end for step
18: end for pass
```

given scheduling function. Because for the scheduling function (5), $\alpha^T = (-1, -1, 1)$, $a_{i,k} = a_{out}$ is moved opposite to the j-axis (orbit), $a'_{k,j} = a'_{out}$ is propagated opposite to the i-axis, and $c_{i,j} = c_{out}$ is cyclically shifted along the k-axis (see Fig. 2).

After n "compute-and-*roll*" time-steps the all updated elements of matrices A, A', and C will be located at the same index points from which we have started processing, i.e., all data is ready for the second and third passes. Totally, three passes are required to finalize the Algorithm 4.

Theorem 1: The 3-D orbital Algorithm 4 solves the idempotent APP in $3n$ "compute-and-roll" steps.

Proof At each *step* (line 2), the innermost loop (lines 4–16) activate simultaneously n^2 index points. If a combination of "computing" (line 5) and "rolling" data (lines 6–15) for all active points is performed in a single step then n steps are needed to implement one pass of the algorithm. Because three passes are required (see the outermost loop (lines 1–18)), totally $3n$ "compute-and-roll" steps are needed to solve the idempotent APP by the Algorithm 4. □

Recall that $3n$ steps is the theoretically low bound which follows from the Eq. (4). Because at each step, n^2 "compute-and-roll" operations are performed, totally, $3n^3$ "multiply-add" operations are needed to complete the APP Algorithm 4, i.e., to find a solution $A^* = C$.

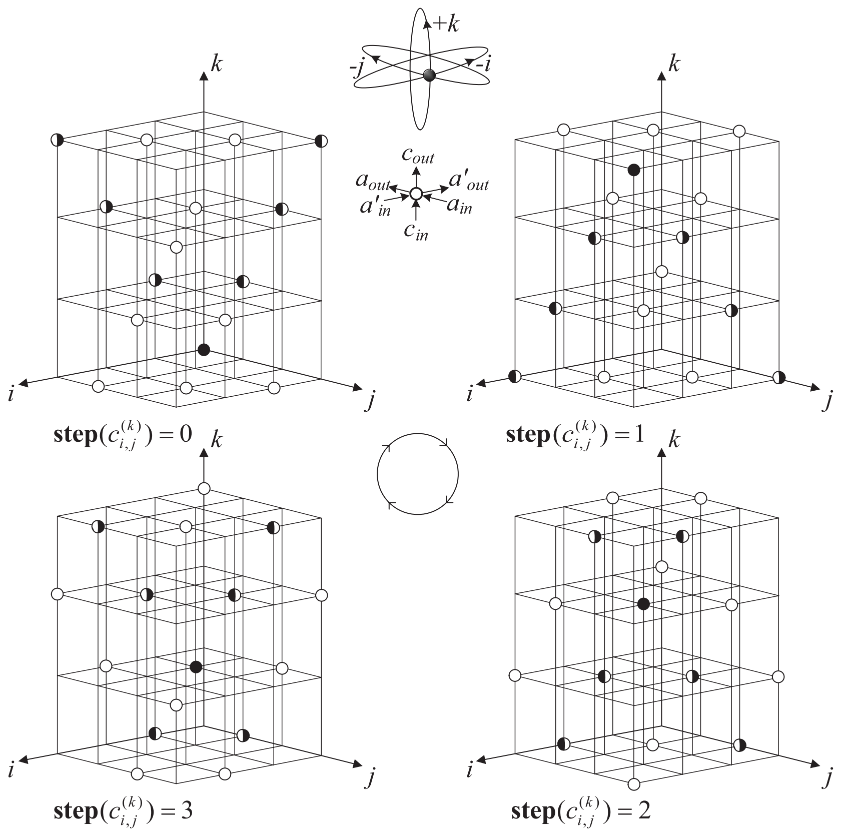
4. 2-D Orbital Array Processors for the APP

It is straightforward to find the all admissible data distributions for the 2-D toroidal array processors by using the appropriate 3-D→2-D linear projections. These linear projections map the 3-D index space of the algorithm into 2-D array processor space while keeping the same time complexity of implementation. For example, three initial data distributions for a 2-D $n \times n$ toroidal array processor, which are obtained by projection of a 3-D index space $\mathcal{I}$ of the APP Algorithm 4 (see Fig. 2) along i-, j-, and k-axis (orbits) are shown in Fig. 3 for $n = 4$. Obviously, each active index point $p = (i, j, k)^T \in \mathcal{I}$ is mapped into either PE(j, k) or PE(i, k) or PE(i, j) for i-, j- or k-projection, respectively. In Fig. 3, the PE with a label (ijk) has initially elements $a_{i,k} = a_{in}$, $a'_{k,j} = a'_{in}$, and $c_{i,j} = c_{in} = \bar{0}$. It is easy to proof the following

Theorem 2: Each 2-D $n \times n$ orbital array processor shown in Fig. 3 solves the idempotent APP in $3n$ "compute-and-roll" steps.

Proof Any linear projection of the 3-D index space $\mathcal{I}$ along/opposite the basis vectors $\mathbf{e}_i = (1, 0, 0)^T, \mathbf{e}_j = (0, 1, 0)^T, \mathbf{e}_k = (0, 0, 1)^T$ guarantees that on each "compute-and-roll" time-step no more than one active index point will be mapped into each PE, i.e. all n^2 active index points are mapped into n^2 different PEs of a planar array processor. It is follows from the simple condition that the scalar product $\alpha^T \cdot \mathbf{e}_x \neq 0$ where $\mathbf{e}_x \in \{\mathbf{e}_i, \mathbf{e_j}, \mathbf{e}_k\}$, α is a scheduling vector, and $\alpha = (-1, -1, 1)^T$ for our example. Moreover, the linear projections of a cubical-like index space $\mathcal{I}$ along/opposite i-, j-, or k-axis define the planar array processors with the minimal number of $n \times n$ toroidally connected PEs. To complete the Algorithm 4, there are $3n$ such "compute-and-roll" steps. □

As it can be seen from Fig. 3, only projection along j-axis satisfies the distribution of an input matrix $A = [a_{i,k}]$ among PEs in a canonical form while other projections require initial skewing of this matrix. For projection along j-axis, the different data distributions on each" compute-and-roll" step are demonstrated in Fig. 4. This projection, however, requires the matrices A' and C to be initially skewed. The matrix $A' = [a'_{k,j}]$, which is a copy of the matrix $A = [a_{i,k}]$, should be distributed within an array processor in (k, j)-fashion. This matrix can be produced (copied) in the required skewed form by using a matrix-matrix multiplication $A' \leftarrow A^T \times I + A'$ before the main processing, where I is the identity matrix, i.e., $a'_{k,j} = \sum_{i=0}^{n-1} a_{i,k} \cdot e_{i,j}$, where initially A' is set to zero matrix and $e_{i,j} \in I$ with $e_{i,j} = 1$ if $i = j$ and $e_{i,j} = 0$, otherwise. In this operation, a matrix A, which was loaded in a canonical form, is involved in a computing in the transposed form but no real transpose is needed. We assume that the identity matrix I is permanently resided in our array processor $\{\text{PE}(i, k) | 0 \leq i, k < n\}$ such that $e_{i,j} \in I$ is stored in PE(i, k). It is not difficult to see that a matrix I is distributed in skewed form. Recall that on each time-step every PE(i, k) involves in a processing (compute-and-roll) the active index point $p = (i, j, k)^T \in \mathcal{I}$, i.e., three scalar operands: $e_{i,j}, a_{i,k}$ and $a'_{k,j}$. Practically, a real scalar multiplication by the constants "0" and "1" can be easily avoided. After loading of a matrix A into array processor, a copying of a matrix A' with a required skewing is implemented on the same array processor by using the same time-step function. i.e., the elements $e_{i,j}$ and $a'_{k,j}$ will be rolled while an element $a_{i,k}$ is fixed in PE(i, k). Of course, this preprocessing stage requires additionally n time-steps. After n compute-and-roll steps each PE(i, k) contains the elements of an index point $(i, j, k)^T$, i.e., it has the required elements $a'_{k,j} = a_{k,j}$ and $a_{i,k}$, and an array processor is ready to start the main processing as it was described above.



+k
-j
-i
c_{out}
a_{out}
a'_{out}
a'_{in}
a_{in}
c_{in}
k
i
j
$\mathbf{step}(c_{i,j}^{(k)}) = 0$
$\mathbf{step}(c_{i,j}^{(k)}) = 1$
$\mathbf{step}(c_{i,j}^{(k)}) = 3$
$\mathbf{step}(c_{i,j}^{(k)}) = 2$

Fig. 2 Distribution of active computations in a 3D toroidal index space on each step under the scheduling function $\mathbf{step}(c_{i,j}^{(k)}) = (-i - j + k) \bmod n$.

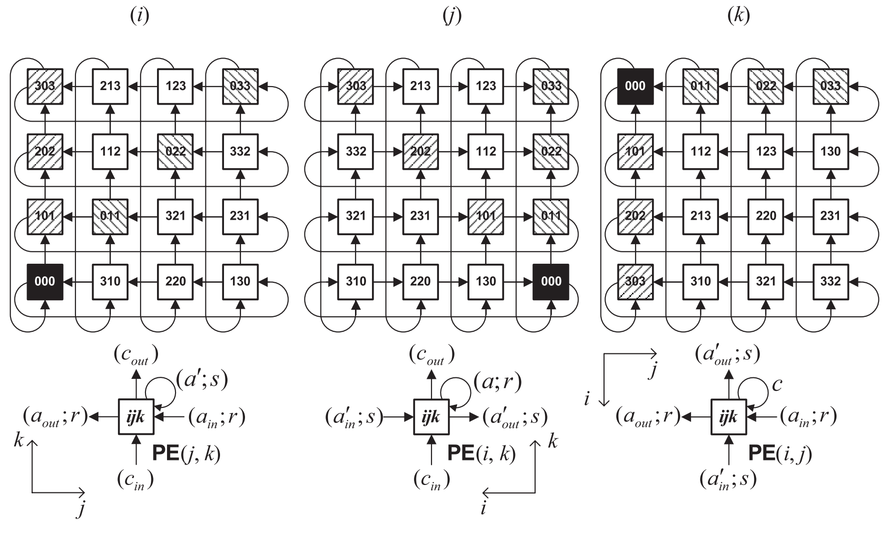


Fig. 3 Three initial data distributions as 2-D projections of the 3-D index space along i-, j-, and k-axis as well as coordinates and input/output data of each PE.

After $3n$ time-steps, the resulting matrix $C = [c_{i,j}]$ will be stored in the array processor in a skewed (i, j)-form, as it is depicted in Fig. 3(j). Therefore, by completion of the main processing, it might be desirable to return a resulting matrix C into canonical (i, k)-form and perform a data-parallel output. This post-processing stage can be done by using one more matrix-matrix multiplication $C' \leftarrow C \times I^T + C'$, where $c'_{i,k} = \sum_{j=0}^{n-1} c_{i,j} \cdot e_{k,j}$ and $C' = [c_{i,k}]$ is a zero matrix initially. Here, we assume again that the identity matrix I is permanently resided in an array processor in a skewed (k, j)-fashion. This matrix is involved in a computing in the transposed form with $e_{k,j} = e_{j,k}$. Obviously, after

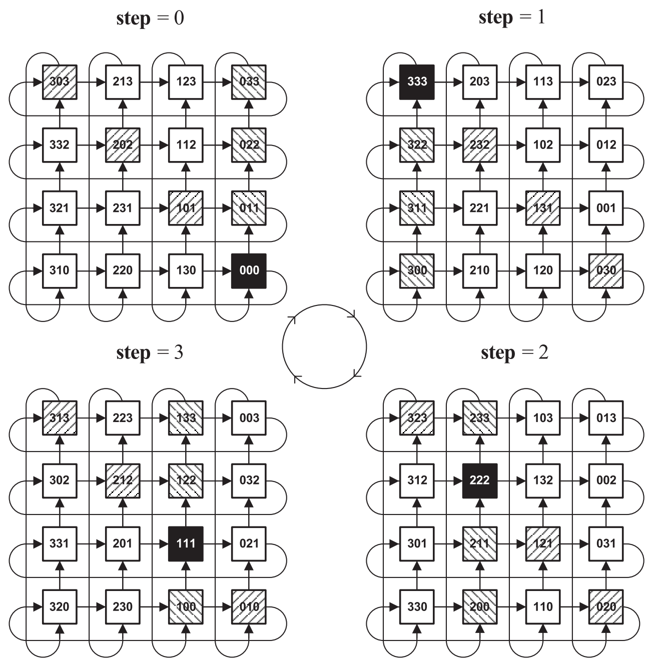


Fig. 4 Data distributions on each time-step of computing for a 2-D projection along j-axis.

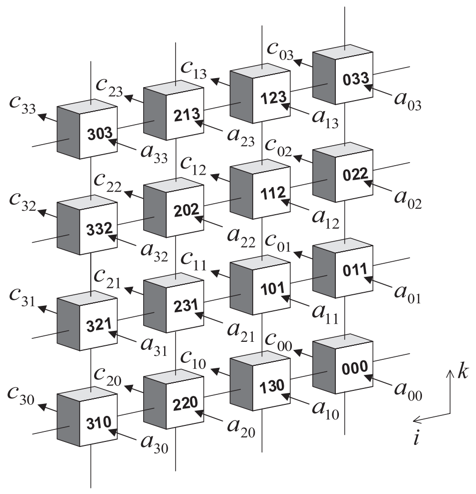

c_{33} c_{23} c_{13} c_{03}
303 213 123 033
a_{33} a_{23} a_{13} a_{03}
c_{32} c_{22} c_{12} c_{02}
332 202 112 022
a_{32} a_{22} a_{12} a_{02}
c_{31} c_{21} c_{11} c_{01}
321 231 101 011
a_{31} a_{21} a_{11} a_{01}
c_{30} c_{20} c_{10} c_{00}
310 220 130 000
a_{30} a_{20} a_{10} a_{00}
k
i

Fig. 5 2-D Frontal Plane Array Processor for the APP obtained by a projection of the 3-D index space along j-axis. The side-around toroidal connections are not shown for simplicity.

n "compute-and-roll" time-steps, the elements $c'_{i,k} = c_{i,k}$ will be correctly located in PE(i,k), $0 \leq i,k < n$, in a canonical form and ready for parallel output (see Fig. 5).

It can be shown that if a matrix A is loaded in the canonical form then a projection along i-axis also requires pre- and post-processing for data copying and alignment while a projection along k-axis will demand only two pre-processing stages since the resulting matrix C is resided inside an array processor in the canonical form as in the famous Cannon's algorithm for matrix multiplication [31]. Thus, for any discussed projection, the total $5n$ time-steps are needed to solve the APP. This total time includes not only the minimal possible $3n$ time-steps to implement the APP algorithm itself, but also an additional time for the required copying and data alignment which is usually missed in the traditional systolic algorithms complexity analysis.

5. Conclusion

In this paper we have introduced a new 3-D orbital algorithm for the APP with a modular time-step scheduling function which guaranties the minimal time of the APP solution while preserving a systolic style of processing: simplicity, regularity, locality of communications, pipelining, etc. The new 2-D toroidal systolic array processors with a frontal plane I/O have been systematically designed as linear projections of the 3-D APP algorithm. We have also proposed a new technique for massively-parallel data manipulation such as data copying and alignment which is based on a matrix-matrix multiply-add operation. We guess that the frontal plane systolic array processors can be relatively easily implemented and efficiently used for many important applications with multidimensional data streams.

Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions which have improved this paper.

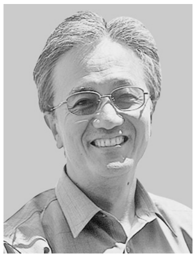
References

[1] D.J. Lehmann, "Algebraic structures for transitive closure," Theor. Comput. Sci., vol.4, no.1, pp.59–76, 1977.
[2] G. Rote, "A systolic array algorithm for the algebraic path problem," Computing, vol.34, pp.191–219, 1985.
[3] Y. Robert and D. Trystram, "Parallel implementation of the algebraic path problem," Proc. Conference on Algorithms and Hardware for Parallel Processing on CONPAR 86, pp.149–156, New York, NY, USA, Springer-Verlag New York, 1986.
[4] B.M. Maggs and S.A. Poltkin, "Minimum-cost spanning tree as a path-finding problem," Inf. Process. Lett., vol.26, no.6, pp.291–293, 1988.
[5] G. Rote, "Path problems in graphs," Computing Supplementum, vol.7, pp.155–189, 1990.
[6] S.G. Sedukhin, "Design and analysis of systolic algorithms for the algebraic path problem," Computers and Artificial Intelligence, vol.11, pp.269–292, 1992.
[7] E. Fink, "A survey of sequential and systolic algorithms for the algebraic path problem," Technical Report CS-92-37, Department of Computer Science, University of Waterloo, 1992.
[8] D. Cachera, S. Rajopadhye, T. Risset, and C. Tadonki, "Parallelization of the algebraic path problem on linear SIMD/SPMD arrays," Technical Report 1346, Irisa, 2001.
[9] C.T. Djamégni, P. Quinton, S. Rajopadhye, and T. Risset, "Derivation of systolic algorithms for the algebraic path problem by recurrence transformations," Parallel Comput., vol.26, no.11, pp.1429–1445, 2000.
[10] T. Risset and Y. Robert, "Synthesis of processor arrays for the algebraic path problem: Unified old results and deriving new architectures," Parallel Processing Letters, no.11, pp.19–28, 1991.
[11] S. Rajopadhye, "An improved systolic algorithm for the algebraic path problem," Integr. VLSI J., vol.14, no.3, pp.279–296, 1993.

[12] S.Y. Kung, S.C. Lo, and P.S. Lewis, "Optimal systolic design for the transitive closure and the shortest path problems," IEEE Trans. Comput., vol.36, no.5, pp.603–614, 1987.
[13] P.Y. Chang and J.C. Tsay, "A family of efficient regular arrays for algebraic path problem," IEEE Trans. Comput., vol.43, no.7, pp.769–777, July 1994.
[14] T. Takaoka and K. Umehara, "An efficient vlsi algorithms for the all pairs shortest path problem," J. Parallel Distrib. Comput., vol.16, no.3, pp.265–270, 1992.
[15] A. Benaini and Y. Robert, "Space-time minimal systolic arrays for Gaussian elimination and the algebraic path problem," Parallel Comput., vol.15, no.1-3, pp.211–225, 1990.
[16] C. Scheiman and P. Cappello, "A processor-time-minimal systolic array for transitive closure," IEEE Trans. Parallel Distrib. Syst., vol.3, no.3, pp.257–269, 1992.
[17] D. Lavenier, P. Quinton, and S. Rajopadhye, "Advanced systolic design," in Digital signal processing for multimedia systems, ed. K.K. Parhi and T. Nishitani, pp.657–692, CRC Press, 1999.
[18] M. Koyanagi, T. Fukushima, and T. Tanaka, "Three-dimensional integration technology and integrated systems," Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific, pp.409–415, Jan. 2009.
[19] Intel, "Tera-scale Computing Research Program," Website, http://techresearch.intel.com/articles/Tera-Scale/1421.htm, 2009.
[20] G. Loh, "3D-stacked memory architectures for multi-core processors," Computer Architecture, 2008. ISCA '08. 35th International Symposium, pp.453–464, June 2008.
[21] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS," Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. pp.98–589, IEEE International, Feb. 2007.
[22] E.H. Heijne, "Gigasensors for an attoscope: Catching quanta in CMOS," IEEE Solid-State Circuits Newsletter, vol.13, no.4, pp.28–34, Oct. 2008.
[23] Wikipedia, "FLOPS: Cost of computing," Website, http://en.wikipedia.org/wiki/GFlop, 2009.
[24] S.M. Chai and D.S. Wills, "Systolic opportunities for multidimensional data streams," IEEE Trans. Parallel Distrib. Syst., vol.13, no.4, pp.388–398, 2002.
[25] L.J. Guibas, H.T. Kung, and C.D. Thompson, "Direct VLSI implementation of combinatorial algorithms," Proc. First Caltech Conference on VLSI, pp.509–525, Pasadena, CA, California Institute of Technology, Jan. 1979.
[26] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms, Second ed., The MIT Press and McGraw-Hill Book Company, 2001.
[27] G.H. Golub and C.F.V. Loan, Matrix Computations, Third ed., The John Hopkins University Press, Baltimore, Maryland, 1996.
[28] M. Penner, J.S. Park, and V.K. Prasanna, "Optimizing graph algorithms for improved cache performance," IEEE Trans. Parallel Distrib. Syst., vol.15, no.9, pp.769–782, 2004.
[29] J.D. Ullmann, Computational aspects of VLSI, Computer Science Press, New York, 1984.
[30] A.S. Zekri and S.G. Sedukhin, "Computationally efficient parallel matrix-matrix multiplication on the torus," ISHPC - 6th Int. Symp. on High Performance Computing, pp.219–226, Springer-Verlag, Sept. 2005.
[31] L. Cannon, A cellular computer to implement the Kalman filter algorithm, Ph.D. Thesis, Montana State Univ., 1969.

Stanislav G. Sedukhin is a professor and head of the Computer Engineering Division at the University of Aizu. He received his Ph.D. and Dr. Sci. (habilitation) from the Russian Academy of Sciences in 1982 and 1993, respectively. His research interests are in architectural synthesis of application-specific array processors and massively-parallel algorithms. Dr. Sedukhin is a member of ACM, IEEE.

Toshiaki Miyazaki is a professor of the University of Aizu, Fukushima, Japan. His research interests are in reconfigurable hardware systems, wireless sensor networks, and adaptive networking technologies. He received his Ph.D. degree in electronic engineering from the Tokyo Institute of Technology in 1994. From 1983 to 2005, he had been worked for NTT. He is a member of IEEE (CS, CAS, ComSoc), and IPSJ.



Kenichi Kuroda received his Dr. Eng. in 1991 from Tokyo University. From 1973 to 1995, he was a researcher at R&D center in NTT and was engaged with SAW, superconductor, and X-ray optics devices. Currently, he is a professor at the University of Aizu. His current research interests are VLSI design education and adaptive systems based on FPGAs. He is a member of IPSJ.