Implementing linear algebra algorithms on high performance architectures *

L. Aleksandrov, M. Candev
Center of Informatics, Bulgarian Academy of Sciences
G.Bonchev Str. 25-A, 1113 Sofia, Bulgaria

H. N. Djidjev
Department of Computer Science, Rice University,
P.O. Box 1892, Houston, TX 77251, USA

Abstract

In this paper we consider the data distribution and data movement issues related to the solution of the basic linear algebra problems on high performance systems. The algorithms we discuss in details are the Gauss and Gauss-Jordan methods for solving a system of linear equations, the Cholesky's algorithm for LL^T-factorization, and QR-factorization algorithm using Householder transformations. It is shown that all those algorithms can be executed efficiently on a parallel system with simple and regular links and with partial pivoting. Detailed implementations of the algorithms are described using a simple parallel language on a systolic-type architecture. Both the theoretical analysis and the simulation results show speedups close to the optimal on moderately large problems.

*This work was partially supported by the NSF grant No. CCR-9409191 and by EPA grant R82-5207-01-0.

1 Introduction

In many scientific and practical computations the linear algebra algorithms are the most time consuming tasks. For example, the simulation of multiphase fluid flows in porous media and other computational fluid dynamics problems by the finite element method in 3-dimensions leads to systems of linear equations of hundreds of thousands of variables. A lot of researchers have concentrated their efforts on studying suitable parallel architectures and finding efficient implementations of the most important linear algebra algorithms [6].

In this paper we study the data distribution and data movement issues related to the efficient implementation of several basic linear algebra algorithms such as the Gauss and Gauss-Jordan eliminations, Cholesky's factorization, and QR-factorization algorithms. We consider versions of the algorithms that use partial pivoting and thus guarantee stability of the solution.

In order to provide detailed implementation and efficiency analysis, we use a concrete parallel architecture that is characterized with simple and regular links between the processors and systolic type of communications.

Our architecture includes a systolic array (SA) consisting of a 2-dimensional mesh of processors [8] as a basic computational unit and one general-purpose processing element (arithmetic-logic unit – ALU). ALU is intended to implement specific operations such as finding pivots and dividing in Gauss algorithm, square roots in Cholesky's algorithm, computing of reflection vectors in Householder's algorithm. A number of appropriately organized blocks of memory are included in the hardware in order to provide high utilization of SA and ALU.

Originally, the architecture was designed to implement efficiently several algorithms based on the Gauss elimination with partial pivoting such as LU-factorization, solving systems of linear equations, matrix inversions, etc., see [2, 3]. Consequently, it has been shown that other basic linear algebra problems as $L^T L$ factorization, and QR factorization, have efficient solutions on that architecture [1, 4].

Since our architecture is simpler than most of the commercially available parallel computers, our algorithms can be easily transformed into efficient implementations on other parallel computers, achieving close to optimal speedup. For instance, our algorithms can be efficiently implemented on networks with hypercube or torus type communication structures or on the different shared memory multiprocessors. Alternatively, our results will justify the design of a systolic-type co-processor that could dramatically increase the performance of an available sequential computer.

The paper is organized as follows. In Section 2, we describe the architecture of the proposed processor and an efficient realization of the matrix multiplication algorithm that will be used as a subroutine for the other algorithms. In Section 3, we describe a general scheme for deriving parallel linear algorithms on our architecture and introduce a language for description of the implementations. In Section 4, realizations of various linear algebra algorithms are described and analyzed.

2 Description of the Architecture

2.1 Basic Units and Interconnection Structure

In this subsection we describe the general architecture of the proposed systolic computer. Since our emphasis will be on the algorithms, our description will not contain low-level technical details. The main components of the computer are a systolic array (SA), an arithmetic-logic unit (ALU), 5 blocks memory which are appropriately sized and organized, and communication links (Figure 1). Below we give a detailed description of the characteristics and functions of each of these components. The computational and storage capacity of the computer is determined by two integer parameters. The parameter s denotes the size of the SA and the parameter S denotes one half of the storage of the basic memory of the computer.

• *Systolic array (SA).* It contains s^2 identical processing elements (PEs) connected in an $s \times s$ square mesh. These elements are sinchronized and operate under common control. PEs either perform computation or load data in their local registers. The only computational operation that PEs perform is $d = c + ba$, where b is the number currently stored in the local register of PE, and a and c are numbers entering its south and west ports respectively. In a single machine cycle the result d is computed and transmitted through the east port of the PE and a is transmitted through its north port, see Figure 2.

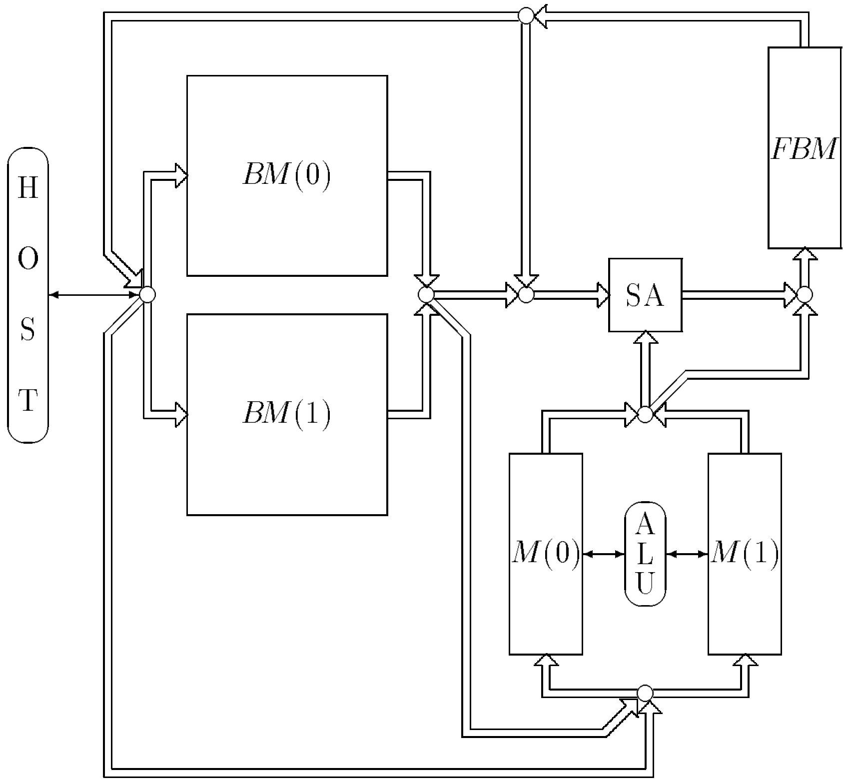

Figure 1. The architecture of the systolic computer consisting of a systolic array SA, a general purpose processor element ALU, 5 blocks memory and communication links.

The loading of data is implemented in the following fashion. For $s-1$ machine cycles each PE receives a number through its west port and transmits the previously received number through its east port. At the s-th machine cycle PEs receive the number b coming from west and stores b or $-b$ in its local register. Thus, given an $s \times s$ matrix B, in s consecutive machine cycles either B or $-B$ can be loaded into the local registers of the PEs of SA.

On this base, the SA performs one of the matrix operations

$$D := C + B\,A \quad \text{or} \quad D := C - B\,A, \tag{1}$$

where B is an $s \times s$ matrix, A, C, and D are $s \times L$ matrices, (see Figure 3).

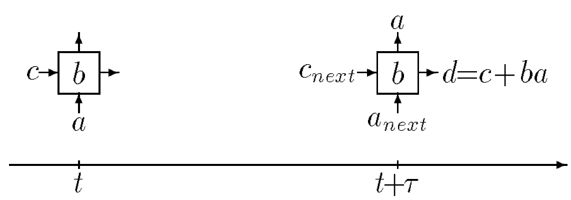

Figure 2. An illustration of the operation executed by the PEs of SA.

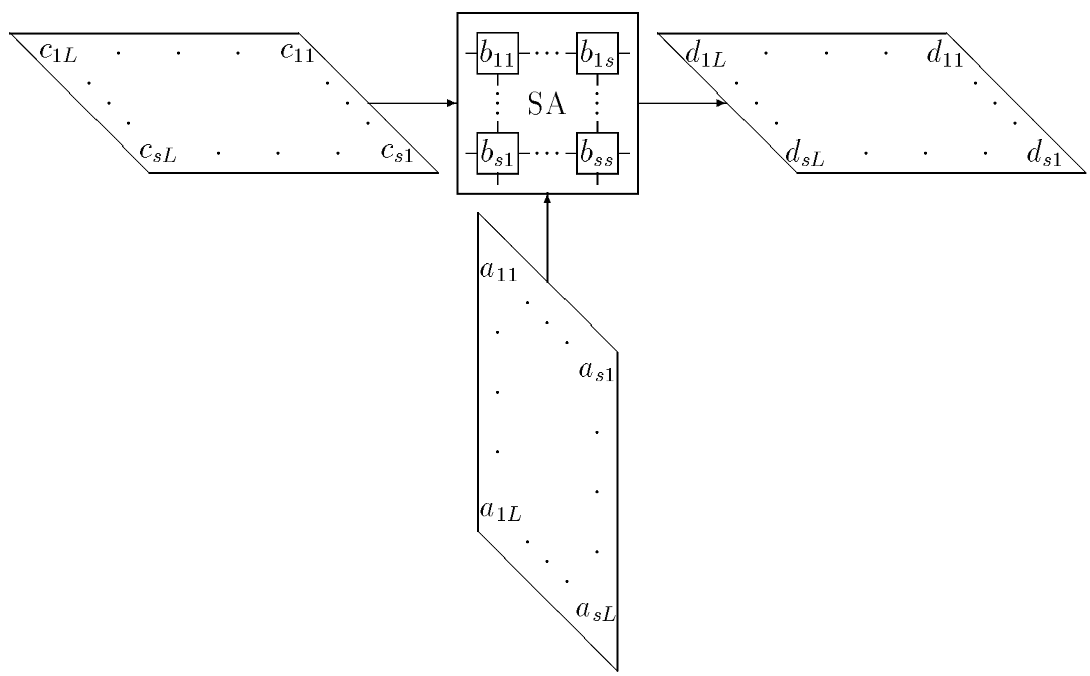

Figure 3. The arrangement and movement of the data during the execution of operation (1) on SA.

Assuming that B or $-B$ have been already loaded into SA the operation (1) is executed as follows. In $L + 2s$ machine cycles the PEs implement their only computational operation $d = c + ba$ and compute D. During that computation the elements of A enter the SA from the south ports, the elements of C enter from the west ports, and the elements of D leave SA from the east ports of SA. The movement of the data is fully synchronized and is as shown on the Figure 3.

• *Arithmetic-logic unit (ALU).* It is a general purpose processor element that is able to perform scalar and logic operations on data stored at its local registers. ALU is connected to the memories $M(0)$ and $M(1)$ and can access these memories only [Figure 1]. ALU works independently and is not synchronized with the other parts of the architecture. ALU is intended to implement irregular operations on data stored at $M(0)$ and $M(1)$. These are finding pivots, transposing, decomposing or inverting certain matrices of size s.

• *Basic memories $BM(0)$ and $BM(1)$.* Each of these memories has storage of S words. Each address of $BM(0)$ and $BM(1)$ contains a tuple of s ordered words (numbers) we call *packet*. In a single machine cycle a packet can be read from or written into an address. The basic memories are used as a storage for the input, intermediate results, and the output. The relation between the storage of the basic memories S and the maximum size N of a matrix that can be operated by the computer is given by $N^2/2 + Ns \leq S$.

• *Auxiliary memories $M(0)$ and $M(1)$.* They have storage $S_1 \approx s\sqrt{2S}$ words each, so that $S_1 \approx Ns + s^2$ and about N packets (one block-row or block-column) of the operated matrix can be stored at any of these memories. Their addresses will contain packets of s words too. In a single machine cycle a packet can be read from or written into each of the auxiliary memories. They are used as a storage of elements of matrices corresponding to the operand A from (1) and of elements to be processed by the ALU. Normally, while one of the auxiliary memories provides data for south ports of SA, the other supplies data for the ALU.

• *Fast buffer memory (FBM).* It has storage of S_1 words and is dual ported. Thus in a single machine cycle a packet (s words) can be read from and an other packet can be written into an address of *FBM*. It is intended to receive data from the east ports of SA and in the same time to be connected with either another memory unit or the west ports of SA.

• *Links between units.* Any link allows a transfer of a packet (s words) from one unit of the computer to another in a single machine cycle. Several pairs of units can be connected simultaneously if it does not contradict the topology of the structure presented at Figure 1.

Note that, in particular, SA and ALU can work in parallel, any of the basic or auxiliary memories can be connected to no more than one computational unit or another memory unit at the same time, and no direct transfer of data between $BM(0)$ and $BM(1)$ or between $M(0)$ and $M(1)$ is possible. Transfer from an auxiliary memory to a basic one, or from SA-east to a basic or auxiliary memory can be done through *FBM* with delay of one machine cycle only. The following connection pattern is a typical one: ALU connected with $M(0)$, $M(1)$ connected with SA-south, $BM(0)$ connected with SA-west, SA-east connected with *FBM* and *FBM* connected with $BM(1)$.

2.2 Example: matrix multiplication

In this subsection we show how matrices can be multiplied on our architecture. We employ an appropriate block version of the standard matrix multiplication algorithm. Next we demonstrate how multiplication can be implemented so that several units of our computer work together and achieve an optimal speed-up. Although matrix multiplication does not utilize all the features of the architecture (e.g., the ALU is not used), we consider that example first because of its simplicity and importance.

Throughout the paper we use the following notation. We denote the vector space of all N-by-M real matrices by $\mathbf{R}^{N\times M}$. For the sake of simplicity, we consider matrices whose sizes N and M are integral multiples of s, i. e. $N = ns$ and $M = ms$ for some integers n and m. By a_{ij}, for $i = 1, \ldots, N$, $j = 1, \ldots, M$, we denote the (i,j)-th entry of A, while A_{ij} for $i = 1, \ldots, n$, $j = 1, \ldots, m$ denotes the (i,j)-th block entry of order s of A. Thus

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1M} \\ \vdots & & \vdots \\ a_{N1} & \ldots & a_{NM} \end{pmatrix} = \begin{pmatrix} A_{11} & \ldots & A_{1m} \\ \vdots & & \vdots \\ A_{n1} & \ldots & A_{nm} \end{pmatrix}$$

and for $i = 1, \ldots, n,\ j = 1, \ldots, m$

$$A_{ij} = \begin{pmatrix} a_{(i-1)s+1\ (j-1)s+1} & \cdots & a_{(i-1)s+1\ js} \\ \vdots & & \vdots \\ a_{is\ (j-1)s+1} & \ldots & a_{is\ js} \end{pmatrix}.$$

We use $\vec{a}_i$, $i = 1, \ldots, N$ to denote the i-th row of A, i.e., $\vec{a}_i = (a_{i1}, \ldots, a_{iM})$, and similarly $\vec{A}_i = (A_{i1}, \ldots, A_{im})$, $i = 1, \ldots, n$, is the i-th block-row of A.

The tails of rows and block-rows are denoted by $\vec{a}_i[k:M] = (a_{ik}, \ldots, a_{iM})$ and $\vec{A}_i[k:m] = (A_{ik}, \ldots, A_{im})$. The j-th column and block-column are denoted respectively by $\vec{a}_{\downarrow j} = (a_{1j}, \ldots, a_{Nj})$, $j = 1, \ldots, M$, and by $\vec{A}_{\downarrow j} = (A_{1j}, \ldots, A_{nj})$, $j = 1, \ldots, m$. Correspondingly, $\vec{a}_{\downarrow j}[k:N] = (a_{kj}, \ldots, a_{Nj})$ and $\vec{A}_{\downarrow j}[k:n] = (A_{kj}, \ldots, A_{nj})$.

Let matrices $A \in \mathbf{R}^{N\times L}$ and $B \in \mathbf{R}^{L\times M}$ are given. We shall compute the matrix $C = AB$ employing a block version of the matrix-matrix multiplication algorithm that is derived from the identity

$$\vec{C}_i = \sum_{j=1}^{l} A_{ij}\vec{B}_j, \qquad i = 1, \ldots, n,$$

where $N = ns$, $M = ms$, and $L = ls$.

Algorithm 2.1: *Block Matrix Multiplication*

Input: $A \in \mathbf{R}^{N\times L}$; $B \in \mathbf{R}^{L\times M}$
Output: $C \in \mathbf{R}^{N\times M}$ such that $C = AB$

for $i = 1, \ldots, n$
$\vec{C}_i^{(1)} := A_{i1}\vec{B}_1$
  for $j = 2, \ldots, l$
  $\vec{C}_i^{(j)} := \vec{C}_i^{(j-1)} + A_{ij}\vec{B}_j$
  end j
end i

Note that all computations in the above algorithm are of the type of the operation 1. Initially, the matrices A and B are stored in $BM(0)$ and $BM(1)$ respectively. Block-rows of the result $\vec{C}_i$, $i = 1, \ldots, n$, are computed consecutively and overwrite $\vec{A}_i$ in $BM(0)$. The computation of each of the block-rows $\vec{C}_i$ with $1 < i < n$ is implemented in the manner described below in an algorithmic form. Suppose that, before the computation of $\vec{C}_i, 1 < i \leq n$, the allocation of the data is as shown on Figure 4.

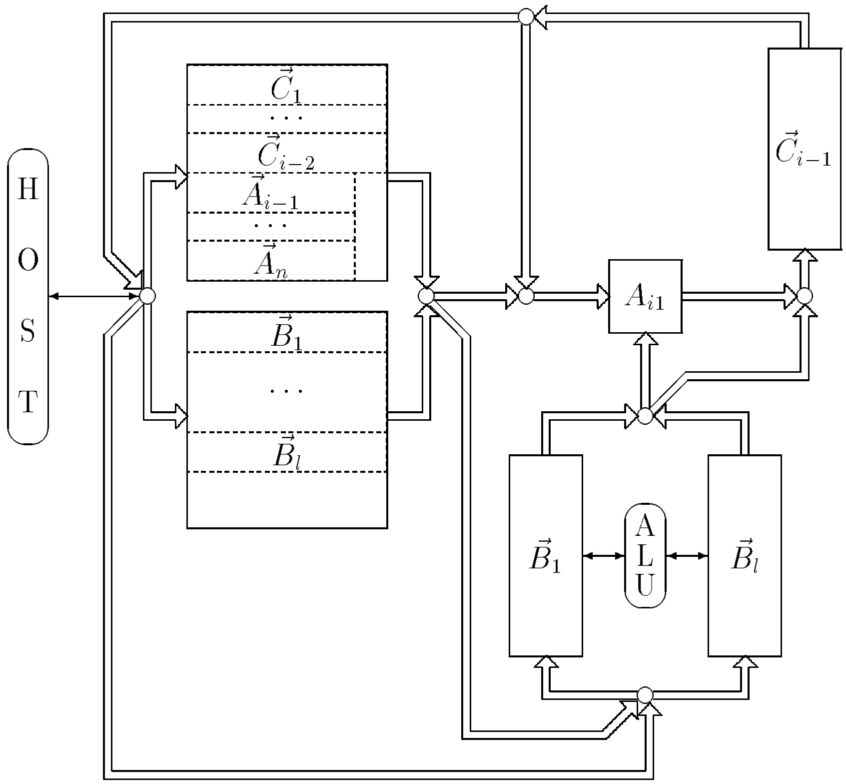

Figure 4. Allocation of the data before computing of $\vec{C}_i$.

ALGORITHM 2.2: *Computation of* $\vec{C}_i$
Input : The data shown on Figure 4
Output : The data allocated as is shown on Figure 4 with i increased by one

1. **do** steps 1.1 – 1.3 **in parallel**
   1.1. Compute $\vec{C}_i^{(1)} = A_{i1}\vec{B}_1$ on SA and store the result in *FBM*;
   Load A_{i2} into local registers of PEs forming SA
   1.2. Load $\vec{B}_2$ into this of the memories $M(1)$ and $M(0)$ that does not contain $\vec{B}_1$
   1.3. Transfer $\vec{C}_{i-1}^{(l)}$ from *FBM* into $BM(0)$ overwriting $\vec{A}_{i-1}$
2. **for** $j = 2, \ldots, l$ **do** steps 2.1 and 2.2 **in parallel**

2.1. Compute $\vec{C}_i^{(j)} = \vec{C}_i^{(j-1)} + A_{ij}\vec{B}_j$ on SA
(note that we use the feature of *FBM* to be connected with both ports SA-west and SA-east);
Load $A_{i,j+1}$ ($A_{i+1,1}$ if $j = l$) into local registers of SA

2.2. Load $\vec{B}_{j+1}$ ($\vec{B}_1$ if $j = l$) into this of the memories $M(0)$ and $M(1)$ that does not contain $\vec{B}_j$

During the first stage of that algorithm SA performs operation (1) with C-operand zero (see Figure 3), *BM*(1) sends data to this of the memories $M(0)$ and $M(1)$ that is not involved in the operation (1), and *FBM* sends data to *BM*(0). In the second stage SA performs operation (1), where *FBM* receives results from SA-east ports and supplies data to SA-west ports. At the same time *BM*(1) transfers data to this of the small memories that is free of supplying data to the SA-south ports.

It is easily seen that after the computation of $\vec{C}_i$ by implementing Algorithm 2.2 the allocation of the data is the same as before the computation with i increased by one. Thus, after an initial step that computes $\vec{C}_1$ and arranges data as shown on Figure 4, $n-1$ consecutive executions of Algorithm 2.2 and a final step of transferring $\vec{C}_n$ to *BM*(0), we will have the matrix $C = AB$ stored in the memory *BM*(0). In the next section we give a detailed description (Algorithm 3.3) of the implementation of matrix multiplication on our architecture.

In order to address the efficiency of the matrix multiplication on our architecture we observe that steps (1.1) and (2.1) are time-dominant in the two stages of Algorithm 2.2 and thus SA works continuously during the consecutive executions of Algorithm 2.2. Thus, matrices can be multiplied on our architecture with asymptotically optimal speed-up of s^2.

3 Our Approach

In this section we describe our approach to implementing the linear algebra algorithms on the systolic computer. We will start by discussing some basic ideas and techniques and to introduce a convenient notation we call a *description language*, that makes possible a short but precise description of the implementations on the computer. As a first example we will describe the implementation of the matrix multiplication algorithm. We will conclude this section by presenting our general approach to evaluating the performance parameters of the implementations.

3.1 The Generic Structure of the Algorithms

Let $\mathcal{M}$ be a matrix algorithm of a general type described below as Algorithm 3.1. It can be viewed as an N-step transformation $A = A^{(0)}, A^{(1)}, \ldots, A^{(N)} = R$ of an input matrix A into an output matrix R. As we show in the next section, many important linear algebra algorithms are of this general type.

ALGORITHM 3.1: *A general type algorithm* $\mathcal{M}$
Input: $A \in \mathbf{R}^{N\times M}$
Output: $R \in \mathbf{R}^{N\times L}$
for $k = 1, \ldots, N$
Find r_k from $a_k^{(k-1)}$
    for $i = k+1, \ldots, N$
    $a_i^{(k)} := a_i^{(k-1)} + f_{ik} r_k$
    end i
end k.

By f_{ik} we denote specific numbers that depend on $\mathcal{M}$ that are already computed and available, as for example certain elements of $A^{(k-1)}$. It will be shown in the next section that many important linear algebra algorithms can be represented in the form of $\mathcal{M}$.

As was described in Section 2, our architecture performs well on code that is rich in operations of the type (1). Therefore, our first objective is to find a block version $\mathcal{M}^b$ of $\mathcal{M}$ containing primarily this type of operations. Formally, $\mathcal{M}^b$ can be obtained from $\mathcal{M}$ by replacing the small letters a, f, and r (matrix elements) by capital letters A, F, and R (submatrices) and N by n. Let us recall that $\vec{A}_i$, $\vec{R}_i$ denote block-row F_{ik} is a block of order s.

Next we propose a general scheme for implementation of $\mathcal{M}^b$ on the architecture described in Section 2. Note that by a rearrangement of the computations in $\mathcal{M}^b$ we make the basic operations of type (1) on SA and the specific operations for finding R_k on ALU to be implemented in parallel.

ALGORITHM 3.2: *A general scheme for implementation of* $\mathcal{M}^b$

Input: $A \in \mathbf{R}^{ns \times ms}$

Output: $R \in \mathbf{R}^{ns \times ls}$

(A) **ALU**: Find $\vec{R}_1$ from $\vec{A}_1$ using algorithm $\mathcal{M}$

(B) **for** $k = 2, \ldots, n$

(1) **SA**: $\vec{A}_k^{(k-1)} := \vec{A}_k^{(k-2)} + F_{k,k-1}\vec{R}_{k-1}$

(2) **do** steps 2.1 and 2.2 **in parallel**

  (2.1) **ALU**: Find $\vec{R}_k$ from $\vec{A}_k^{(k-1)}$ using algorithm $\mathcal{M}$

  (2.2) **for** $i = k+1, \ldots, n$

    SA: $\vec{A}_i^{(k-1)} := \vec{A}_i^{(k-2)} + F_{i,k-1}\vec{R}_{k-1}$

    end i

  end k.

The $s \times s$ blocks F_{ik} are usually blocks of (or determined from) $A^{(k-1)}$ depending on $\mathcal{M}^b$. The implementations presented in the next section, including Gauss-elimination type algorithms, Cholesky and Householder factorizations are derived from this general scheme. Alternative schemes are also possible and competitive [1].

Let us describe the arrangement of the data during the implementation of $\mathcal{M}^b$. Assume that the basic memories are divided into equal parts capable of storing one block-row $\vec{A}_i$ of A. Denote by S_1, $S_3, \ldots$ parts of $BM(0)$ and by S_2, $S_4, \ldots$ parts of $BM(1)$. During the input phase of Algorithm 3.2, $\vec{A}_i^{(0)} = \vec{A}_i$, $i = 1, \ldots, n$, is stored into S_i and during the implementation $\vec{A}_i^{(k)}$ is stored into S_{i-k} (where we assume $S_{-j} = S_{n-j}$ for $j = 0, \ldots, n-1$). Note that $\vec{A}_i^{(0)}$, $\vec{A}_i^{(1)}, \ldots$ are alternatively stored in $BM(i \bmod 2)$ and $BM((i+1) \bmod 2)$, which provides the efficient execution of the steps 1 and 2.2 on SA. While one of the memories BM supplies data to the SA-west ports, the other is connected to FBM and receives the result of the computation. After the iteration that computes $A^{(k-1)}$ for $k = 2, \ldots, n$ the block-row $\vec{R}_{k-1}$ is stored in S_{n-k+2} (or in S_{k-1} for Gauss-Jordan algorithm). At the end of step (B) the last block-row of the result $\vec{R}_n$ is already computed in (2.1) and resides in one of the memories $M(0)$ or $M(1)$.

3.2 Description Language for Implementations

To make our description precise and short we introduce a Description Language (DL). Except self-explaining operators, it contains procedures of the following 3 types:

• *SA-procedure.* Let M, M', M'', and M''' denote memories of our architecture that satisfy the following conditions: M' is a memory connected to SA-south ports, i.e., one of the auxiliary memories, M''' is a memory connected to SA-west ports, that is either a basic memory or *FBM*, and M is either different from M' and M''' or $M = M''' = FBM$. Let $\vec{A}$, $\vec{C}$, $\vec{D} \in \mathbf{R}^{s\times L}$ be matrices – block-rows, and $B \in \mathbf{R}^{s\times s}$ be a $s \times s$ block. Suppose that $\vec{A}$, B, and $\vec{C}$ are currently stored in memories M', M'', M''' respectively. Then by

$$\mathbf{SA}: \ M[\vec{D}] := M'''[\vec{C}] + M''[B] \ M'[\vec{A}]$$

we denote the following sequence of operations on the architecture:

1) Transfer the matrix B from memory M'' into the local registers of PEs forming SA;

2) Perform the operation (1) on the SA storing the result $\vec{D}$ in M (if $M \neq FBM$ then $\vec{D}$ is transfered through the dual ported memory *FBM*) with delay of one machine cycle only. When $\vec{C}$ is zero matrix, the term $M'''[\vec{C}]$ will be omitted.

• *ALU-procedure.* Let M be an auxiliary memory, and $\vec{A} \in \mathbf{R}^{s\times L_1}$, $\vec{R} \in \mathbf{R}^{s\times L_2}$. By

$$\mathbf{ALU}: \ M\ [\vec{A}] \to [\vec{R}]$$

we denote the computation of $\vec{R}$ from $\vec{A}$ on the ALU and its storing at the place of $\vec{A}$ in M.

• *Data transfer.* By

$$M'[\vec{A}] \longrightarrow M''$$

we denote a transfer of $\vec{A}$ from M' to M'', where M' and M'' are memories and $\vec{A} \in \mathbf{R}^{s\times L}$. If direct transfer is not allowed, $\vec{A}$ is transported through *FBM*, that is denoted by

$$M'[\vec{A}] \stackrel{via\ FBM}{\longrightarrow} M''.$$

• *Memory notations* We shall use notations $BM(k)$ and $M(k)$ where k is an integer denoting $BM(0)$ and $M(0)$ if k is even and $BM(1)$ and $M(1)$ otherwise, i. e.

$$BM(k) = BM(k \bmod 2) \quad \text{and} \quad M(k) = M(k \bmod 2).$$

Next we present a formal description of implementation of matrix-matrix multiplication algorithm on our architecture in terms of DL (cf. with Algorithm 2.2).

ALGORITHM 3.3: *Implementation of Matrix-Matrix Multiplication*
Input: $A \in \mathbf{R}^{ns\times ls}$ stored in $BM(0)$; $B \in \mathbf{R}^{ls\times ms}$ stored in $BM(1)$
Output: $C \in \mathbf{R}^{ns\times ms}$ stored in $BM(0)$ such that $C = AB$
(A) $BM(1)[B_1] \longrightarrow M(0)$
(B) **for** $i = 1, \ldots, n$
(1) **do** steps 1.1–1.3 **in parallel**
    (1.1) **SA**: $FBM[\vec{C}_i^{(1)}] := BM(0)[A_{i1}]\ M((i-1)l)[\vec{B}_1]$
    (1.2) $BM(1)[\vec{B}_2] \longrightarrow M((i-1)l+1)$
    (1.3) **if** $i > 1$ **then** $FBM[\vec{C}_{i-1}^{(l)}] \longrightarrow BM(0)$
(2) **for** $j = 2, \ldots, l$ **do** steps 2.1 and 2.2 **in parallel**
    (2.1) **SA**: $FBM[\vec{C}_i^{(j)}] := FBM[\vec{C}_i^{(j-1)}] + BM(0)[A_{ij}]\ M((i-1)l+j+1)[\vec{B}_j]$
    (2.2) $BM(1)[\vec{B}_{j \bmod l+1}] \longrightarrow M((i-1)l+j))$
    end j
end i
(C) $FBM[C_n^{(l)}] \longrightarrow BM(0)$.

3.3 Evaluation of the Efficiency of an Implementation

In this subsection we describe our approach to estimating the efficiency of the implementations. Generally, for a given implementation, we are going to evaluate the number of machine cycles needed for its execution on the computer and to compare that number to the number of *flops* (floating point operations, see [7]) in the original sequential algorithm. Following the widely accepted terminology, we are going to talk about running times instead number of cycles, operations, or flops. We assume, that one machine cycle on the computer takes one unit of time and an operation on the ALU takes α

units. Further, we assume that one flop of the sequential algorithm takes one unit of time. Thus we can compute or estimate the execution time of the considered implementation and to compare it to the "sequential" time, which is the number of flops in the corresponding sequential algorithm. The ratio between the sequential time of an algorithm and its implementation on the architecture is called *speed-up* of that implementation and is denoted by $\mathcal{S}$. The best possible speed-up of an implementation is $2s^2 + 1/\alpha$, since at most $2s^2 + 1/\alpha$ flops per unit of time can be performed on the computer. The *efficiency* $\mathcal{E}$ of an implementation is defined as the ratio between its speed-up and $2s^2 + 1/\alpha$.

Next we illustrate our approach to estimate the speed-up of our implementations by formally applying the general scheme described above.

Denote the sequential time of Algorithm 3.1 by T_{SEQ} and the time for the execution of Algorithm 3.2 on the architecture by T_{PAR}. Denote also the time for Step A of Algorithm 3.2 by T^1_{ALU}. The k-th iteration of the loop in Step B of Algorithm 3.2 consists of two parts. First is the operation (1) of SA and we denote the time for that operation by $T^k_{0,SA}$. Next is the parallel operation (2) that takes the maximum of the time T^k_{SA} for Step 2.1 executed on SA and the time T^k_{ALU} for Step 2.2 executed on ALU.

According to our notation, the time for execution of Algorithm 3.2 is

$$T_{PAR} = T^1_{ALU} + \sum_{k=2}^{n} \left(T^k_{0,SA} + \max \{ T^k_{ALU}, T^k_{SA} \} \right). \tag{2}$$

It is shown in the next section that for a wide range of linear algebra algorithms the general scheme applies and for sufficiently large inputs there exists a value k_0 so that $T^k_{ALU} \leq T^k_{SA}$ for $k \leq k_0$ and $T^k_{ALU} > T^k_{SA}$ for $k > k_0$. Therefore, for the running time of their implementations it holds

$$T_{PAR} = T^1_{ALU} + \sum_{k=2}^{n} T^k_{0,SA} + \sum_{k=2}^{k_0} T^k_{SA} + \sum_{k=k_0+1}^{n} T^k_{ALU}. \tag{3}$$

In general, when the sizes M and N of the input increase, then k_0 tends to $n = N/s$ and the term $\sum_{k=2}^{k_0} T^k_{SA}$ asymptotically dominates in (3). On the other hand

$$\sum_{k=2}^{n} T^k_{SA} \leq \frac{T_{SEQ}}{2s^2},$$

since SA performs $2s^2$ flops included in the sequential algorithm per unit of time. Thus, for the speed-up and the efficiency of implementations of considered linear algebra algorithms on our architecture we obtain

$$\begin{aligned}\mathcal{S} &= \frac{T_{SEQ}}{T_{PAR}} \geq 2s^2(1-o(1)) \\ \mathcal{E} &= \frac{\mathcal{S}}{2s^2+1/\alpha} \geq 1 - \frac{1}{1+2s^2\alpha} - o(1).\end{aligned}$$

Therefore, under the condition that the term $\sum_{k=2}^{k_0} T_{SA}^k$ dominates in (3), the speed-up of our general implementation is close to $2s^2$, which is the maximum that can be achieved on our systolic array. Consequently, the efficiency is quite close to the optimal, e.g., if $s = 4$ and $\alpha = 2$, then $\mathcal{E}$ is close to $64/65 \approx 0.985$.

4 Algorithms

In this section we present block versions of five basic matrix algorithms and their implementations on our architecture. The first three of them are connected to the problem of solving a system of linear equations (SLE) with general type matrix and multiple right hand sides. We present first an implementation of the Forward Substitution algorithm to solve SLE with a triangular matrix. Next we present an implementation of LU-factorization based on the Gauss elimination method with partial pivoting. Combining these two algorithms we can solve SLE with multiple right hand sides on the systolic computer with high efficiency. As an alternative, that is of its own interest, we present an implementation of the Gauss–Jordan algorithm with partial pivoting for matrix inversion and for solving SLE.

Our fourth example shows that SLE with symmetric and positively definite matrices can be solved on the systolic computer efficiently. It is an implementation of the R^TR-factorization of a symmetric and positively definite matrix based on the Cholesky algorithm.

The fifth example concerns orthogonalization. We present an efficient implementation of the Householder QR-factorization algorithm for a rectangular full-rank matrix. Notably, the implementation was obtained by the aid of a modification of a recently discovered in [5, 9] block version of Householder factorization, that is consistent with our general scheme.

We describe each of the algorithms and implementations in a separate subsection. Our presentation follows closely the framework and notations from the previous section. We begin with a short formulation of the considered linear algebra problem. We give a version of a traditional sequential algorithm consistent with Algorithm 3.1. Next we discuss the obtaining of a block algorithm of the type of Algorithm 3.2 and comment the execution of Step 2.1 on ALU and of the block updates in Step 2.2 on SA. Then, we present a formal description of an implementation of this algorithm on our architecture using the description language described in Subsection 3.2. At the end we estimate the running time of the implementation and evaluate its speed-up and efficiency.

4.1 Solving Triangular Systems of Linear Equations

An important linear algebra problem is solving of a system of linear equations $Ax = b$, where $A \in \mathbf{R}^{N \times N}$ and $x, b \in \mathbf{R}^{N \times 1}$. A classic factorization method consists of representing the matrix A as a product of two triangular matrices and then solving appropriate systems with these triangular matrices.

We consider triangular SLE with multiple right hand sides. Let $L \in \mathbf{R}^{N \times N}$ be a nonsingular lower triangular and $B \in \mathbf{R}^{N \times M}$. In this subsection we present an algorithm and its implementation for finding a matrix $X \in \mathbf{R}^{N \times M}$ that solves the matrix equation $LX = B$. This problem is equivalent to solving of a number of systems of linear equations with the same triangular left-hand matrix.

The sequential algorithm which we use is the well known Forward Substitution. The consistency of this algorithm to Algorithm 3.1 is obvious.

ALGORITHM 4.1.1 : *Forward Substitution*
Input : $L \in \mathbf{R}^{N \times N}$ — nonsingular lower triangular; $B \in \mathbf{R}^{N \times M}$
Output : $X \in \mathbf{R}^{N \times M}$ such that $LX = B$

for $k = 1, \ldots, N$
$\vec{x}_k := l_{kk}^{-1}\, \vec{b}_k^{(k-1)}$
    for $i = k+1, \ldots, N$
    $\vec{b}_i^{(k)} := \vec{b}_i^{(k-1)} - l_{ik} \vec{x}_k$
    end i
end k.

Let us remind that the notation $\vec{a}_i$ denotes the i-th row of a given matrix A, see Subsection 2.2. The execution of the Forward Substitution takes approximately MN^2 flops.

We derive a formal block version of this algorithm be replacing small letters with capitals and rearrange it according to Algorithm 3.2.

ALGORITHM 4.1.2: *Rearranged block version of Forward Substitution*
Input : $L \in \mathbf{R}^{ns\times ns}$ — nonsingular lower triangular; $B \in \mathbf{R}^{ns\times ms}$
Output : $X \in \mathbf{R}^{ns\times ms}$ such that $LX = B$
(A) **ALU**: Find $\vec{X}_1$ from $\vec{L}_{11}$ and $\vec{B}_1$
(B) **for** $k = 2, \ldots, n$
(1) **SA**: $\vec{B}_k^{(k-1)} := \vec{B}_k^{(k-2)} + L_{k,k-1}\vec{X}_{k-1}$
(2) **do** steps 2.1 and 2.2 **in parallel**
  (2.1) **ALU**: Find $\vec{X}_k$ from L_{kk} and $\vec{B}_k^{(k-1)}$
  (2.2) **for** $i = k+1, \ldots, n$
    SA: $\vec{B}_i^{(k-1)} := \vec{B}_i^{(k-2)} + L_{i,k-1}\vec{X}_{k-1}$
    end i
  end k.

For the execution of Step A and Step 2.1 ALU uses the Forward Substitution (Algorithm 4.1.1) with $N = s$. The block updates in Step 2.2 are of the type of the operation (1) and will be executed on the SA. Note that Algorithm 4.1.2 is computationally equivalent to Algorithm 4.1.2. Below we present a detailed implementation of the Forward Substitution using our description language.

ALGORITHM 4.1.3: *Implementation of Forward Substitution*
Input : $[\vec{L}_i, \vec{B}_i]$ stored in $BM(i+1)$ $i = 1, \ldots, n$
Output : $\vec{X}_i$ stored in $BM(n-i)$, $i = 1, \ldots, n$
(A) $BM(0)[L_{11}, \vec{B}_1] \longrightarrow M(0)$
  ALU: $M(0)$ $[L_{11}, \vec{B}_1] \to [\vec{X}_1]$
(B) **for** $k = 2, \ldots, n$
(1) $BM(k+1)[L_{kk}] \longrightarrow M(k+1)$
  SA: $M(k+1)[\vec{B}_k^{(k-1)}] := BM(1)[\vec{B}_k^{(k-2)}] - BM(k+1)[L_{k,k-1}]\ M(k)[\vec{X}_{k-1}]$
(2) **do** steps 2.1 and 2.2 **in parallel**
  (2.1) **ALU**: $M(k+1)$ $[L_{kk}, \vec{B}_k^{(k-1)}] \to [\vec{X}_k]$

(2.2) **for** $i = k+1, \ldots, n$

SA: $BM(i+k)[\vec{B}_i^{(k-1)}] := BM(i+k+1)[\vec{B}_i^{(k-2)}] -$
$BM(i+1)[L_{i,k-1}]\ M(k)[\vec{X}_{k-1}]$

end i

(3) $M(k)[\vec{X}_{k-1}] \stackrel{via\ FBM}{\longrightarrow} BM(n-k+1)$

end k

(C) $M(n+1)[\vec{X}_n] \stackrel{via\ FBM}{\longrightarrow} BM(0)$.

Let us now estimate the running time of the presented implementation and to evaluate its speed-up and efficiency. Following the scheme from Subsection 3.3 we denote T_{ALU}^k the running time of the ALU-procedure in Step A for $k = 1$ and in Step 2.1 for $k = 2, \ldots, n$. The running time of the SA-procedure in Step 1 is denoted by $T_{0,SA}^k$ and by T_{SA}^k the total time of $n-k$ SA-procedures in Step 2.2. Thus the running time of the above implementation is

$$T_{PAR} = T_{ALU}^1 + \sum_{k=2}^{n}(T_{0,SA}^k + \max(T_{ALU}^k, T_{SA}^k)) + T_{COM},$$

where T_{COM} denotes the total time for data transfer. The leading terms of these times are easily computed and are as follows:

$$T_{ALU}^k \approx ms^3\alpha, \qquad T_{0,SA}^k \approx ms, \qquad T_{SA}^k \approx (n-k)ms, \qquad T_{COM} \approx nms.$$

The maximum

$$\max(T_{ALU}^k, T_{SA}^k) = \begin{cases} T_{SA}^k & \text{for} \quad k \le k_0 \\ T_{ALU}^k & \text{for} \quad k \ge k_0, \end{cases}$$

where k_0 is computed approximately from the equation $T_{ALU}^{k_0} = T_{SA}^{k_0}$ and is $k_0 = \max(0, \lfloor n - s^2\alpha \rfloor)$. Finally for the running time of the implementation of the Forward Substitution we obtain

$$\begin{aligned} T_{PAR} &\approx ms^3\alpha + mns + \sum_{k=2}^{n} ms + \sum_{k=2}^{k_0}(n-k)ms + \sum_{k=k_0}^{n} ms^3\alpha \\ &\approx \begin{cases} \frac{mn^2s}{2}(1 + (s^2\alpha/n)^2) & \text{for} \quad s^2\alpha \le n \\ mns^3\alpha & \text{for} \quad s^2\alpha \ge n., \end{cases} \end{aligned}$$

For the speed-up and efficiency of the implementation of the Forward Substitution, the following approximations are valid

$$\mathcal{S} = \frac{T_{SEQ}}{T_{PAR}} \approx \begin{cases} \frac{2s^2}{(1+(s^3\alpha/N)^2)} & \text{for} \quad s^3\alpha \leq N \\ \\ \frac{N}{s\alpha} & \text{for} \quad s^3\alpha \geq N, \end{cases}$$

$$\mathcal{E} = \frac{\mathcal{S}}{2s^2+1/\alpha} \approx \begin{cases} \frac{2s^2\alpha}{(1+2s^2\alpha)(1+(s^3\alpha/N)^2)} & \text{for} \quad s^3\alpha \leq N \\ \\ \frac{N}{s(1+2s^2\alpha)} & \text{for} \quad s^3\alpha \geq N, \end{cases}$$

Clearly the speed-up of Forward Substitution approximates $2s^2$, which is the optimal for our systolic array and the efficiency is close to one.

4.2 Gauss-Jordan Transformations – Matrix "Division"

Let $A \in \mathbf{R}^{N\times N}$ be a nonsingular matrix and $B \in \mathbf{R}^{N\times M}$. In this subsection we present an algorithm and its implementation for finding the product $A^{-1}B$. Note that this problem is rather general and can be readily applied for solving SLE $AX = B$ with multiple right hand sides since $X = A^{-1}B$ and for finding the inverse of A when B is an unit matrix. We employ a parallel version of well-known Gauss-Jordan algorithm with row pivoting. Essentially, this algorithm computes a sequence of pairs of matrices

$$(A, B) = (A^{(0)}, B^{(0)}), (A^{(1)}, B^{(1)}), \ldots, (A^{(N)}, B^{(N)}),$$

where $A^{(k)} \in \mathbf{R}^{N\times(N-k)}$, $B^{(k)} \in \mathbf{R}^{N\times M}$, $k = 1, \ldots, N$. The product $A^{-1}B$ is found as a row permutation of the matrix $B^{(N)}$. Extended matrices $(A^{(k)}, B^{(k)})$ are computed by the Gauss-Jordan transformations as presented below in form consistent with our general scheme (Algorithm 3.1).

Algorithm 4.3.1 : *Gauss-Jordan Transformations with Row Pivoting*
Input : $A \in \mathbf{R}^{N\times N}$, $B \in \mathbf{R}^{N\times M}$
Output : $B^{(N)} \in \mathbf{R}^{N\times M}$ and a permutation vector $J = (j_1, \ldots, j_N)$
for $k = 1, \ldots, N$
Determine j_k so that $|a^{(k-1)}_{kj_k}| = \max_{k\leq j\leq N} |a^{(k-1)}_{kj}|$
$(\vec{a}^{(k)}_k[k{+}1\!:\!N], \vec{b}^{(k)}_k) := (a^{(k-1)}_{kj_k})^{-1} (\vec{a'}^{(k-1)}_k[k\!:\!N], \vec{b}^{(k-1)}_k)$
    for $i = 1, \ldots, N, \quad i \neq k$
    $(\vec{a}^{(k)}_i[k{+}1\!:\!N], \vec{b}^{(k)}_i) := (\vec{a'}^{(k-1)}_i[k\!:\!N], \vec{b}^{(k-1)}_i) - a^{(k-1)}_{ij_k} (\vec{a}^{(k)}_k[k{+}1\!:\!N], \vec{b}^{(k)}_k)$

end i

end k.

The execution of this algorithm requires approximately $N^3 + 2MN^2$ flops. The notation $\vec{a'}_i^{(k-1)}[k:N]$ means the vector $\vec{a}_i^{(k-1)}[k:N]$ with discarded entry a_{ij_k}. The result $A^{-1}B$ is a row permutation of $B^{(N)}$ defined by the vector $J = (j_1, \ldots, j_N)$.

The block version of the above algorithm that corresponds to Algorithm 3.2 is obtained directly following our scheme.

ALGORITHM 4.3.2: *Block Version of Gauss-Jordan Algorithm*

Input: $A \in \mathbf{R}^{ns\times ns}$, $B \in \mathbf{R}^{ns\times ms}$

Output: $B^{(ns)} \in \mathbf{R}^{ns\times ms}$ and a permutation vector $J = (j_1, \ldots, j_{ns})$

(A) **ALU**: Compute $(\vec{A}_1^{(1)}[2:n], \vec{B}_1^{(1)})$, and $J_1 = (j_1, \ldots, j_s)$
from $(\vec{A}_1^{(0)}[1:n], \vec{B}_1^{(0)})$ by using Algorithm 4.3.1

(B) **for** $k = 2, \ldots, n$

(1) **SA**: $(\vec{A}_k^{(k-1)}[k:n], \vec{B}_k^{(k-1)}) := (\vec{A'}_k^{(k-2)}[k-1:n], \vec{B}_k^{(k-2)}) - A_{kJ_{k-1}}^{(k-2)} (\vec{A}_{k-1}^{(k-1)}[k:n], \vec{B}_{k-1}^{(k-1)})$

(2) **do** steps 2.1 and 2.2 **in parallel**

(2.1) **ALU**: Compute $(\vec{A}_k^{(k)}[k+1:n], \vec{B}_k^{(k)})$ and $J_k = (j_{(k-1)s+1}, \ldots, j_{ks})$
from $\vec{A}_k^{(k-1)}[k:n], \vec{B}_k^{(k-1)})$ by using Algorithm 4.3.1

(2.2) **for** $i = 1, \ldots, n, \quad i \neq k-1, k$

SA: $(\vec{A}_i^{(k-1)}[k:n], \vec{(B)}_i^{(k-1)} := \vec{A'}_i^{(k-2)}[k-1:n], \vec{B}_i^{(k-2)} - A_{iJ_{k-1}}^{(k-2)} (\vec{A}_{k-1}^{(k-1)}[k:n], \vec{B}_{k-1}^{(k-1)})$

end i

end k

(C) **for** $i = 1, \ldots, n-1$

SA: $\vec{B}_i^{(n)} := \vec{B}_i^{(n-1)} - A_{iJ_n}^{(n-1)} \vec{B}_n^{(n)}$

end i.

ALU performs Step A and Step 2.1 using Algorithm 4.2.1 with s instead of N in the upper bounds in the two loops. We denoted by $J_k = (j_{(k-1)s+1}, \ldots, j_{ks})$, $k = 1, \ldots, n$, the corresponding subvectors of J.
By $\vec{A'}_i^{(k-1)}[k:n]$ we denoted the block-row matrix $A_i^{(k-1)}[k:n]$ with removed

columns with numbers in J_k. These removed columns constitute an $s \times s$ matrix A_{iJ_k} which plays the role of the pivot element in the sequential algorithm. The block updates in Steps 1 and 2.2 are of the type of the systolic operation (1) and are executed on SA.

Below we give a detailed formal description of the implementation of Gauss-Jordan algorithm. Note that the permutation subvectors J_k, $k = 1, \dots, n$, of the k-th execution of Step 2.1, are part of the results. They can be used afterwards to output the true result $A^{-1}B$.

ALGORITHM 4.3.3: *Implementation of Gauss-Jordan Algorithm*

Input: $[\vec{A}_i, \vec{B}_i]$ stored in $BM(i+1)$, $i = 1, \dots, n$

Output: $[B_i^{(n)}; J_i]$ stored in $BM(i+1)$, $i = 1, \dots, n$

(A) $BM(0)[\vec{A}_1^{(0)}[1:n], \vec{B}_1^{(0)}] \longrightarrow M(0)$

ALU: $M(0)\ \ [\vec{A}_1^{(0)}[1:n], \vec{B}_1^{(0)}] \to [\vec{A}_1^{(1)}[2:n], \vec{B}_1^{(1)}; J_1]$

(B) **for** $k = 2, \dots, n$

(1) **SA**: $M(k+1)[\vec{A}_k^{(k-1)}[k:n], \vec{B}_k^{(k-1)}] := BM(1)[\vec{A'}_k^{(k-2)}[k-1:n], \vec{B}_k^{(k-2)}] -$
$BM(1)[A_{kJ_{k-1}}^{(k-2)}]\ M(k)[\vec{A}_{k-1}^{(k-1)}[k:n], \vec{B}_{k-1}^{(k-1)}]$

(2) **do** steps 2.1 and 2.2 **in parallel**

(2.1) **ALU**: $M(k+1)\ \ [\vec{A}_k^{(k-1)}[k:n], \vec{B}_k^{(k-1)}] \to [\vec{A}_k^{(k)}[k+1:n], \vec{B}_k^{(k)}; J_k]$

(2.2) **for** $i = 1, \dots, n, \quad i \neq k-1, k$

SA: $BM(i+k)[\vec{A}_i^{(k-1)}[k:n], \vec{B}_i^{(k-1)}] := BM(i+k+1)[\vec{A'}_i^{(k-2)}[k-1:n], \vec{B}_i^{(k-2)}] -$
$BM(i+k+1)[A_{iJ_{k-1}}^{(k-2)}]\ M(k)[\vec{A}_{k-1}^{(k-1)}[k:n], \vec{B}_{k-1}^{(k-1)}]$

end i

(3) $M(k)[\vec{A}_{k-1}^{(k-1)}[k:n], \vec{B}_{k-1}^{(k-1)}; J_{k-1}] \overset{via\ FBM}{\longrightarrow} BM(n+1)$

end k

(C) **for** $i = 1, \dots, n-1$

SA: $BM(i+n+1)[\vec{B}_i^{(n)}] := BM(i+n)[\vec{B}_i^{(n-1)}] -$
$BM(i+n)[A_{iJ_n}^{(n-1)}]\ M(n+1)[\vec{B}_n^{(n)}]$

end i

$M(n+1)[\vec{B}_n^{(n)}; J_n] \overset{via\ FBM}{\longrightarrow} BM(n+1)$.

To evaluate the running time of this implementation we use the following notations: T_{ALU}^k, for $k = 0, \dots, n-1$ denotes the execution time for the ALU operation in Step A for $k = 0$ and in Step 2.1 for $k = 1, \dots n-1$; $T_{0,SA}^k$

for $k = 2, \ldots, n$ denotes the time for the SA operation in Step 1; T_{SA}^k for $k = 2, \ldots, n$ denotes the execution time of Step 2.2; $T_{n,SA}$ denotes the time for the execution of the loop in Step C; and T_{COM} denotes the total time for data transfer in first and last line of the implementation and in Step 3. When n is big enough, the approximate values for these times are

$$T_{ALU}^k \approx 2(n+m-k)s^3\alpha, \quad T_{0,SA}^k \approx (n+m-k)s,$$
$$T_{SA}^k \approx T_{n,SA} \approx n(n+m-k)s, \quad T_{COM} \approx n(n+m).$$

For the total time of the implementation we obtain

$$\begin{aligned} T_{PAR} &= T_{ALU}^0 + T_{n,SA} + T_{COM} + \sum_{k=2}^{n} \left(T_{0,SA}^k + \max(T_{ALU}^k, T_{SA}^k) \right) \\ &\approx 2(n+m-k)s^3\alpha + n(n+m-k)s + n(n+m) \\ &\qquad + \sum_{k=2}^{n} ((n+m-k)s + (n+m-k)s \max(2s^2\alpha, n)) \\ &\approx n(\frac{n}{2} + m) \max(2s^3\alpha, ns) = \frac{N(N+2M)\max(2s^3\alpha, N)}{2s^2}. \end{aligned}$$

The corresponding approximate values for the speed-up and the efficiency are

$$\begin{aligned} \mathcal{S} &= \frac{T_{SEQ}}{T_{PAR}} \approx 2s^2 \min(1, \frac{N}{2s^3\alpha}), \\ \mathcal{E} &= \frac{\mathcal{S}}{2s^2 + 1/\alpha} \approx \left(\frac{2s^2\alpha}{2s^2\alpha + 1} \right) \min(1, \frac{N}{2s^3\alpha}). \end{aligned}$$

At the end of this subsection we pay some additional attention to the problem of finding of A^{-1} for a given nonsingular matrix $A \in \mathbf{R}^{N \times N}$. Of course, we can compute A^{-1} by applying Algorithm 4.3.3 with $B = I$. However, because of the sparsity of I, at each iteration a considerable part of it will remain unchanged, and therefore many superfluous operations will be performed. So to preserve their efficiency Algorithms 4.3.2 and 4.3.3 have to be slightly changed. Instead of starting with the extended matrix (A, I), we start with $A^{(0)} = A$ and compute iteratively the sequence $A^{(1)}, \ldots A^{(n)}$, where $A^{(k)} \in \mathbf{R}^{ns \times ns}$ for $k = 0, \ldots, n$. The matrix $A^{(n)}$ is a row-permutation of A^{-1} defined by the vector $J = (j_1, \ldots, j_{ns})$ containing the numbers of the successive pivots. The size of the iterated matrices remains unchanged, since at each step of iteration the s pivot columns are removed and the next s columns of I are added. The corresponding changes in ALU and SA operations of Algorithm 4.3.2 are

ALU: Compute $\vec{A}_k^{(k)}$ and $J_k = (j_{(k-1)s+1}, \ldots, j_{ks})$ from $(\vec{A}_k^{(k-1)}, I_s)$,
and
SA: $A_i^{(k-1)} := (\vec{A'}_i^{(k-2)}, O_s) - A_{iJ_{k-1}}^{(k-2)}\, A_{k-1}^{(k-1)}$,

where I_s and O_s are the unit and the zero blocks of size s and $\vec{A'}_i^{(k-1)}$ the block-row $\vec{A}_i^{(k-1)}$ without columns $j_{(k-1)s+1} \ldots, j_{ks}$. The approximate running times of corresponding implementation of described matrix inversion algorithm are

$$T_{ALU}^k \approx 2ns^3\alpha, \quad T_{0,SA}^k \approx ns, \quad T_{SA}^k \approx T_{n,SA} \approx n^2 s, \quad T_{COM} \approx n^2,$$

and consequently the approximate value of the total running time is

$$T_{PAR} \approx \frac{N^2}{s^2} \max(2s^3\alpha, N).$$

The approximate values for the speed-up and efficiency remain unchanged.

4.3 LL^T-Factorization by Cholesky's Algorithm

It is well known that symmetric and positive definite matrices can be factorized more economically in a product of two triangular matrices. For a given symmetric and positively definite matrix $A \in \mathbf{R}^{N\times N}$, the Cholesky's algorithm computes a lower triangular matrix $L \in \mathbf{R}^{N\times N}$ such that $A = LL^T$. In this subsection an implementation of the Cholesky's algorithm on the systolic computer is presented.

We represent the Cholesky's algorithm for a given symmetric positive definite matrix $A \in \mathbf{R}^{N\times N}$ in a form consistent with Algorithm 3.1, that finds the upper triangular matrix $U = L^T$.

ALGORITHM 4.4.1: *Cholesky U^TU-Factorization*
Input: $A \in \mathbf{R}^{N\times N}$ — symmetric positive definite matrix
Output: $U \in \mathbf{R}^{N\times N}$ — upper triangular such that $A = U^TU$

for $k = 1, \ldots, N$
$u_{kk} := \sqrt{a_{kk}^{(k-1)}}$
$\vec{u}_k[k+1:N] := u_{kk}^{-1}\ \vec{a}_k^{(k-1)}[k+1:N]$
  for $i = k+1, \ldots, N$
  $\vec{a}_i^{(k)}[i:N] := \vec{a}_i^{(k-1)}[i:N] - u_{ki}\ \vec{u}_k[i:N]$
  end i
end k.

The execution of this algorithm requires $N^3/3$ flops. This is twice better than the LU-factorization procedure and in addition only the upper half of the symmetric input matrix A is used.

Following our scheme we derive a block version of the Cholesky's algorithm, that is appropriately rearranged for implementation on the systolic computer.

ALGORITHM 4.3.2: *Block Version of Cholesky U^TU-Factorization*
Input: $A \in \mathbf{R}^{ns\times ns}$ — symmetric positive definite matrix
Output: $U \in \mathbf{R}^{ns\times ns}$ — upper triangular such that $A = U^TU$
(A) **ALU**: Find $\vec{U}_1$ from $\vec{A}_1$ by using Algorithm 4.4.1
(B) **for** $k = 2, \ldots, n$
(1) **SA**: $\vec{A}_k^{(k-1)}[k:n] := \vec{A}_k^{(k-2)}[k:n] - U_{k-1,k}^T\ \vec{U}_{k-1}[k:n]$
(2) **do** steps 2.1 and 2.2 **in parallel**
  (2.1) **ALU**: Find $\vec{U}_k[k:n]$ from $\vec{A}_k^{(k-1)}[k:n]$ by using Algorithm 4.4.1
  (2.2) **for** $i = k+1, \ldots, n$,
    SA: $\vec{A}_i^{(k-1)}[i:n] := \vec{A}_i^{(k-2)}[i:n] - U_{k-1,i}^T\ \vec{U}_{k-1}[i:n]$
    end i
 end k

ALU executes Steps A and 2.1 by applying Algorithm 4.4.1 with an upper bound of the two loop ranges s instead of N. SA executes update operations in Steps 1 and 2.2, which are of the type of the systolic operation (1). However, in this case transposed blocks of the result U are used by the updates.

These transposed blocks will be computed by the ALU together with the corresponding block-rows of U and will be stored together with them in the auxiliary memories. So in the k-th iteration ALU finds $\vec{U}_k[k:n]$ first, computes blocks $U^T_{k,k+1}, \ldots U^T_{kn}$ next and stores them in the auxiliary memory connected to ALU during that iteration. During the same iteration the other auxiliary memory is involved in the SA operations in Steps 1 and 2.1. This memory contains the transposed blocks $U^T_{k-1,k}, \ldots U^T_{k-1n}$ computed in the previous iteration. These blocks are loaded into SA via FBM and next the corresponding SA operation is executed as usually. The implementation of Cholesky's algorithm is presented below in details.

ALGORITHM 4.4.2: *Implementation of Cholesky U^TU-Factorization*
Input : $\vec{A}_i[i:n]$ stored in $BM(i-1)$, $i = 1, \ldots, n$
Output : $\vec{U}_i[i:n]$ stored in $BM(n-i)$, $i = 1, \ldots, n$

(A) $BM(0)[\vec{A}_1[1:n]] \longrightarrow M(0)$
  ALU: $M(0)\ \ [\vec{A}_1[1:n]] \rightarrow [\vec{U}_1[1:n],\, U^T_{12}, \ldots U^T_{1n}]$
(B) **for** $k = 2, \ldots, n$
(1) **SA**: $M(k+1)[\vec{A}_k^{(k-1)}[k:n]] := BM(1)[\vec{A}_k^{(k-2)}[k:n]] - M(k)[U^T_{k-1,k}]\ M(k)[U_{k-1}[k:n]]$
(2) **do** steps 2.1 and 2.2 **in parallel**
  (2.1) **ALU**: $M(k+1)\ \ [\vec{A}_k^{(k-1)}[k:n]] \rightarrow [\vec{U}_k[k:n],\, U^T_{k,k+1}, \ldots U^T_{kn}]$
  (2.2) **for** $i = k+1, \ldots, n$
    SA: $BM(i+k)[\vec{A}_i^{(k-1)}[i:n]] := BM(i+k+1)[\vec{A}_i^{(k-2)}[i:n]] - M(k)[U^T_{k-1,i}]\ M(k)[\vec{U}_{k-1}[i:n]]$
    end i
(3) $M(k)[\vec{U}_{k-1}[k-1:n]] \overset{via\ FBM}{\longrightarrow} BM(n-k+1)$
  end k
(C) $M(n+1)[U_{nn}] \overset{via\ FBM}{\longrightarrow} BM(0)$.

Note that our implementation uses the upper half of the symmetric matrix A and therefore the upper bound for the size of the input matrix is twice bigger than this for the LU-factorization.

We evaluate the running time of the implementation of the Cholesky's algorithm following the general scheme. We use approximate values for the running times of the procedures involved in the implementation and sum them. The corresponding approximate values are as follows: for the ALU procedures in Steps A and 2.1 –

$$T_{ALU}^{k} \approx (n-k)s^3\alpha \quad k=1,\ldots,n;$$

for the SA procedures in Steps 1 and 2.2 –

$$T_{0,SA}^{k} \approx (n-k)s, \quad T_{SA}^{k} \approx \frac{(n-k)^2 s}{2} \quad k=2,\ldots,n;$$

and for data transfer – $T_{COM} \approx n^2 s$.

For the running time of the implementation we obtain

$$\begin{aligned} T_{PAR} &= T_{ALU}^{1} + T_{COM} + \sum_{k=2}^{n}(T_{0,SA}^{k} + \max(T_{ALU}^{k}, T_{SA}^{k})) \\ &\approx ns^3\alpha + n^2 s + \sum_{k=2}^{n}(n-k)s + \sum_{k=2}^{n}\max\left(\frac{(n-k)^2 s}{2}, (n-k)s^3\alpha\right) \\ &\approx \begin{cases} \frac{n^3 s}{6}(1+4(\frac{s^2\alpha}{n})^3) & \text{for} \quad 2s^2\alpha \le n \\ \frac{n^2 s^3\alpha}{2} & \text{for} \quad 2s^2\alpha \ge n, \end{cases} \end{aligned}$$

The corresponding approximate values for the speed-up and efficiency are

$$\begin{aligned} \mathcal{S} &= \frac{T_{SEQ}}{T_{PAR}} \approx \begin{cases} \frac{2s^2}{(1+4(s^3\alpha/N)^3)} & \text{for} \quad 2s^3\alpha \le N \\ \frac{2N}{3s\alpha} & \text{for} \quad 2s^3\alpha \ge N, \end{cases} \\ \mathcal{E} &= \frac{\mathcal{S}}{2s^2+1/\alpha} \approx \begin{cases} \frac{2s^2\alpha}{(1+2s^2\alpha)(1+4(s^3\alpha/N)^3)} & \text{for} \quad 2s^3\alpha \le N \\ \frac{2N}{3s(1+2s^2\alpha)} & \text{for} \quad 2s^3\alpha \ge N, \end{cases} \end{aligned}$$

4.4 QR-Factorization by Householder Transformations

In this subsection we show that the numerically favorable and widely used Householder reflection method for QR-factorization of a rectangular full-rank matrix can be effectively implemented on our architecture. The key-stone of that implementation is so called WY-representation [5] for products of Householder matrices and its compact version, proposed in [9].

Let $A \in \mathbf{R}^{M\times N}$, $M \ge N$ be a full-rank matrix. The problem of QR-factorization of A is to compute an orthogonal matrix $Q \in \mathbf{R}^{M\times M}$ and an upper triangular matrix $R \in \mathbf{R}^{M\times N}$ such that $A = QR$. We begin by presenting the well-known Householder QR-factorization algorithm [7].

Algorithm 4.5.1: *Householder QR-factorization*
Input: $A \in \mathbf{R}^{M\times N}$, $M \geq N$, $\mathrm{rank}(A) = N$
Output: $R \in \mathbf{R}^{M\times N}$ as described above
for $k = 1, \ldots, N$
(1) Determine a unit 2-norm reflection vector $v_k = (v_{kk}, \ldots, v_{Mk})^T$ so that
$$(I - 2v_k v_k^T)\, a^{(k-1)}_{\downarrow k}[k\!:\!M] = (r_{kk},\, 0, \ldots, 0)^T$$
(2) $z_1[k\!+\!1\!:\!N] := -2v_k^T a^{(k-1)}[k:M, k+1:N]$
(3) $r_k[k\!+\!1\!:\!N] := a_k^{(k-1)}[k+1\!:\!N] + v_{kk} z_1[k+1\!:\!N]$
for $i = k+1, \ldots, M$
$a_i^{(k)}[k\!+\!1\!:\!N] := a_i^{(k-1)}[k+1\!:\!N] + v_{ik} z_1[k+1\!:\!N]$
end k.

This algorithm requires $MN^2 - N^3/3$ flops. If the matrix Q is required, it can be easily accumulated during the algorithm from the formulas $Q^{(0)} = I$ and $Q^{(k)} = \mathrm{diag}(I_{k-1},\, I - 2v_k v_k^T) Q^{(k-1)}$, $k = 1, \ldots, N$.

The Householder QR-factorization is used in many applications as least-square, eigenvalue and other problems. According to our approach we need a block version of Algorithm 4.5.1 that will allow us to apply our major scheme, as described in Section 3. For full-rank matrices such a block version has been proposed in [5]. It is based on so called WY-representation of products of Householder matrices. Subsequently, in [9] a compact WY-representation which reduces the space requirements has been presented.

The algorithm below computes a lower trapezoidal matrix $Y \in \mathbf{R}^{M\times N}$, $W \in \mathbf{R}^{M\times N}$, and an upper triangular matrix $T \in \mathbf{R}^{N\times N}$ so that

$$A = (I + WY^T)\, R, \qquad Y\,T = W$$

and $I + WY^T$ is orthogonal.

Algorithm 4.5.1a: *Generation of Y, W, and T for Algorithm 4.5.1*
Input: $A, R \in \mathbf{R}^{M\times N}$, $M \geq N$, $\mathrm{rank}(A) = N$
Output: Y, W, and T as described above
$Y^{(0)}, W^{(0)}, T^{(0)} :=$ [empty]
for $k = 1, \ldots, N$

(1) $z := -2Y^{(k-1)T}\hat{v}_k, \quad$ where $\hat{v}_k = (0, \ldots, 0,\ v_k^T)^T \in \mathbf{R}^{M\times 1}$
(2) $Y^{(k)} := [Y^{(k-1)},\ \hat{v}_k]$
(3) $W^{(k)} := [W^{(k-1)},\ -2\hat{v}_k + W^{(k-1)}z]$
(4) $T^{(k)} := \begin{bmatrix} T^{(k-1)} & T^{(k-1)}z \\ 0 & -2 \end{bmatrix}$
end k.

This algorithm together with Algorithm 4.5.1 requires $2MN^2 - N^3/3$ flops.

By using Algorithm 4.5.1/1a it is easy to obtain a block version of the Householder QR factorization as follows.

ALGORITHM 4.5.2: *Block Householder QR-factorization*
Input: $A \in \mathbf{R}^{ms\times ns}$, $m \geq n$, $\text{rank}(A) = ns$
Output: $R, Y \in \mathbf{R}^{ms\times ns}$ and $T = \text{diag}(T_{11}, \ldots, T_{nn}) \in \mathbf{R}^{ns\times ns}$ as described above
for $k = 1, \ldots, n$
(1) Apply Algorithm 4.5.1/1a on $A^{(k-1)}_{\downarrow k}[k:m] \in \mathbf{R}^{(m-k+1)\times s}$ to find $R_{\downarrow k}[k\!:\!m]$,
$Y_{\downarrow k}[k\!:\!m]$, $W_{\downarrow k}[k\!:\!m]$, and T_{kk}
(2) $Z_1[k{+}1\!:\!n] := W^T_{\downarrow k}A^{(k-1)}[k:m, k+1:n]$
(3) $R_k[k{+}1\!:\!n] := A^{(k-1)}_k[k+1:n] + Y_{kk}Z_1[k+1:n]$
for $i = k+1, \ldots, m$
$A^{(k)}_i[k{+}1\!:\!n] := A^{(k-1)}_i[k+1:n] + Y_{ik}Z_1[k+1:n]$
end k.

This algorithm requires $MN^2(1 + 2s/N) - N^3/3$ flops. It is seen that the amount of $2MNs$ additional flops are paid in this block version.

The next algorithm is a reorganized version of Algorithm 4.5.2 suitable for our architecture (cf. with Algorithm 3.2).

ALGORITHM 4.5.3: *Modified Block Householder QR-factorization*
Input: $A \in \mathbf{R}^{ms\times ns}$, $m \geq n$, $\text{rank}(A) = ns$
Output: $R, Y \in \mathbf{R}^{ms\times ns}$ and $T = \text{diag}(T_{11}, \ldots, T_{nn}) \in \mathbf{R}^{ns\times ns}$ as described above
(A) Find R_{11}, $Y_{\downarrow 1}[1:m]$, $W_{\downarrow 1}[1:m]^T$, and T_{11} applying Algorithm 4.5.1/1a on $A_{\downarrow 1}[1\!:\!m]$

(B) **for** $k = 2, \ldots, n$

(0) $Z_1[k:n] := W_{\downarrow k-1}[k-1:m]^T\, A^{(k-2)}[k-1:m, k:n]$

(1) $A^{(k-1)}_{\downarrow k}[k:m] := A^{(k-2)}_{\downarrow k}[k:m] + Y_{\downarrow k-1}[k:m]\, Z_{1k}$

(2) **do** steps 2.1 and 2.2 **in parallel**

(2.1) Find R_{kk}, $Y_{\downarrow k}[k:m]$, $W_{\downarrow k}[k:m]^T$, and T_{kk} applying Algorithm 4.5.1/1a

on $A^{(k-1)}_{\downarrow k}[k:m]$

(2.2) $R_{k-1}[k:n] := A^{(k-2)}_{k-1}[k:n] + Y_{k-1,k-1}\; Z_1[k:n]$

for $i = k, \ldots, m$

$A^{(k-1)}_i[k+1:n] := A^{(k-2)}_i[k+1:n] + Y_{i,k-1}\; Z_1[k+1:n]$

end k.

Finally, we obtain the following implementation.

Algorithm 4.5.4: *Implementation of Householder QR-factorization*

Input: A_i stored in S_i, $i = 1, \ldots, m$

Output: $[Y_i[1:i], R_i[i:n], T_{ii}]$ stored in S_{m-i+1}, $i = 1, \ldots, n$;
Y_i stored in S_{m-i+1}, $i = n+1, \ldots, m$

(A) **for** $i = 1, \ldots, m$ $BM(i+1)[A_{i1}] \longrightarrow M(0)$

ALU: $M(0)$ $[A_{\downarrow 1}[1:m]] \to [R_{11}, T_{11}, Y_{\downarrow 1}[1:m], W_{\downarrow 1}[1:m]^T]$

(B) **for** $k = 2, \ldots, n$

(1) $BM(0)[A^{(k-2)}_{k-1}[k:n]] \longrightarrow M(m)$

for $i = k-1, \ldots, m$ **do** steps 1.1 and 1.2 **in parallel**

(1.1) **SA**: $FBM[Z^{(i)}_1[k:n]] :=$

$$\left\{ \begin{array}{cl} \text{—} & (\text{if } i = k-1) \\ FBM[Z^{(i-1)}_1[k:n]] & (\text{if } i > k-1) \end{array} \right\} + M(k)[W^T_{i,k-1}]\; M(i+k+m+1)[A^{(k-2)}_i[k:n]]$$

(1.2) **if** $i < m$ **then**

$BM(i+k)[A^{(k-2)}_{i+1}[k:n]] \longrightarrow M(i+k+m)$

end i

$FBM[Z_1[k:n]] \longrightarrow M(k)$

for $i = k, \ldots, m$

SA: $M((k+1) \bmod 2)[A^{(k-1)}_{ik}] := BM(i+k+1)[A^{(k-2)}_{ik}] + BM(i+1)[Y_{i,k-1}]\; M(k)[Z_{1k}]$

(2) **do** steps 2.1 and 2.2 **in parallel**

(2.1) **ALU**: $M(k+1)$ $[A^{(k-1)}_{\downarrow k}[k\!:\!m]] \rightarrow$

$[R_{kk},\, T_{kk},\, Y_{\downarrow k}[k\!:\!m],\, W_{\downarrow k}[k\!:\!m]^T]$

(2.2) **SA**: $\left\{ \begin{array}{ll} FBM & \text{(if } k \text{ is even)} \\ BM(1) & \text{(if } k \text{ is odd)} \end{array} \right\} [R_{k-1}[k\!:\!n]] :=$

$BM(0)[A^{(k-2)}_{k-1}[k\!:\!n]] +$

$BM(k)[Y_{k-1,k-1}]\ M(k)[Z_1[k\!:\!n]]$

if $k \bmod 2 = 0$ **then** $FBM[R_{k-1}[k\!:\!n]] \longrightarrow BM(0)$

for $i = k, \ldots, m$

SA: $BM(i+k)[A^{(k-1)}_i[k+1\!:\!n]] :=$

$BM(i+k+1)[A^{(k-2)}_i[k+1\!:\!n]] +$

$BM(i+1)[Y_{i,k-1}]\ M(k)[Z_1[k+1\!:\!n]]$

(3) $M(k)[R_{k-1,k-1},\, T_{k-1,k-1}] \overset{via\ FBM}{\longrightarrow} BM(n-k+1)$

for $i = k-1, \ldots, m$ $M(k)[Y_{i,k-1}] \overset{via\ FBM}{\longrightarrow} BM(n-i+1)$

end k

(C) $M(n+1)[R_{nn},\, T_{nn}] \overset{via\ FBM}{\longrightarrow} BM(0)$

for $i = n, \ldots, m$ $M(n+1)[Y_{i,n}] \overset{via\ FBM}{\longrightarrow} BM(n-i+1)$.

Concerning the time complexity, we have

$$T^k_{ALU} \approx 2(m-k)s^3\alpha, \quad T^k_{0,SA} \approx (n-k)(m-k)s, \quad T^k_{SA} \approx (n-k)(m-k)s.$$

Therefore, we obtain $k_0 = \lfloor n - 2s^2\alpha \rfloor$ and

$$\begin{aligned} T_{PAR} \ &\sim \ 2ms^3\alpha + \sum_{k=2}^{n}(n-k)(m-k)s + \sum_{k=2}^{\lfloor n-2s^2\alpha\rfloor}(n-k)(m-k)s \\ &+ \sum_{k=\lfloor n-2s^2\alpha\rfloor+1}^{n} 2(m-k)s^3\alpha \sim mn^2s - \frac{n^3s}{3} + 2(m-n)s^5\alpha^2 + \frac{4s^7\alpha^3}{3} \\ &= \frac{MN^2}{s^2} - \frac{N^3}{3s^2} + 2(M-N)s^4\alpha^2 + \frac{4s^7\alpha^3}{3}. \end{aligned}$$

5 Conclusion

We described implementations of several important linear algebra algorithms on a simple architecture with mesh-connected processor array and systolic type of data movement. These algorithms can easily be adapted to run on a variety of other popular architectures without losing their efficiency. The methodology can also be easily extended to many other basic linear algebra algorithms not discussed in the paper.

References

[1] L. Aleksandrov and H. Djidjev, *Factorization of Symmetric Positive Definite Matrices on a Systolic Processor*, Proc. of the IFIP WG 10.3, Working Conference on Parallel Processing, Pisa, I April, 1988, in Parallel Processing, North-Holland, 1988.

[2] L. Aleksandrov and H. Djidjev, *Algorithms for a Specialized Matrix Systolic Processor*, SIAM/SIAG Second Intern. Conf. on Vector and Par. Comp., Tromso, Norway, 1988.

[3] L. Aleksandrov and H. Djidjev, *Solving Systems of Linear Equations on a Systolic Processor*, Tec. Rep. 1, Center of Informatics and Computer Technology, BAS, Sofia, 1988.

[4] L. Aleksandrov and M. Candev, *Householder QR-Decomposition on a Specialized Systolic Processor*, Advances in Parallel Algorithms, I. Dimov and O. Tonev (Eds.), IOS Press, Amsterdam, 1994, pp. 3–10.

[5] C. Bischof and C. Van Loan, *The WY Representation for Products of Householder Matrices*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. s2–s13.

[6] K.A. Gullivan, R.J. Plemmons, and A.H. Sameh, *Parallel Algorithms for Dense Linear Algebra Computations*, SIAM Review 32 (1990), 54–135.

[7] G. Golub and C. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, 1989.

[8] H. Kung and C. Leiserson, *Systolic Arrays (for VLSI)*, Sparse Matrix Proceedings, SIAM, 1978, pp. 256–282.

[9] R. Schreiber and C. Van Loan, *A Storage-Efficient WY Representation for Products of Householder transformations*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 53–57.