Regular Processor Arrays for Matrix Algorithms with Pivoting *

V. P. Roychowdhury T. Kailath

Information Systems Laboratory
Stanford University

Abstract

Pivoting operations are often necessary to ensure the satisfactory numerical behavior of several matrix algorithms. The resulting disruption of the regular operation flow makes it uncertain whether systolic array implementations (as commonly understood) can be found for such algorithms. By using results from the theory of Regular Iterative Algorithms (RIAs), introduced by Karp, Miller and Winograd [5] and recently extended by Rao, Jagadish and Kailath (see *e.g.,* [16], [4], [17]), we show how to obtain regular (though nonsystolic) processor arrays for algorithms with pivoting. First, the fact that pivoting algorithms cannot be systolic (as formally defined *e.g.,* in [22] and [17]) is established. Then it is shown how RIAs can be formulated for the Gaussian elimination algorithm with partial pivoting and how the algorithm can then be implemented on the so called regular iterative arrays (locally connected arrays of essentially identical processor modules, with register pipelines and/or LIFO buffers in some of the links). The generalizations of these ideas to other pivoting algorithms have been worked out in [22].

1 Introduction

Kung and Leiserson [8], [6], [7], proposed a class of regular locally connected structures known as systolic arrays for high speed parallel implementation of a number of algorithms in areas such as numerical linear algebra and graph theory. Since then, many authors have devised systolic arrays for a variety of problems. However, it is fair to say that until recently no general systematic method for design has been presented, though partial results appear in Moldovan [14], [13], Quinton [15], and especially Kung [9], [10]. Moreover, till recently there was no theoretical framework available for addressing issues such as the following. In their pioneering work, Kung and Leiserson [8] had presented a systolic array for the solution of linear equations by using the Gaussian elimination algorithm but *without* pivoting. However, except in special cases, one would need to use pivoting algorithms in order to obtain numerically meaningful results. Now pivoting (either row or column or full) disturbs the regular pattern of operations in Gaussian elimination, and therefore it is perhaps not surprising that no one has yet devised a systolic array for pivoting algorithms. But does that mean that it is impossible to do so, or just that no one clever enough has still attacked the problem? It would be useful to know an answer to such a question. Moreover, suppose we have not succeeded in easily finding a systolic array for a given algorithm, one might ask if there is not some generalization of the notion of systolic arrays that would allow an acceptable solution? In other words, what feature(s) of a systolic array might we be willing to give up?

While there may always be questions outside the scope of any effort, nevertheless in the recent work of Rao, Jagadish, and Kailath (see [16], [4], [18], [17], [19], [3]), building on a seminal paper of Karp, Miller, and Winograd [5], some progress has been made in this direction. For example, this work allows us to say the following. A formal definition of systolic arrays can be obtained that captures their generally accepted properties, especially regularity (mostly identical processors),

*This work was supported in part by the SDIO/IST, managed by the Army Research Office under Contract DAAL03-87-K-0033, the Department of the Navy (NAVELEX) under Contract N00039-84-C-0211, NASA Headquarters, Center for Aeronautics and Space Information Sciences (CASIS) under Grant NAGW-419-S1.3, and the SDI/IST Program managed by the Office of Naval Research, under Contract N00014-85-K-0550.

spatial locality ('local' interconnections), temporal locality (no delay-free operations, or more precisely, all combinational elements are latched) and pipelined operation (throughput independent of the order, suitably defined, of the system). Some authors (*e.g,* Leiserson *et al.* [11]) use only a subset of these properties, but the present consensus in the literature appears to require all those mentioned above (see *e.g.,* [19] and [10]). Using this definition, and its consequences, one can formally show [22] for example that there is no systolic array for algorithms that use pivoting (Gaussian elimination, QR-factorization, *etc*). However, there is a reasonable generalization of the concept that does allow implementation of such algorithms (and many others, including of course all systolic algorithms). The generalization allows the presence of register pipelines of various lengths at different points in a regular array of (mostly) identical processors, and sometimes also some LIFO (Last-In-First-Out) buffers. Rao *et al.* called such arrays Regular Iterative Arrays, and algorithms implementable on such arrays were dubbed as Regular Iterative Algorithms. For RIAs with bounded index spaces, Rao *et al.* [16], [17], [18] provided formal methods to determine lower bounds on I/O latency and memory requirements; systematic procedures for implementing RIAs on regular processor arrays that can achieve the lower bound on I/O latency and can generate all possible architectures were also developed. The implications of these results are quite extensive. It has been shown [16] [4] [10] that many algorithms in digital filtering (convolution, correlation, autoregressive, and moving-average filtering), numerical linear algebra, discrete methods for PDEs and ODEs, graph theory (transitive closure, some coloring problems) can be reformulated as RIAs.

In this paper, we shall show how the results of Rao *et al.*, and some additions to it that we have obtained, can be used to obtain regular processor arrays for matrix pivoting algorithms, and in particular work out the example of Gaussian elimination with partial (row or column) pivoting. This method can also be extended to other pivoting algorithms, *e.g.,* Gaussian elimination with full pivoting and QR-factorization with column pivoting [22]. Space limitations will not allow us to report all the deatils which the interested reader can find in [22]. We shall begin by reviewing the definition and basic properties of RIAs. Due to space limitations it is not possible to provide a complete characterization of the *systolic algorithms* (defined as the algorithms implementable on systolic arrays) here. However, we shall state a necessary property of systolic algorithms using which we can show that pivoting algorithms are non-systolic. In section 3, we show how Gaussian elimination with partial pivoting can be written in RIA form. We shall then outline how several architectures can be derived, including it turns out certain arrays previously obtained by Capello [1] (2-dimensional arrays) and by Ipsen, Saad and Schultz [2] (1-dimensional arrays). Our approach (Pivoting algorithms $\rightarrow$ RIAs $\rightarrow$ Dependence Graphs $\rightarrow$ Processor arrays) allows us to generate different architectures (both 1 and 2-dimensional) by simply changing specific parameters in the procedure that maps the dependence graph of the algorithms to processor arrays. the details). Finally, section 4 has some concluding remarks.

2 Regular Iterative Algorithms

A major step towards extracting parallelism in a given algorithm is to express the algorithm in a proper language. Sequential programming languages such as Fortran or Pascal have built-in orderings of the computations that obscure parallelism in the algorithm. Also, in an effort to minimize storage requirements the practice of overwriting on variables is encouraged in these languages, which compounds the problem of extracting parallelism even more. *Single Assignment* languages were designed to overcome the difficulties mentioned above by requiring that every variable defined in the program take on a unique value during the course of execution. Given a single assignment algorithm, it is possible to capture the information regarding the parallelism in the algorithm by means of a *dependence graph.* To implement the algorithm in parallel one simply has to schedule the computations on the available processors subject to the precedent constraints specified by the dependence graph. In general, the problem of optimally scheduling is hard and any available structure in the dependence graph is very desirable.

The Regular Iterative algorithms mentioned in the introduction are necessarily in the single assignment form and have the added property that the dependence graph of the algorithm is highly regular, which fact was exploited by Karp *et al.* [5], and later by Rao *et al.* (see *e.g.,* [16], [18], [17], [20]) to optimally schedule them on processor arrays.

Example 1: A simple RIA.

For all tuples (i,j) , $1 \leq i,j \leq N$ **do**

$$x(i,\ j) = jx(i-1,\ j+1)y(i,\ j)$$
$$y(i,\ j) = iy(i+1,\ j-1)+x(i,\ j-1)$$

This example displays the following (characteristic) features of an RIA:

Each variable in the RIA is identified by a label and an *index vector* ($I = [i\ j]^T$, in the example). The main feature is the regularity of the dependences among the variables with respect to the index points. That is, if $x(I)$ is computed using the value of $y(I-d)$ then the *index displacement vector* d, corresponding to this direct dependence, is the same regardless of the index point I.

As a consequence of this regularity, the dependence graph of an RIA has an iterative structure, which can be clearly demonstrated by drawing the dependence graph within the index space. The formal definition of the RIA follows:

Definition 1 *A Regular Iterative algorithm is defined by a triple* {**I**, **X**, **F**}*where*

1. **I** *is the Index-Space which is the set of all lattice points enclosed within a specified region in a S-dimensional Euclidean space.*
2. **X** *is a set of V variables that are defined at every point in the index space, where the variable x_j defined at the index-point I will be denoted as $x_j(I)$ and takes on an unique value in any particular instance of the algorithm, and*
3. **F** *is the set of functional relations among the variables, restricted to be such that if $x_i(I)$ is computed using $x_j(I - D_{ji})$, then*
   - **a.** *D_{ji} is a constant vector independent of I and the extent of the index space, and*
   - **b.** *for every J contained in the index-space, $x_i(J)$ is directly dependent on $x_j(J - D_{ji})$ (if $(J - D_{ji})$ falls outside the index-space, then, $x_j(J - D_{ji})$ is an external input to the algorithm).*

We should comment that the functional relations among the variables in **F** may involve conditional branches, as will be seen in later sections.

2.1 Writing Algorithms in RIA form

Algorithms are seldom specified as RIAs and certain amount of work has to be done to write an algorithm in the RIA form. Unfortunately, this first step is still somewhat heuristic, though some useful rules can be deduced from the very many diverse problems that have already been studied. The heuristic rules that have been found for conversion can be stated as a three-step procedure:

1. *Single Assignment Form:* Introduce additional indices for the variables in the algorithm so that the resulting modified algorithm is in Single Assignment form.
2. *Index-Matching*: Arrange that all variables have the same number of indices.
3. *Localization*: This step ensures that a variable at any particular index point must be dependent upon variables at neighboring index-points only. This can be accomplished, in most cases, by using conditionals and by propagating the variables across the index-space in a regular fashion.

In the next section, we shall illustrate the use of these rules by deriving the RIA for GAussian elimination algorithm with partial pivoting.

2.2 Non-systolic Algorithms and Matrix Pivoting Algorithms

Can the class of systolic algorithms be precisely defined? Of course the answer will depend on whether the systolic arrays can be precisely defined. Several authors including S. Y. Kung [9], Melhelm and Rheinboldt [12], and Leiserson and Saxe [11] have provided characterizations of systolic arrays, though they do not all use the same conception of such arrays. A comparative study of various definitions along with a formal definition that captures the generally accepted qualitative features of systolic arrays (*e.g.*, topological regularity, functional regularity, temporal locality, and pipelineability) can be found in [22], [16], [19]. Based on this definition it can be proved that systolic algorithms form a precise sub-class of the RIAs. The necessary and sufficient conditions for an RIA to be systolic can be stated in various ways however, one characterization of systolic algorithms that is particularly useful in our context is given by the following theorem.

Theorem 1 *If N is the maximum of the bounds on the indices in the index-space of a systolic RIA, then the I/O latency of the algorithm is $O(N)$.*

A proof of this can be found in [22], however one conversant in systolic arrays can easily relate to it because all the known systolic algorithms have linear schedules and hence satisfies the above theorem. As a counter example, one can show that the simple RIA in example 1 has a directed path of length N^2 connecting all $x(i,j)$. Thus, a linear schedule is not possible and the algorithm is non-systolic.

A simple analysis will point to the difficulties that appear in implementing pivoting algorithms on systolic arrays. A characteristic feature of all pivoting algorithms applied to an $N \times N$ matrix is that there are N steps and that each step has two distinct phases: (i) determining the maximum of $\Theta(N)$ elements; (ii) performing operations that are dependent on the result of (i). Note that the operations in the next step of the algorithm cannot begin until and unless phase (i) of the present step has been completed. Since there are N steps in the algorithm, the dependence graph of the algorithm has a path of length $\Omega(N \times ($ The path length required to compute the maximum of $\Theta(N)$ elements)). The minimum path length in any dependence graph (when the in-degree of nodes is restricted to be bounded) for computing the maximum of $\Theta(N)$ elements is $\Omega(\log N)$. Hence the I/O latency of the matrix pivoting algorithms is $\Omega(N \log N)$. Thus systolic array implementation (with size parameter N) is not possible because the I/O latency of a systolic algorithm is $O(N)$ whereas the I/O latency of the pivoting algorithms is $\Omega(N \log N)$. We should remark that the present analysis is valid for the algorithms that are known; it does not rule out the possibility that faster algorithms for performing pivoting operation, with lower I/O latency can exist.

3 Gaussian Elimination with Partial Pivoting

The Gaussian elimination algorithm with partial pivoting is the most widely used procedure for solving systems of linear equations and factorizing matrices. We showed in section 2 that it cannot be implemented on systolic arrays. However, Rao *et al.* have shown that if the pivoting algorithms can be expressed as RIAs, then they can be implemented on regular processor arrays with register pipelines and LIFO buffers. Such architectures have almost all the advantages, such as spatial locality, modularity, and easy programmability, that make systolic arrays so appealing for VLSI. In this section we shall show that the partial pivoting algorithms can indeed be written as RIAs. In particular, we shall show that for an $N \times N$ matrix the algorithm can be implemented on a linear array of processors with $O(N)$ memory at each processor and in $O(N^2)$ time (we should note here that some of the results of this section were presented by the authors in [21]). The linear arrays ingeniously derived in [2] can be shown to be particular instances of the family of architectures that can be generated by our procedure. Also we can show that the two-dimensional arrays reported in [1] can be naturally generated from arrays that can be derived from our systematic procedure.

3.1 RIA Representation

In order to come up with an RIA for the gaussian elimination algorithm with partial pivoting problem we first analyze the sequential method of executing it. A sequential representation of the algorithm is as follows:

```
For $k$ := 1 to $(m-1)$ do
begin
    Determine $p \epsilon \{k, k+1, \ldots, n\}$ so $|a_{pk}| = \max |a_{ik}|$;
    $r_k$ := $p$;
    For $i$ := $k$ to $n$ do
    begin
        Swap $a_{ki}$ and $a_{pi}$;
        end
        $w_j := a_{kj}$;
        For $i := k + 1$ to $m$ do
        begin
            $\eta := a_{ik}/a_{kk}$;
            $a_{ik} := \eta$;
            For $j = k + 1$ to $n$ do
            begin
                $a_{ij} := a_{ij} - \eta w_j$;
            end
        end
    end
```

The permutation of rows performed at each step of the algorithm is data-dependent and cannot be predicted prior to its execution. This makes the dependence graph of this algorithm data-dependent and appears to have been the main stumbling block in previous attempts to obtain systolic array implementations. However, if we temporarily relax the restriction that the final output has to be in triangular form, then we can select the pivot element at each step and use the corresponding row for elimination, except that we do not need to swap rows at each step. We can then output for each column the row number of the pivot element and this information can easily be used to triangularize the output matrix in $O(N)$ time (for details see [22]). This modified version has a data-independent dependence graph and can be divided into $M - 1$ steps, one for every value of k. Let a_{ij}^k denote the value of a_{ij} at the k^{th} step. Also, let c_{ij}^k be an one-bit tag such that if $c_{ij}^k = 1$ then it implies that the corresponding element belongs to a row which has been used for elimination at some step $k_0 < k$. The k^{th} step of the algorithm consists of two distinct passes over the matrix elements and the passes can be enumerated as follows:

- The first pass determines the pivotal row, that is the row with the maximum element in the k^{th} column (note we will be careful in not considering the rows that have already been selected as pivotal rows in previous steps). This will be done by sequentially examining all the elements of the k^{th} column. Let $r_{(i-1)k}^k$ be the row with the maximum value in the k^{th} column among the first $i - 1$ rows and let $s_{(i-1)j}^k$ denote the j^{th} element of the row $r_{(i-1)k}^k$. Then we can define a boolean variable t_{ik}^k as follows:

  $$t_{ik}^k = s_{(i-1)k}^k < |a_{ik} \times \overline{c_{ik}}|$$

  where the bar denotes the logical complement operation. Thus, the variable is set to 1 if $c_{ik}^k = 0$ and $\left|a_{ik}^k\right| > s_{(i-1)k}^k$, otherwise it is set to 0. Note that the requirement $c_{ik}^k = 0$ insures that the rows that have already participated as pivotal rows are not considered. The candidate for the pivotal row ($i.e.$, r_{ik}^k and its contents can now be updated as follows:

  $$r_{ik}^k := (r_{(i-1)k}^k \times \overline{t_{ik}^k}) + (i \times t_{ik}^k);$$

  $$s_{ij}^k := (\overline{t_{ik}^k} \times s_{(i-1)j}^k) + (t_{ij}^k \times a_{ij}^k)$$

  Hence, at the end of first pass r_{Mk}^k is the row-number of the pivotal row and the contents of the pivotal row are in $s_{Mj}^k \; \forall \, j = k \ldots N$.

- The second pass uses the pivotal row to update the untagged elements ($i.e.$, a_{ij}^k for which $c_{ij}^k = 0$) and also tags the elements of the pivotal row. The updating for the i^{th} row can be easily done by defining a multiplier ρ_{ik}^k as follows:

  $$\rho_{ik}^k := (\frac{a_{ik}^k}{s_{Mk}^k}) \times \overline{c_{ik}^k \vee (i = r_{Mk}^k)}$$

where $\vee$ is the logical *or* operation. Notice that if $c^k_{ik} = 1$ (*i.e.*, the i^{th} row was chosen as a pivotal row in an earlier update) or $i = r^k_{Mk}$ (*i.e.*, the i^{th} row is the pivotal row as determined in the previous step) then the multiplier is set to 0, so that these rows are not modified again. The elements and the corresponding tags of the i^{th} row can now be updated as follows:

$$a^{k+1}_{ij} := a^k_{ij} - \rho^k_{ik} s^k_{Mj}$$

$$c^{k+1}_{ij} := c^k_{ij} \vee (i = r^k_{Mk})$$

Once this updating is completed one can start the $k+1$ step.

The purpose of introducing the subscripts and superscripts in the description of the variables was to avoid overwriting on variables. In fact, from the description of the modified algorithm so far, one can directly code it in single assignment form as follows:

For all triples (i,j,k), $1 \le i \le M;\ k \le j \le N$ and $1 \le k \le M-1$ **do**

$$\begin{aligned}
t(i,k,k) &= s(i-1,k,k) < \left|a(i,k,k) \times \overline{c(i,k,k)}\right| \\
r(i,k,k) &= (r(i-1,k,k) \times \overline{t(i,k,k)}) + (i \times t(i,k,k)) \\
s(i,j,k) &= (\overline{t(i,k,k)} \times s(i-1,j,k)) + (t(i,k,k) \times a(i,j,k)) \\
\rho(i,k,k) &= \left(\frac{a(i,k,k)}{s(i,k,k)}\right) \times \overline{c(i,k,k) \vee (i = r(i,k,k)} \\
a(i,j,k+1) &= a(i,j,k) - \rho(i,k,k)s(M,j,k) \\
c(i,j,k+1) &= c(i,j,k) \vee (i = r(M,k,k))
\end{aligned}$$

where the necessary initializations are

$$a(i,j,0) = a_{ij};\ c(i,j,0) = 0;\ s(0,k,k) = 0.$$

3.1.1 An RIA for Gaussian Elimination with Partial Pivoting

An RIA can be derived from the single assignment form of the algorithm by introducing propagating variables, in the same manner as was done for the Gaussian elimination algorithm without pivoting. For example, $s(i,j,k)$ uses $t(i,k,k)$ and this leads to a global dependence. The global dependence can be removed by propagating $t(i,j,k)$ along the j^{th} coordinate as follows:

$$t(i,\ j,\ k) = \begin{cases} s(i-1,\ j,\ k) < \left|a(i,\ j,\ k) \times \overline{c(i,\ j,\ k)}\right| & \text{if } j = k \\ t(i,\ j-1,\ k) \text{ if } j > k \end{cases}$$

Similarly, $a(i,j,k)$ depends on $s(M,j,k)$ and to localize this dependence we define a new variable $w(i,j,k)$ that propagates $s(M,j,k)$ along the negative i direction. Using such techniques one can localize all the dependences and an RIA for performing Gaussian elimination with partial pivoting is:

For all triples (i,j,k), $1 \le i \le M;\ k \le j \le N$ and $1 \le k \le M-1$ **do**

$$s(i,\ j,\ k) = \overline{t(i,\ j,\ k)} \times s(i-1,\ j,\ k) + t(i,\ j,\ k) \times a(i,\ j,\ k)$$

$$t(i,\ j,\ k) = \begin{cases} p(i-1,\ j,\ k) < \left|a(i,\ j,\ k) \times \overline{c(i,\ j,\ k)}\right| & \text{if } j = k \\ t(i,\ j-1,\ k) \text{ if } j > k \end{cases}$$

$$r(i,\ j,\ k) = \begin{cases} r(i-1,\ j\ k) \times \overline{t(i,\ j,\ k)} + i \times t(i,\ j,\ k) & \text{if } j = k \\ \text{null otherwise} \end{cases}$$

$$w(i,\ j,\ k) = \begin{cases} s(i,\ j,\ k) \text{ if } i = M \\ w(i+1,\ j,\ k)\ if\ i < n \end{cases}$$

$$x(i,\,j,\,k) \;=\; \begin{cases} r(i,\,j,\,k) & \text{if } i = M \text{ and } j = k \\ x(i+1,\,j,\,k) & \text{if } j = k \\ x(i,\,j-1,\,k) & \text{if } j > k \end{cases}$$

$$\rho(i,\,j,\,k) \;=\; \begin{cases} \left(\dfrac{a(i,\,j,\,k)}{w(i,\,j,\,k)}\right) \times \overline{c(i,\,j,\,k) \vee (x(i,\,j,\,k) = i)} \\ \qquad\qquad\qquad\qquad \text{if } j = k \\ \rho(i,\,j-1,\,k) \quad \text{if } j > k \end{cases}$$

$$c(i,\,j,\,k+1) \;=\; c(i,\,j,\,k) \vee (i = x(i,\,j,\,k))$$

$$a(i,\,j,\,k+1) \;=\; a(i,\,j,\,k) - \rho(i,\,j,\,k) \times w(i,\,j,\,k)$$

The necessary initializations are

$$a(i,\,j,\,0) \;=\; a_{ij};\; c(i,j,0) = 0;\; s(0,j,k) = 0.$$

3.2 Implementation on Processor arrays

An optimal implementation of a given algorithm minimizes the time required for completion of the algorithm and maximizes the utilization of the processors. The minimum time required to solve an algorithm is determined by the longest path in its dependence graph. It is quite simple to observe that the minimum I/O latency for the RIA in Section 5.1 is $O(M^2)$ for an $M \times N$ matrix (*i.e.* for every value of k there is a path of $O(M)$ and k ranges from 1 to $M-1$) and the total number of computations (*i.e.* the number of nodes in the index space of the RIA) is $O(M^2N)$. Hence, an optimal implementation can be realized by an array of processors consisting of $O(N)$ processors and requiring $O(M^2)$ time.

In order to implement the algorithm on processor arrays we have to partition the tasks in the dependence graph in an efficient among the processors. To keep the problem of scheduling easy and tractable one can restrict attention to linear (linear sub-spaces *e.g.,* hyper-planes) partitions. Here, a set of parallel lines (planes, hyper-planes; called the *iteration space*) are drawn through the index-space of the RIA, so that all computations that correspond to the index-points that lie on the same line (plane, hyper-plane) are assigned to the same processor. It can be shown that there is always at least one choice of the iteration space such that the resulting allocation will lead to asymptotically optimal scheduling. The processor array can be obtained by projecting the dependence graph along the sub-spaces defined by the iteration space.

Algebraically, the iteration space U is a $S \times P$ integral matrix where S is the dimension of the index-space of the RIA and $P < S$. Computations corresponding to index-points I and J are mapped on to the same processor if and only if $I - J \;=\; U\alpha$ where α is a $P \times 1$ vector. For example, a possible choice of the iteration space in the case of the RIA for gaussian elimination with pivoting is:

$$U \;=\; \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}^T .$$

The parallel planes along which the projections are done are shown in Fig. 1. This results in a processor array with N processors and is shown in Fig. 2. Often an algebraic approach for obtaining the processor array is useful. Let P be any $(S-P) \times S$-dimensional integer matrix of rank $(S-P)$ that is orthogonal to U, i.e.,$PU \;=\; 0$. Then, the processor array is defined by the lattice of points obtained by mapping the index space according to $p \;=\; PI$, $\forall I \in$ index space. p defines the location of the processor that carries out the computation at the index point I. The necessary interprocessor communication links are then defined by the vector weights on the edges: if $x(I)$ is dependent on $y(I-D)$, then there must be a directed link in the processor array from $P(I-D) \;\rightarrow\; PI$ for all I. In our example, a possible choice of P for the given choice of U is $P \;=\; [0\;1\;-1]$. This readily gives us the interprocessor communications (as shown in Fig 2) and also tells us about the computations that each processor performs. For example,

$$P \begin{bmatrix} i \\ k \\ k \end{bmatrix} = [0\;1\;-1] \begin{bmatrix} i \\ k \\ k \end{bmatrix} \;=\; 0.$$

for all i and k. This brings out a major feature of this architecture, *viz.*, the computations required to determine the pivot and the multipliers (which occur at index points where $j = k$) have been mapped to the 0^{th} processor; the rest of the processors need to perform only elimination operations. In fact it is readily apparent from the mapping technique that at the k^{th} step of the algorithm the columns k through N are stored in processors 0 through $N - k - 1$. Then at the end of k^{th} step the columns are shifted one processor to the left to start the $(k+1)^{th}$ step. The partitioning of the computations to individual processors has to be followed by efficient scheduling of the computations. A general scheduling technique for RIAs, which achieves the lower bound on the I/O latency of the algorithm is discussed in [16] [17]. The scheduling and exact operation of each processor in the linear array has been worked out in [22].

A second interesting array can be obtained by choosing the iteration space as

$$U = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^T.$$

The size of the array (shown in Fig. 3) is M, and will have less number of processors than the previous architecture if $M < N$. However, unlike the first array, the reader can verify that every processor in this array takes part in computing the pivot element and the multipliers. In fact, the i^{th} processor has the i^{th} row stored in it and this architecture is referred to in [2] as the row-oriented architecture. Another architecture discussed in the same paper where the j^{th} processor contains the j^{th} column is obtained by choosing

$$U = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^T.$$

If we have p processors instead of N processors $(p < N)$, then one can map multiple columns to the same processor and one obtains a block-column scheme for implementing the algorithm.

Several two dimensional architectures can be obtained by choosing the iteration space to be one dimensional. A triangular processor array is obtained by choosing $U = [1\ 0\ 0]^T$. This array can be pipelined with pipelining period of $O(N)$ (thus dependent on the problem size). However, we can now solve N problems simultaneously in $O(N^2)$ time. If we choose $U = [0\ 0\ 1]^T$ then we obtain a $N \times N$ (assuming $M = N$) 2-D array that corresponds to the case where every element of the matrix resides in a single processor. Now, if we assign a block of $p \times p$ elements to the same processor then we obtain a square array with N^2/p^2 processors (*i.e.*, an array of size $N/p \times N/p$) which yields the architectures discussed in [1]. It is however quite remarkable that by thus partitioning the array one can obtain an improvement in the processing time. In particular, if $p = N^{1/3}$ then it is shown in [1] that the processing time can be as small as $O(N^{5/3})$.

4 Concluding Remarks

In this paper we have shown how the concept of Regular Iterative Algorithms can be used to systematically map pivoting algorithms on regular iterative arrays that generalize the concept of systolic arrays. The regular processor arrays that we can design includes the linear arrays designed by several other researchers for pivoting algorithms. A fair treatment of several related issues which could not be dealt with in detail in this paper can be found in [22].

5 Acknowledgements

The authors would like to thank Dr. S. K. Rao for his helpful comments and inspiration.

References

[1] Peter R. Capello. A mesh automaton for solving dense linear systems. *Proc. Of International Conference on Parallel Processing*, St. Charles, IL,, August 1985.

[2] I. Ipsen, Y. Saad, and H. M. Schultz. Complexity of dense linear system solution on a multiprocessor ring. *Linear Algebra Appl.*, 77, 1986.

[3] H. V. Jagadish. *Techniques for the Design of Parallel and Pipelined VLSI Systems for Numerical Computations.* PhD thesis, Stanford University, Stanford, California, Dec. 1985.

[4] H. V. Jagadish, S. K. Rao, and T. Kailath. Multi-processor architectures for iterative algorithms. *Proceedings of the IEEE*, Sept. 1987.

[5] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967.

[6] H. T. Kung. Let's design algorithms for VLSI systems. In *Proc. Caltech Conf. on VLSI*, pages 65–90, Jan. 1979.

[7] H. T. Kung. Why systolic architectures? *IEEE Computer*, 25:37–46, Jan. 1980.

[8] H. T. Kung and C. E. Leiserson. Systolic arrays for VLSI. In *Sparse Matrix Proceedings*, pages 245–282, Philadelphia:Society of Industrial and Applied Mathematicians, 1978.

[9] S. Y. Kung. On supercomputing with Systolic/Wavefront array processors. *Proceedings of the IEEE*, 39–46, July 1984.

[10] S. Y. Kung. *VLSI Array Processors*. Prentic Hall Series, 1987.

[11] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. *Proc. Third CalTech Conf. on VLSI*, Jan. 1983.

[12] R. G. Melhem and W. C. Rheinboldt. A mathematical model for the verification of systolic networks. *SIAM J. Computing*, 13:541–565, Aug. 1984.

[13] D. I. Moldovan. On the analysis and synthesis of VLSI algorithms. *IEEE Trans. on Computers*, C-31:1121–1126, Nov. 1982.

[14] D. I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proceedings of the IEEE*, 113–120, Jan. 1983.

[15] P. Quinton. *The Systematic Design of Systolic Arrays*. Technical Report, INRIA Report, Paris, 1983.

[16] S. K. Rao. *Regular Iterative Algorithms and their Implementation on Processor Arrays*. PhD thesis, Stanford University, Stanford, California, 1985.

[17] S. K. Rao. *Systolic Arrays and their Extensions*. Prentice Hall Series, (to appear), 1988.

[18] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proceedings of the IEEE*, To appear in 1988.

[19] S. K. Rao and T. Kailath. What is a Systolic Algorithm? In *SPIE Proceedings*, Real-Time Signal Processing, 1986.

[20] V. P. Roychowdhury and T. Kailath. Analysis and optimal imlementation of regular iterative algorithms with semi-infinite index spaces. 1987. Manuscript under preparation, ISL, Stanford University, Stanford, CA 94305.

[21] V. P. Roychowdhury and T. Kailath. A Nonsystolic Algorithm Implementable on Regular Processor Arrays. *Proc. International Symposium on Circuits and Systems*, San Jose, CA,, May 1986.

[22] V. P. Roychowdhury and T. Kailath. Regular Processor Arrays for Matrix Pivoting Algorithms. *Submitted to Communications of ACM*, Feb. 1988.

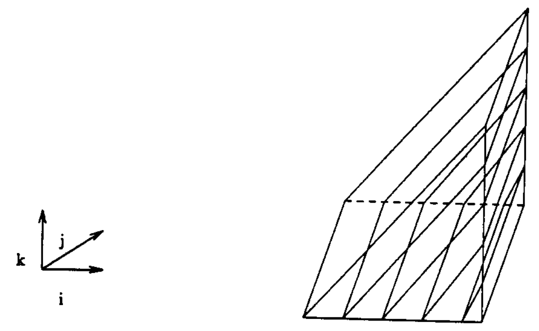

Figure 1: Planes along which projections are done in the index-Space of the RIA for Gaussian elimination with partial pivoting. The architecture derived is shown in 2.

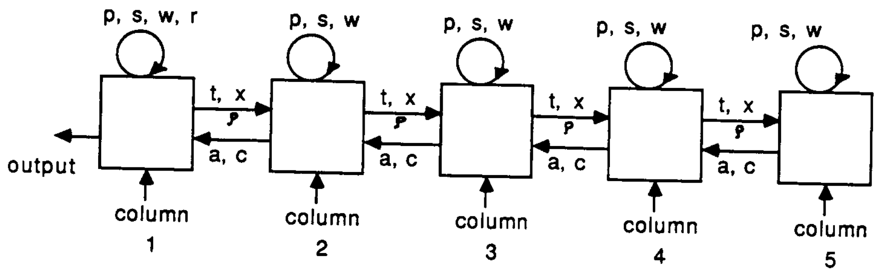

Figure 2: A linear processor array for performing Gaussian elimination with partial pivoting

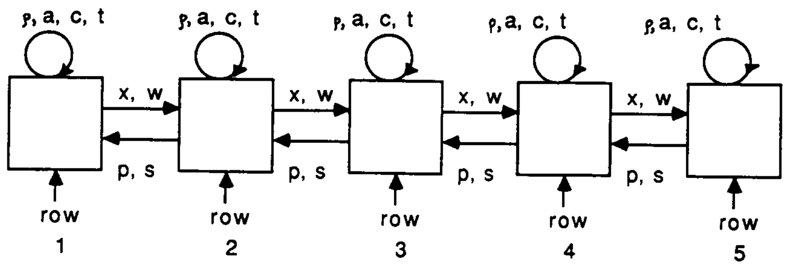

Figure 3: A row-oriented linear processor array for performing Gaussian elimination with partial pivoting.