Parallel Image Processing with the Block Data Parallel Architecture

WINSER E. ALEXANDER, MEMBER, IEEE, DOUGLAS S. REEVES, AND CLAY S. GLOSTER JR.

Invited Paper

Many digital signal and image processing algorithms can be speeded up by executing them in parallel on multiple processors. The speed of parallel execution is limited by the need for communication and synchronization between processors. In this paper, we present a paradigm for parallel processing that we call the block data flow paradigm (BDFP). The goal of this paradigm is to reduce interprocessor communication, and relax the synchronization requirements for such applications. We present the block data parallel architecture which implements this paradigm, and we present methods for mapping algorithms onto this architecture. We illustrate this methodology for several applications including two-dimensional (2-D) digital filters, the 2-D discrete cosine transform, QR decomposition of a matrix, and Cholesky factorization of a matrix. We analyze the resulting system performance for these applications with regard to speedup and efficiency as the number of processors increases. Our results demonstrate that the block data parallel architecture is a flexible, high-performance solution for numerous digital signal and image processing algorithms.

I. INTRODUCTION

Many image processing applications require higher rates of computation than single processors can attain. *Even the faster, superscalar processors promised for the future will not be fast enough to execute many digital signal and image processing applications.* The use of multiple-processor computers has the potential to provide these higher rates of computations. Many attempts have been made to implement high performance multiprocessor systems. Parallel computers designed for a particular application (application specific processors) can often obtain high processor utilization and high throughput. The drawbacks of such computers are their inability to be used for multiple applications, and their high design cost due to a low installed base. General purpose parallel computers are much more flexible, but this flexibility often results in an actual performance that is far short of the performance advertised.

Manuscript received July 10, 1995; revised January 16, 1996. This work was supported by Office of Naval Research Grants N00014-92-J-1201 and N00014-83-K-0138.

The authors are with the Departments of Electrical and Computer Engineering and Computer Science, North Carolina State University, Raleigh, NC 27695–7911 USA.

Publisher Item Identifier S 0018-9219(96)04993-6.

The category of special purpose systems falls in between these two extremes. Special purpose systems are programmable and they are designed to have high performance for a given type of application. The digital signal processor (DSP) is a good example of a special purpose processor. It provides excellent performance on signal processing algorithms such as convolution, filtering, inner product computations, etc. However, it is less suitable as the CPU for a general purpose desktop system.

Most of the high performance systems designed for demanding digital signal and image processing applications have been special purpose SIMD systems [1], [2]. Many of these have used bit serial arithmetic. These systems have proven to give very high performance for many applications. However, they have not totally solved the problem of providing high performance for digital signal and image processing applications because of input/output (I/O) problems, synchronization difficulty as the number of processors increases, and lack of flexibility in adapting to different applications.

The most successful multiprocessor systems for digital signal and image processing either have mesh architectures or are pipelined. Mesh architectures often provide very large speedup after the image has been loaded but overall performance often suffers from I/O limitations. Pipelined machines can accept data at a fast rate and often they can accept it at real-time rates. However, pipelined multiprocessor systems have historically been difficult to program or reconfigure for different tasks. The focus of our research has been to develop a systematic approach for high performance digital signal and image processing that can obtain speedup comparable to mesh architectures while still accepting data at real-time rates comparable to pipelined architectures. Thus we consider the entire problem of implementing image processing applications at high rates including specification of the algorithm, partitioning it for implementation on a multiprocessor system, developing a high performance multiprocessor architecture for image processing applications and evaluating the performance of the application on the multiprocessor architecture.

There are several reasons why a multiprocessor system may not achieve advertised performance on a given application. The most important of these include the following:

- There is a mismatch between the parallelism in the algorithm and the multiprocessor architecture.
- A synchronization bottleneck occurs because some of the processors must wait for results from other processors or from the input device.
- A resource contention problem occurs because two or more processors need to simultaneously use the same system resource. For example, two processors may need to read data from a shared memory at the same time.
- Timing problems due to clock skew, clock distribution, etc. limit the performance of the system as the number of processors increases.
- Programmers find that high performance can only be achieved by careful consideration of the target architecture, and laborious manual optimization of the code.

In this paper, we introduce the block data flow paradigm (BDFP) as an approach to achieve high throughput and efficient multiprocessor execution for many digital signal and image processing applications. Our approach incorporates both a special purpose multiprocessor architecture called the block data parallel architecture (BDPA) and a methodology for programming this architecture. The programming methodology uses a convenient algorithm specification technique, and assists or automates the process of partitioning the computation and scheduling it for efficient parallel execution. We describe the architecture and the programming methodology, followed with a number of examples of the performance that can be achieved with this approach. We conclude that *the BDPA can provide almost linear speedup along with high efficiency for many digital signal and image processing problems.* Although we emphasize the use of the BDPA with the BDFP in this paper the BDFP can be effectively used with many other architectures without a major loss in performance.

Many of the efforts to develop algorithms for high performance digital signal and image processing involve the development of systolic arrays [3], [4] or bit-level pipelining [5]. Our approach does not compete with these approaches because our approach can incorporate these approaches at the processor level. Rather, we emphasize the use of multiple processor systems to achieve high performance.

II. The Block Data Flow Paradigm

We developed the BDFP as a systematic approach to solving the algorithm partitioning problem for many digital signal and image processing algorithms. The BDFP is based on the following principles.

1) The input data is partitioned into large blocks.
2) The processing for a block is assigned to a single processor module (PM). Processing of multiple blocks occurs in parallel on multiple PM's.
3) The processing of a block begins as soon as it is available, as advocated by research on dataflow architectures. No global control is needed to indicate the explicit start and stop times for block processing.
4) Each PM computes the output derived from its assigned input block and transmits this output to the output device as soon as it is ready.
5) PM's which must exchange partial or intermediate results do so asynchronously. A point-to-point (single stage) interconnection network is used for communication.

A PM may be a single processor or a cluster of tightly coupled processors. For example, a PM may be a single application-specific processor, a special purpose processor, a commercial DSP, a systolic array or a general purpose processor. We assume that a PM takes longer to process an input sample than it takes to receive that sample. This guarantees that once a PM starts receiving a block of data, it always has data available until it finishes processing that block.

A. Block Processing

Large scale tasks can be divided into smaller tasks using either an algorithm partitioning strategy or a data partitioning strategy. The BDFP supports the use of the data partitioning strategy at the highest level and it can support the use of the algorithm partitioning strategy at the PM level. With the data partitioning strategy, the whole data set (image or matrix, for example) is divided into data blocks and the computations for each data block are scheduled on different PM's.

We use the term "block processing" to denote the case when data partitioning is used and each PM performs all of the computations necessary to produce the output corresponding to its assigned data block. Block processing was originally very popular to solve the limited memory problem when using early computers for block convolution of images [6]. It is attractive for use in multiprocessor systems because the overhead of data transmission (handshaking, buffer management, etc.) is averaged over a large amount of data and it minimizes the amount of coordination, or synchronization, needed between processors. It is also compatible with message passing protocols that are used by several commercial multiprocessors. This form of coarse-grained parallel processing is a fundamental part of the BDFP.

Block data processing maximizes the opportunity for intermediate computational results to be used locally. Only intermediate results necessary for processing blocks of data assigned to other PM's need to be communicated to other PM's. As a result, a simple interconnection network with point-to-point connections is sufficient. This approach helps reduce interprocessor communication. A large reduction in interprocessor data communication can tremendously improve the overall efficiency of a multiprocessor system.

With an algorithm partitioning strategy, a complex algorithm is decomposed into a sequence of simple operations.

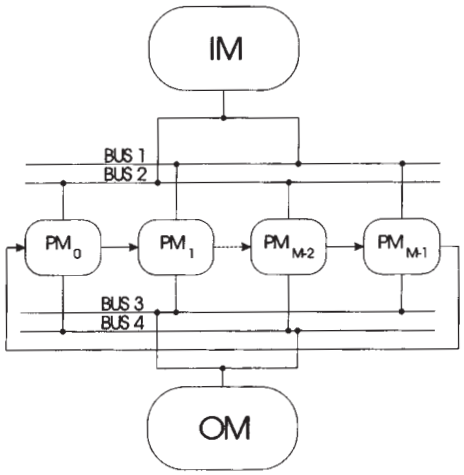


Fig. 1. High-level representation of the BDPA.

Processors cooperate closely in generating the output for a single input sample. Algorithm partitioning is a finer-grained form of parallel processing which can be used within a PM. In Section IV, we present data partitioning and algorithm partitioning as it relates to the BDFP.

B. The Data Flow Transmission Protocol

Data transmission protocols may be categorized into synchronous data transmission protocols and asynchronous data transmission protocols. The synchronous data transmission protocol is fast and simple and there is no handshaking overhead. However, the synchronous data transmission protocol places a tight timing restriction on the system. This can be a problem for large-scale systems. On the other hand, the asynchronous data transmission protocol does not have this timing restriction, but it does have a considerable amount of overhead associated with it. A system to implement the BDFP should use a globally asynchronous data transmission protocol with a large data grain, and a locally synchronous data transmission protocol with small data elements within the PM. This accomplishes two results: 1) it eliminates clock skew problems that result from global synchronization and 2) it reduces overhead due to asynchronous handshaking.

Each PM in a system that implements the BDFP has its own stored program. Once it has all of the required data, it can operate independently of other PM's at its maximum rate. Processing begins as soon as the input becomes available, as defined in the dataflow model of computation. The use of large data blocks reduces the requirement for synchronization and communication between processors. This is the key to achieving high efficiency and near-linear speedup on a multiprocessor system for many applications.

III. The Block Data Parallel Architecture

The BDFP is intended to be a generic approach to partitioning algorithms for implementation on a multiprocessor system. We developed the BDPA to match the requirements of the BDFP. While the BDPA is one example of an architecture that can take full advantage of algorithms that have been partitioned according to the BDFP, other approaches may be used as well. However, the BDFP and the associated BDPA have evolved from our extensive research on developing high performance applications for digital signal and image processing. We discuss the architectural features of the BDPA in this section.

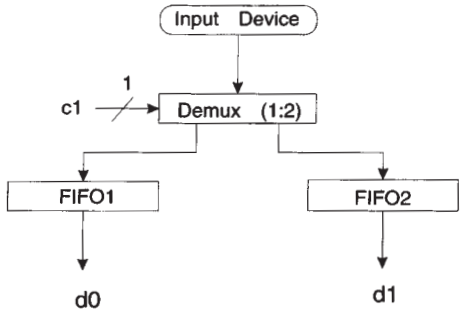


Fig. 2. Conceptual representation of an IM for the BDPA.

The data flow computation model is different from the Von Neumann computation model. Data flow processors are stored-program computers. If sufficient resources are provided, the system can exploit all concurrency present in the program. This approach can be naturally extended to an arbitrary number of processors. It reduces the data, control, and resource dependencies between processors. However, it is difficult to manage the data flow model for a general purpose multiprocessor system [7]. We implemented the data flow paradigm at the processor array level with a large data-block-grain and used a unidirectional ring to connect the PM's together. In addition, each PM has a separate connection to the input module and a separate connection to the output module. With these restrictions, we have successfully implemented the data flow paradigm for the BDPA.

The BDPA is our example architecture for implementation of the BDFP and consists of three major components: an input module (IM), a processor module array (PMA), and an output module (OM). Fig. 1 is a high-level block diagram of the BDPA showing the three primary modules. Buffering of data between all components is accomplished with first-in/first-out (FIFO) buffers. These buffers allow PM's to operate asynchronously, which eliminates the clock distribution problem that adversely affects multiprocessor systems with a large numbers of processors. Input data is transmitted directly from the input device to the PM that will use it. The interprocessor communication is limited to passing necessary intermediate computational results. Output results go directly from each PM to an output device. Thus the BDFP avoids the I/O bottleneck of most multiprocessor systems.

A. Input Module

We assumed that the input is a data stream due to the nature of digital signal and image processing applications. Thus the input data stream may come from an input device such as a camera or the input may be stored on a disk or

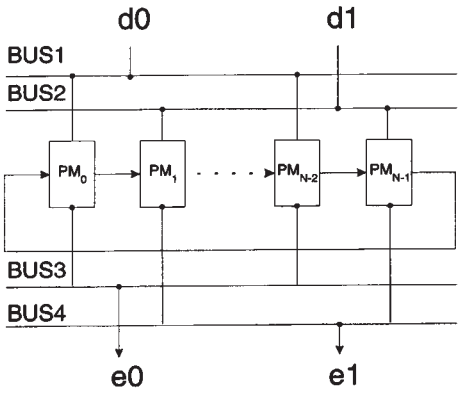


Fig. 3. Conceptual representation of a PMA for the BDPA.

in the main memory of a host system. The IM serves as a buffer between the host system (or an input device) and the processor module array. It converts the input data stream into blocks of data. It maintains a direct input channel to each PM and sends data blocks to each PM through these channels without any interference from other PM's. A block diagram of the IM is shown in Fig. 2.

B. The Processor Module Array

The PMA contains enough PM's to provide the computational power required for the application to be implemented. The interprocessor communication is limited to being local and in one direction to make the implementation of the BDFP easier. All of the PM's have separate input and output channels in addition to the connections between processors. The use of two input buses and two output buses increases the available I/O bandwidth, without significantly increasing the hardware complexity. The IM uses the input FIFO's to sequentially send input blocks to the PM's, and the PM's sequentially send output results to the OM using the output FIFO's. The PM's are divided into two groups: an odd-numbered PM group and an even-numbered PM group. Each PM group is directly connected to one input FIFO and one output FIFO. FIFO buffers are also used to connect adjacent processors in the unidirectional ring. If the data communication is restricted to being in only one direction, then the PM's can skew their operations. This makes it possible to process blocks in a pipelined fashion. Fig. 3 shows a block diagram of a PMA for the BDPA, with two input buses and two output buses.

A PM may itself consist of multiple processors. Thus the BDPA can easily accommodate the use of algorithm partitioning at the PM level. Processors within a PM may utilize various interconnection networks including a mesh, a linear array, a star network, or they can be fully connected. They may also be tightly coupled and can be closely synchronized with each other.

C. The Output Module

The OM collects processing results from each PM and converts the blocks of data into a synchronized output data stream. It also may contain a postprocessing submodule. The postprocessing submodule can be very flexible in order to meet the requirements for different applications. It can contain different function modules for this purpose. For example, the postprocessing submodule may implement different dynamic scaling algorithms for digital signal and image processing. Fig. 4 shows a block diagram of the OM for the BDPA.

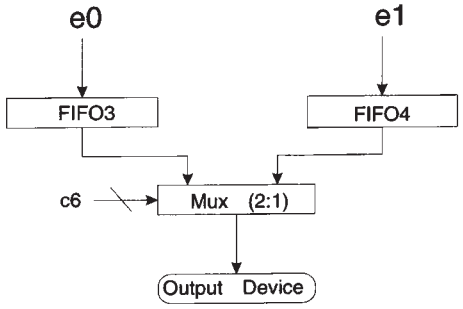


Fig. 4. Conceptual representation of an OM for the BDPA.

The use of the BDFP helps to reduce data storage requirements. In the BDPA, the input data blocks flow into the system and the output data blocks flow out of the system. There is no need to store all of the data or all the entries of a matrix into a BDPA system.

It is sometimes necessary to depart slightly from the BDFP in order to run an application on the BDPA. For example, some algorithms require blocks of input data to be overlapping. Also, some applications may require data communication to be bidirectional, instead of unidirectional. While it is possible to modify the BDPA to accommodate these applications, performance will typically be reduced and/or special hardware modifications may need to be made.

IV. Mapping Algorithms to the BDPA

One of the major hurdles in effective use of parallel computers is the difficulty of programming them. Users generally must understand the target parallel computer architecture, and must carefully write their programs to efficiently use that architecture. This requires considerable effort and usually, the programs that are produced are not portable. In this section we present our methods for automatically mapping algorithms for efficient execution on the BDPA. Our method is general and provides programs with increased portability.

The steps in mapping an algorithm onto a parallel architecture include algorithm specification, partitioning, and scheduling. The specification technique must be flexible and convenient to use for the targeted applications (digital signal and image processing). This specification of the algorithm is used to partition it into separate tasks to be implemented in parallel. The individual tasks are then scheduled for execution at specific times on specific processors.

Our goal is to minimize the sources of overhead for parallel execution: fixed overhead ("startup time"), contention

for shared resources, communication time, synchronization, and uneven processor workload assignment. The impact of startup time is reduced as the problem size becomes larger. Our target application is continuous, real-time image and signal processing, for which the startup time becomes insignificant. The use of FIFO's, one-way communication, and large block sizes greatly reduces the impact of synchronization in the BDFP. The shared resources that could become bottlenecks are the IM, the OM, and the communication links between the PM's. The FIFO's in the IM and OM obviously must be large enough to keep up with the desired input and output rates. The use of dual FIFO's in the IM and OM and dual buses for input and output to the PM's ensure that the data rates in the PMA are lower than the data rate for the input data stream. Blocks of data are formed in the IM and routed sequentially to the PM's using the duality of input buses. In addition, the use of FIFO's between PM's ensures that the system can keep up as long as there are enough processors.

The remaining sources of overhead are excessive communication, contention for the (shared) communication links, and unbalanced workload assignment. In the two sections below, we describe a mapping methodology designed to minimize these problems. Our methodology works at two levels. At the higher level, the initial algorithm specification is transformed into an equivalent state space model which requires less communication and synchronization for parallel execution. This state space model is then examined to find a good data partitioning. At the high level, scheduling of data partitions onto PM's is a relatively simple task. At the lower level, the computation required for each data partition is further partitioned and scheduled onto multiple processors, if necessary. This partitioning is based upon an analysis of the signal flow graph (SFG) for the algorithm. During algorithm partitioning and scheduling the requirements for synchronization and communication are precisely calculated and an attempt is made to minimize their impact.

Our mapping methodology exploits both coarse-grained and fine-grained parallel processing. Coarse-grained parallelism, based upon data partitioning, is always used. Fine-grained processing is then resorted to if data partitioning alone does not achieve the desired processing rate. We now describe each of these levels in detail.

A. State Space Method

We use a generalized state space approach for partitioning the algorithm of interest at the high level. The state space model matches our concept of using data partitioning at the high level in that the states are updated and the output is computed as linear combinations of the most recent states and the current input. Although the assignment of the state variables is arbitrary in general, states can be chosen to minimize the data communication requirements between processors. In this section, we show how to represent digital signal and image processing algorithms in the state space model.

Many image processing algorithms fall into the category of discrete linear shift-invariant (DLSI) systems. Any algorithm that can be represented as a set of finite difference equations with constant coefficients meets the criteria for a DLSI system. For example, filter kernels or finite impulse response (FIR) filters for image processing involve computing the output as a weighted average of pixels from neighboring input pixels. Such a filter may be expressed in finite difference form as

$$g(n_1, n_2) = \sum_{j_1=-M_1}^{M_1} \sum_{j_2=-M_2}^{M_2} b(j_1, j_2) f(n_1 - j_1, n_2 - j_2). \tag{1}$$

For example, $M_1 = M_2 = 2$ for the case of a 5×5 kernel. In a similar way, we can represent 2-D infinite impulse response (IIR) filters in difference equation form (using weighted values of previously computed outputs).

The state space representation has been widely used for many years to represent digital control systems [8]. The state space model converts the difference equation model to a matrix-vector model where the output can be represented as a linear combination of the current input and the most recently computed states in each tuple. Similarly, the states are updated as linear combinations of the current input and the most recently computed states in each tuple. This provides an advantage for the computational model because all required data is either a current input or a recently computed state variable. Thus it is easier to develop a computational model that has low data communication requirements when using the state space representation. This is easy to see for the one-dimensional case because all of the states can be stored in a local memory and only the new input needs to be communicated to perform the computations for the new output.

We can use the state space model as a systematic approach to partition DLSI systems for implementation on a multiprocessor system. For example, an image can be considered to be a 2-D signal with the indices for the rows and columns representing the tuples. Indexing along the columns can be considered to be the horizontal tuple and indexing along the rows the vertical tuple. Thus a horizontal delay represents a delay of one pixel along the current row and a vertical delay represents a delay of a row to access the pixel in the previous row along the same column. Our data partitioning method based on the state space model transforms the algorithm so that the update of the current state and the current output involves only the current input and the most recent state variables for each tuple. In this case, the update of the state and computation of the output involves only the current input, the horizontal state variables, and the vertical state variables.

After mapping the algorithm into the state space representation, one can obtain a partitioned algorithm with low data communication requirements in the following way. All of the computations for a single row of the image can be scheduled onto the same PM, and processing of different rows can be scheduled on different PM's. State variables (i.e., intermediate results) that are delayed by one horizontal

delay are used by the same PM, and state variables that are delayed by one row are transferred to the PM handling the next row. If the image is acquired in a raster scan fashion or is input by row, then the state variables to be transferred to the next PM are not needed until after the next row of data is input. This skewing of row computations matches the sequential input of the rows very well.

We gained insight into these data communication patterns from our research on implementing 2-D digital filters on a multiprocessor system [9]. We later generalized this approach to the use of block processing [10]. A block of data may be a row or a column of an image, a row or column of a matrix, or it may be a subimage or submatrix depending on the requirements of the application. Partitioning of the state space representation minimizes the amount of intermediate results that must be communicated between PM's. In addition, the state space method ensures that data transfer is localized if adjacent blocks are scheduled on adjacent PM's in the unidirectional ring.

B. Mapping DLSI Systems to the State Space Model

In this section, we show how to map a DLSI system onto the BDPA as an example. The method is appropriate for any algorithm that can be represented by a finite difference equation or a SFG. Our goal was to achieve real-time filtering of a continuous sequence of images.

The state space representation can be given in matrix form. It is very convenient to do so because the representations are more concise.[1] We now discuss the state space representation of the general 2-D DLSI system with quarter-plane support.

The general-order, causal 2-D finite difference equation with quarter-plane support is given by [6]

$$g(n_1,n_2)=\sum_{j_1=0}^{L_1}\sum_{j_2=0}^{L_2} b(j_1,j_2)f(n_1-j_1,n_2-j_2) - \sum_{\substack{j_1=0\\ j_1+j_2>0}}^{L_1}\sum_{j_2=0}^{L_2} a(j_1,j_2)g(n_1-j_1,n_2-j_2). \tag{2}$$

The parameters $a(j_1,j_2)$ and $b(j_1,j_2)$ in (2) are coefficients which determine the characteristics of the algorithm. Since the coefficients can take on arbitrary values, this equation can represent many 2-D problems including 2-D kernel or convolution filters (FIR), 2-D IIR digital filters, image processing operations such as averaging a region of pixels, simulation, control systems, etc.

For the 1-D case, a similarity transformation can be used to optimize the state space representation [8]. However, there is a fundamental problem in extending this concept to the 2-D case because an arbitrary bivariate transfer function cannot be factored into distinct poles and zeros and cannot be expanded into partial fractions [6]. Thus, approaches which involve factorization or partial fraction expansion are not generally extendible to 2-D systems due to the lack of a fundamental theory for 2-D systems.

The state space approach can however be used for mapping 2-D DLSI systems [11], [12]. Roesser's state space model for 2-D DLSI systems is perhaps the most widely accepted model [11]. This model provides for the update of the next state for a set of vertical state variables and a set of horizontal state variables as a linear combination of the current input and the present vertical and horizontal state variables. The output is also a linear combination of the present vertical and horizontal state variables and the current input

$$\begin{bmatrix} S_H(n_1+1,n_2) \\ S_V(n_1,n_2+1) \end{bmatrix} = \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} \end{bmatrix} S(n_1,n_2) + \begin{bmatrix} \mathbf{B_1} \\ \mathbf{B_2} \end{bmatrix} [f(n_1,n_2)]$$
$$[g(n_1,n_2)] = [\mathbf{C_1}\ \mathbf{C_2}]S(n_1,n_2) + \mathbf{D}[f(n_1,n_2)] \tag{3}$$

where

$$S(n_1,n_2) = \begin{bmatrix} S_H(n_1,n_2) \\ S_V(n_1,n_2) \end{bmatrix}. \tag{4}$$

Roesser's state space model is based upon assigning state variables to the output of the delay elements. However, the assignment of state variables is arbitrary and we find it more convenient to assign the state variables to the input of the delay elements. This makes the state space representation compatible with the eventual hardware implementation because a state variable identifies a parameter which must be stored for a later computation. This alternate choice for the state variables is equivalent to the parameter substitution

$$\begin{aligned} Q_H(n_1,n_2) &= S_H(n_1+1,n_2) \\ Q_V(n_1,n_2) &= S_V(n_1,n_2+1). \end{aligned} \tag{5}$$

With this substitution, the indices for the modified state vectors are the same as those for the current input. Thus the modified model is conceptually simpler because it more closely resembles the finite difference equation model. It also simplifies our later derivations.

We can combine the vertical state variables and the horizontal state variables into a state vector for a given location in the 2-D array. Thus

$$Q(n_1,n_2) = \begin{bmatrix} Q_H(n_1,n_2) \\ Q_V(n_1,n_2) \end{bmatrix}. \tag{6}$$

This state vector and the next output can be computed using a linear combination of the current input, the most recent vertical state variables, and the most recent horizontal state variables. This revised model is equivalent to Roesser's original model.

[1] However, the coefficient matrices are typically sparse. In the actual mapping process, it is better to implement individual equations rather than use matrix operations.

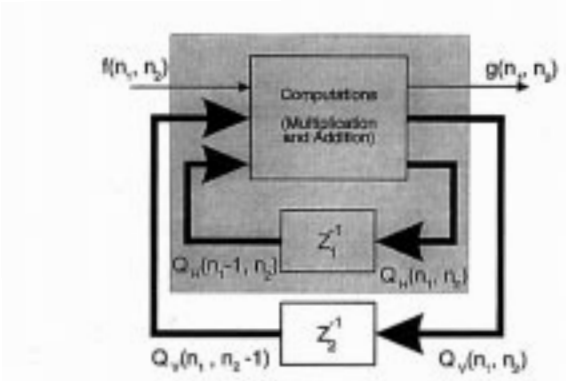


Fig. 5. Two-dimensional generalized finite state machine.

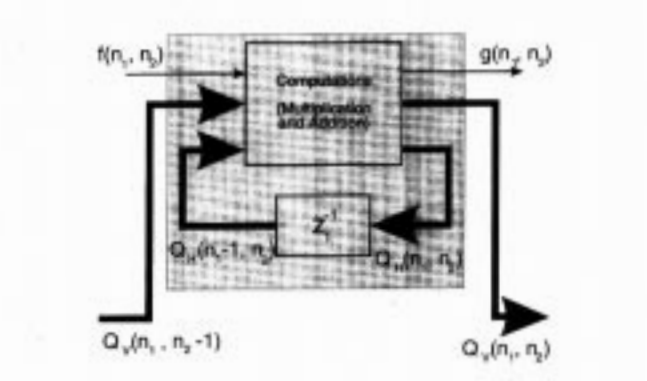


Fig. 6. Two-dimensional generalized finite state machine for multiprocessor system.

This modified state model for the causal 2-D DLSI system with quarter plane support is given by

$$\begin{aligned} Q(n_1,n_2) &= \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} \end{bmatrix} \begin{bmatrix} Q_H(n_1-1,n_2) \\ Q_V(n_1,n_2-1) \end{bmatrix} \\ &\quad + \begin{bmatrix} \mathbf{B_1} \\ \mathbf{B_2} \end{bmatrix} [f(n_1,n_2)] \\ [g(n_1,n_2)] &= [\mathbf{C_1} \quad \mathbf{C_2}] \begin{bmatrix} Q_H(n_1-1,n_2) \\ Q_V(n_1,n_2-1) \end{bmatrix} \\ &\quad + \mathbf{D}[f(n_1,n_2)]. \end{aligned} \tag{7}$$

In (7), $Q_H(n_1,n_2)$ is a column vector whose elements are the current values of the state variables for the horizontal processing direction corresponding to the index n_1. $Q_V(n_1,n_2)$ is a column vector whose elements are the current values of the state variables for the vertical processing direction corresponding to the index n_2. Note that (n_1-1,n_2) implies a delay in the horizontal direction and (n_1,n_2-1) implies a delay in the vertical direction. $\mathbf{A_{11}}$, $\mathbf{A_{12}}$, $\mathbf{A_{21}}$, $\mathbf{A_{22}}$, $\mathbf{B_1}$, $\mathbf{B_2}$, $\mathbf{C_1}$, $\mathbf{C_2}$, and $\mathbf{D}$ are appropriate coefficient matrices such that (2) and (7) are equivalent.

In order to show the relationship between the 2-D state space model and the computational model, we represent the 2-D state space model as a linear finite state machine in Fig. 5. In this computational model, the horizontal state variables $Q_H(n_1,n_2)$ are stored inside the processor and the vertical state variables $Q_V(n_1,n_2-1)$ are sent to external memory. The reason for this will become clear when we discuss implementing the algorithm on a multiprocessor system. The output is also computed and sent to the output device. Thus the processor takes $f(n_1,n_2)$ and $Q_V(n_1,n_2)$ as inputs and computes $Q_H(n_1,n_2)$, $Q_V(n_1,n_2)$, and $g(n_1,n_2)$. It stores $Q_H(n_1,n_2)$ internally for the next computation, sends $Q_V(n_1,n_2)$ to external memory for use one row later and sends $g(n_1,n_2)$ to the output device.

Our scheme for implementing the 2-D state space representation with a multiprocessor system involves scheduling the computations for each row on a different processor. Thus in this scheme, the vertical state variables from the previous row, $Q_V(n_1,n_2-1)$ are received from the processor assigned to perform the computations for the previous row and the vertical state variables $Q_V(n_1,n_2)$ are sent to the processor assigned the computations for the next row. This modification is shown in Fig. 6.

We can extend the discussion above to cover nonlinear systems by considering the modifications required in the state space representation in order to represent nonlinear systems. This is important because a large number of important applications for image processing can be represented as nonlinear systems. The median filter is a good example since the output is the median for a region of neighboring pixels in the image.

Thus we see that the direct application of the state space representation along with scheduling the computations for each row on a different processor can lead to a multiprocessor system design with low data communication requirements. In (7), the system is linear because the elements for all of the coefficient matrices are constants. In addition, it shows the state vectors and the input being multiplied by these matrices. This is a direct result of the original equation being a finite difference equation with constant coefficients as shown in (2). If we define these matrices as functions of the independent variables and define a general operator $\Gamma[\ ]$ that operates on a matrix and vector to obtain an output vector, we can represent the 2-D nonlinear state space model as

$$\begin{aligned} Q_H(n_1,n_2) &= \Gamma[\mathbf{A_{11}}(n_1,n_2), Q_H(n_1-1,n_2)] \\ &\quad + \Gamma[\mathbf{A_{12}}(n_1,n_2), Q_V(n_1,n_2-1)] \\ &\quad + \Gamma[\mathbf{B_1}(n_1,n_2), [f(n_1,n_2)]] \\ Q_V(n_1,n_2) &= \Gamma[\mathbf{A_{21}}(n_1,n_2), Q_H(n_1-1,n_2)] \\ &\quad + \Gamma[\mathbf{A_{22}}(n_1,n_2), Q_V(n_1,n_2-1)] \\ &\quad + \Gamma[\mathbf{B_2}(n_1,n_2), [f(n_1,n_2)]] \\ [g(n_1,n_2)] &= \Gamma[\mathbf{C_1}(n_1,n_2), Q_H(n_1-1,n_2)] \\ &\quad + \Gamma[\mathbf{C_2}(n_1,n_2), Q_V(n_1,n_2-1)] \\ &\quad + \Gamma[\mathbf{D}(n_1,n_2), [f(n_1,n_2)]]. \end{aligned} \tag{8}$$

From these equations, we see the primary modification we need to make in our 2-D linear finite state machine model is that the processor must be able to perform the nonlinear operations depicted by $\Gamma[\ ]$. Another complexity is that the coefficient matrices may vary as functions of the row or column numbers. Normally, this means that the coefficients must also be computed inside the processor. Typical

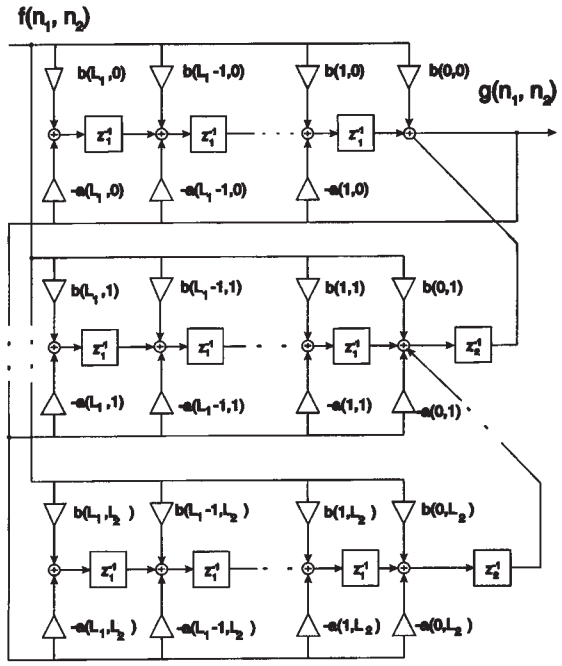


Fig. 7. Block diagram representation of a 2-D system.

nonlinear operations that lead to nonlinear systems include sorting, comparisons, division, and logical operations. The data communication patterns that we developed for the DLSI system will also work for nonlinear systems. We later present our results for mapping the Cholesky factorization, QR decomposition and back substitution as examples that fit this model.

We further generalized the above discussion to include the block state model for computing [10], [13], [14]. For this model, the input is partitioned into blocks and the computations for each block are scheduled for different processors. The vertical state variables $Q_V(n_1, n_2)$ are replaced by block state variables $Q_P(n_1, n_2)$. Otherwise, the high level computational model is the same as shown in Fig. 6. We later present the mapping of the 2-D discrete cosine transform (DCT) as an example that fits this model.

C. A 2-D Example of the State Space Method

We now consider the state space representation of the 2-D IIR digital filter as an example. The state space representation for a given 2-D DLSI system is not unique. In addition, the problem of defining a representation with the minimum number of states has not been solved [15]. Our intended application for this example is real-time filtering of images where the image comes from a camera or other video source or is stored by row in memory. For this application, a horizontal delay represents a storage of one word while a vertical delay represents a storage of an entire row of data. This is true because we must keep all of the vertical state variables for a row to be able to have access to a state variable from the previous row each time we bring in a new pixel. Therefore we selected a computational form that minimizes the number of vertical delays. We then assigned state variables to the inputs of the delay elements to obtain the state space representation. This procedure can be applied to obtain a state space representation from any SFG although it is not guaranteed to have a minimum number of states.

Fig. 7 gives a block diagram representation of a 2-D filter partitioned to have a minimum number of vertical delays (z_2^{-1}). Note that the number of vertical delays is the same as the order of the filter in the z_2 variable, which is the minimum possible number. We can obtain the desired state space representation by assigning a horizontal state variable to the input of each of the horizontal delay blocks (associated with the z_1 variable) and a vertical state variable to each of the vertical delay blocks (associated with the z_2 variable). We then write the resulting equations in matrix form as given in (7).

Fig. 8 gives a section of the block diagram of the 2-D DLSI system having one horizontal delay and one vertical delay. Assigning state variables as described above, the typical vertical state equation for the 2-D DLSI system can be represented as

$$\begin{aligned} q_{2,j_2}(n_1,n_2) &= b(0,j_2)f(n_1,n_2) - a(0,j_2)g(n_1,n_2) \\ &\quad + q_{1,I_1}(n_1-1,n_2) + q_{2,j_2+1}(n_1,n_2-1) \\ &\quad 1 \le j_2 \le L_2 \\ &\quad I_1 = j_2L_1 + 1 \\ q_{2,L_2+1}(n_1,n_2-1) &= 0. \end{aligned} \tag{9}$$

In a similar way, the typical horizontal state variable is given by

$$\begin{aligned} q_{1,I_1}(n_1,n_2) &= b(j_1,j_2)f(n_1,n_2) - a(j_1,j_2)g(n_1,n_2) \\ &\quad + q_{1,I_1+1}(n_1-1,n_2) \\ &\quad 1 \le j_1 \le L_1 - 1;\ 0 \le j_2 \le L_2 \\ &\quad I_1 = j_2L_1 + j_1. \end{aligned} \tag{10}$$

$$\begin{aligned} q_{1,I_1}(n_1,n_2) &= b(j_1,j_2)f(n_1,n_2) - a(j_1,j_2)g(n_1,n_2); \\ &\quad j_1 = L_1; \quad 0 \le j_2 \le L_2. \\ &\quad I_1 = j_2L_1 + j_1. \end{aligned} \tag{11}$$

The output equation is given by

$$\begin{aligned} g(n_1,n_2) &= b(0,0)f(n_1,n_2) + q_{1,1}(n_1-1,n_2) \\ &\quad + q_{2,1}(n_1,n_2-1). \end{aligned} \tag{12}$$

The above equations are still not in the state space form because the horizontal and vertical state variables are defined in terms of the output. Therefore, we need to substitute for the output in these equations to obtain a state space representation where each state variable and the output is defined in terms of the current input and the most recent values of the state variables. Thus we obtain

$$\begin{aligned} &q_{1,I_1}(n_1,n_2) \\ &\quad = [b(j_1,j_2) - a(j_1,j_2)b(0,0)]f(n_1,n_2) \\ &\quad\quad - a(j_1,j_2)[q_{1,1}(n_1-1,n_2) + q_{2,1}(n_1,n_2-1)] \\ &\quad\quad + q_{1,I_1+1}(n_1-1,n_2); \end{aligned}$$

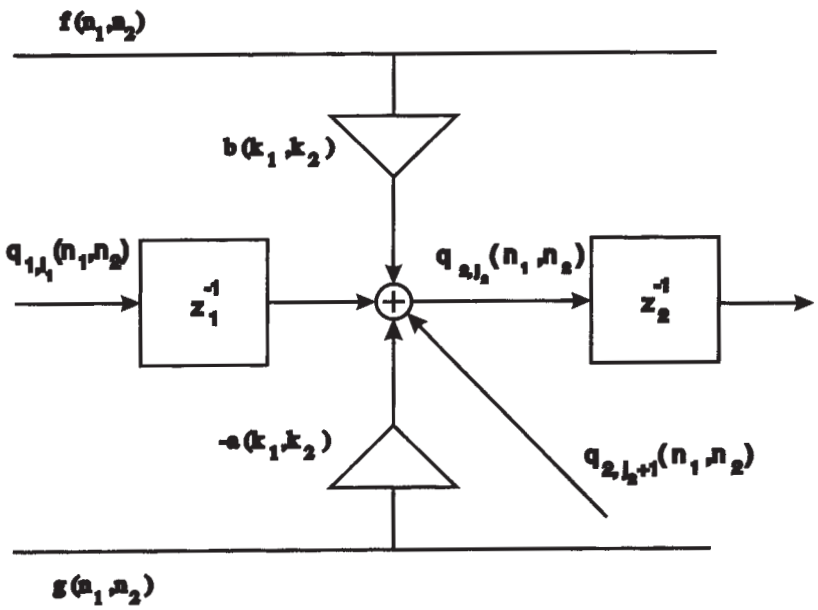


Fig. 8. A section of the block diagram of a 2-D DLSI system having one horizontal delay and one vertical delay.

$$1 \le j_1 \le L_1 - 1;\ 0 \le j_2 \le L_2 \tag{13}$$
$$I_1 = j_2 L_1 + j_1.$$
$$\begin{aligned} q_{1,I_1}(n_1, n_2) &= [b(j_1, j_2) - a(j_1, j_2) b(0,0)] f(n_1, n_2) \\ &\quad - a(j_1, j_2)[q_{1,1}(n_1 - 1, n_2) + q_{2,1}(n_1, n_2 - 1)]; \end{aligned}$$
$$j_1 = L_1; \quad 0 \le j_2 \le L_2. \tag{14}$$
$$I_1 = j_2 L_1 + j_1.$$
$$\begin{aligned} q_{2,j_2}(n_1, n_2) &= [b(0, j_2) - a(0, j_2) b(0,0)] f(n_1, n_2) \\ &\quad - a(0, j_2)[q_{1,1}(n_1 - 1, n_2) + q_{2,1}(n_1, n_2 - 1)] \\ &\quad + q_{1,I_1}(n_1 - 1, n_2) + q_{2,j_2+1}(n_1, n_2 - 1) \end{aligned}$$
$$1 \le j_2 \le L_2 \tag{15}$$
$$I_1 = j_2 L_1 + 1$$

$$q_{2,L_2+1}(n_1, n_2 - 1) = 0. \tag{16}$$

As an example, a second order filter ($L_1 = L_2 = 2$) requires six horizontal state variable equations, two vertical state variable equations and one output equation.

D. The Order Graph Method

In this section, we describe a method of partitioning and scheduling that exploits the algorithm's potential for fine-grained parallel processing. This method is intended to be used to improve a parallel execution schedule that is not fast enough for a given application after using the high level block state space approach described above. This method is based upon the use of algorithm partitioning to develop a schedule for processors in a PM.

After data partitioning, each PM has been assigned a block of data and a set of computations to perform on that block of data. Our approach is to develop a SFG for this set of computations. If the PM consists of several closely coupled processors, this SFG can be partitioned and scheduled for execution in parallel on these processors.

There are many techniques for mapping algorithms to parallel processors, including parallelizing compilers [16]–[19], loop optimization [20]–[24], targeting of recurrence equations to systolic arrays [25]–[29], heuristic scheduling [30]–[33], and flowgraph manipulation [2], [34]–[39]. Some recent surveys of these and other methods can be found in [40]–[44]. Quantitative comparison of these methods is difficult, since they are targeted for different classes of machines, use varying specification techniques, and (most importantly) usually fail to report detailed results for any standard benchmarks. Few of them have all of the desired characteristics for signal and image processing:

- use of a familiar and convenient specification technique;
- fully automated algorithm partitioning and scheduling;
- scalability to large problems;
- exploitation of fine-grained parallel processing;
- reduction of communication overhead;
- consideration of network topology;
- practicality of implementation;
- retargetable for lots of different parallel architectures;
- optimization of processing rate and/or number of processors; and
- acceptable running time.

The method closest to our own is the cyclostatic realization method [2], [34], [35], which has some (but not all) of the desired characteristics, and is intended for parallel signal and image processing.

The computation to be partitioned is specified by a SFG. The SFG shows (by directed edges) the sequence in which operations (indicated by operator nodes) must be performed, and on what clock cycle (as indicated by delay nodes). The user also specifies the desired delay T_d between successive input samples (or, alternatively, the desired rate of input processing $R_d = 1/T_d$), and/or the available number of processors, P_d. Note that the minimum number of clock cycles which is possible for this SFG (as determined by the longest delay path between two consecutive delay nodes) imposes a lower bound on T_d. If T_d is less than this, the SFG would have to be restructured (using, for instance, retiming [45]) so that the minimum clock rate is no greater than T_d.

A flowchart of the order graph method is shown in Fig. 9. The major steps in this method:

1) preprocess the SFG to minimize interprocessor communication;
2) generate different partitions for the graph in a way that heuristically balances the workload of the processors;
3) generate schedules for each candidate partition; and
4) validate the correctness of the schedule, determine the impact of interprocessor communication, and compare the resulting sampling rate and number of processors to the desired rate and number of processors.

Each of these is now described in more detail.

When SFG's are partitioned for parallel execution, splitting a feedback loop (i.e., a cycle in the graph) can result in excessive communication and synchronization overhead. The order graph method preprocesses the SFG to produce a directed acyclic graph (DAG) to avoid this. The first step is finding all the strongly connected components[2] in the graph. This step can be done in linear time [46]. A DAG is produced from the SFG by replacing each strongly connected component with a single node. The incoming edges of this new node are the union of the incoming edges of the strongly connected component, and likewise for the outgoing edges. The computational delay of this new node is the sum of the computational delays of all nodes in the strongly connected component. The DAG no longer captures the exact timing dependencies which are shown in the original SFG. For this reason, it is used solely for generating candidate partitions of the SFG. The benefit is that feedback loops are not split for execution on multiple processors. Once the partitions have been generated from the DAG, scheduling these partitions and validating the schedule is done from the original SFG, so no timing dependencies are overlooked.

It may be that the computational delay of a strongly connected component exceeds the desired sampling interval T_d. This would mean that algorithm partitioning fails to achieve the user-specified performance bound, since a single node is not normally split for parallel execution.[3] This is accommodated by converting the replacement node back into the original graph of the corresponding strongly connected component, removing a feedback edge from this graph, identifying the new strongly connected components, and creating a new DAG from these components. This process continues until no node in the DAG has a computational delay that exceeds T_d. We use heuristics to determine the order for which replacement nodes are processed (when more than one has a delay exceeding T_d), and the order in which feedback edges are removed. Details can be found in [47].

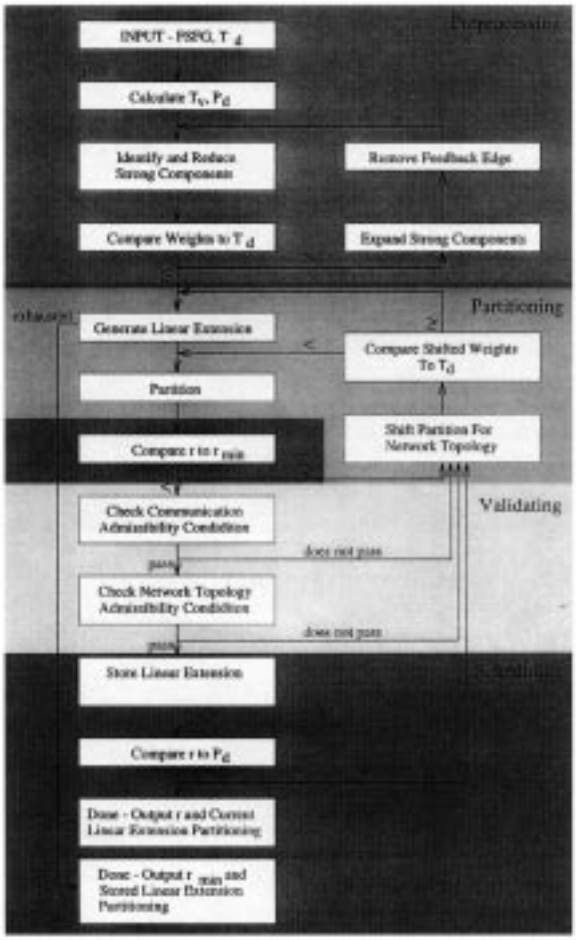


Fig. 9. Flowchart of the order graph method.

The output of the preprocessing stop is a DAG which shows the precedence of operations that must be performed. A linear extension is a sequential ordering, or list, of the nodes in a DAG that preserves the precedence indicated by the edges of the graph. For example, for a graph with nodes $V = \{1, 2, 3, 4, 5\}$ and the edges $1 \rightarrow 3$, $2 \rightarrow 3$, $2 \rightarrow 4$, $3 \rightarrow 5$, $4 \rightarrow 5$, the possible linear extensions are [1,2,3,4,5], [2,1,3,4,5], [1,2,4,3,5], [2,1,4,3,5], and [2,4,1,3,5]. The generation of all linear extensions from the DAG is accomplished using the algorithm found in [48].

[2] A strongly connected component in a directed graph contains either a single node or all the nodes involved in a feedback loop.

[3] Although our method allows this to be done if the user so desires.

This algorithm has complexity $O(|V|t)$ where $|V|$ is the number of nodes and t is the number of linear extensions which satisfy the precedence conditions.

Once a linear extension is found, it is partitioned for parallel execution. A set of partitions is found by processing the linear extension from left to right, building partitions in a greedy fashion. Let partition p start with the node in the ith position of the linear extension, and let the weight of this partition be equal to the computational delays of all the nodes it includes. Partition p includes all nodes from the ith to the jth position such that the weight of p is less than or equal to T_d, and the weight of $p + d_{j+1}$, where d_k is the delay of node k, exceeds T_d. For the linear extension [1,2,3,4,5], if each node has a computational delay of one and the desired input sampling delay is two, the greedy partitioning procedure gives [1,2], [3,4], [5].[4]

The aim of this heuristic is to minimize the number of partitions, and to find partitions whose execution times are approximately equal. This minimizes the number of processors required for parallel execution, and balances the workload of the processors. All linear extensions of the DAG are generated exhaustively, and for each linear extension, all possible potential solutions are found using the greedy heuristic. This process stops when the first potential solution is found that is valid and meets the performance goals, or when no such solution exists. A potential solution is rejected if the number of partitions in it exceeds the user-specified bound on the number of processors available for use. Our method of generating partitions ensures that all precedence and timing constraints are properly met.

If there are p partitions to be executed in parallel on p processors, there are $p!$ possible schedules. Schedules may be generated exhaustively until a valid schedule is found, or no such schedule exists. The order graph method focuses on two potential schedules: a fully pipelined schedule and a strictly parallel schedule. The communication requirements must also be considered to determine if a schedule is valid. We assume that communication and computation are fully overlapped (as is true, for instance, for the Inmos Transputer and the Texas Instruments TMS320C40), that receiving or sending each data word requires some fixed time c (specified by the user), and that communicating w words (either sending or receiving) requires total time $w \cdot c$. Each solution's total communication time must not exceed the desired sampling delay T_d, or the solution is invalid.[5]

The total communication time per partition can be tabulated by inspection of the partitioned SFG. This process is complicated by the fact that one processor can be scheduled to execute multiple partitions (at different times) for the same input sample. In this case, the data transmitted between partitions does not require communication between different processors, and so should not affect the total communication time for those partitions. A final validity check requires that the usage per input sample for each communication link does not exceed the desired sampling period T_d. We assume that the network topology specified by the user contains exactly one path between each source and destination processor. From the schedule, the amount of communication per input sample between each pair of processors is tabulated. This communication time is added to the load for each link on the path that connects the two processors. In this way, contention for communication links is considered and minimized.

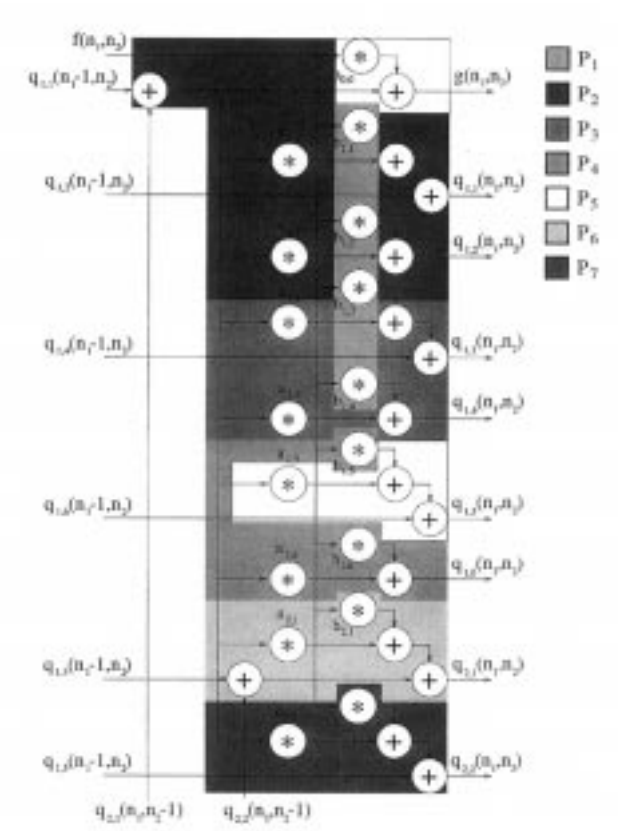


Fig. 10. Signal flow graph for the second-order 2-D IIR filter, with mapping generated by the order graph method.

We have developed a prototype tool to implement the order graph method. An example of mapping a 2-D IIR filter for parallel execution on seven processors is shown in Fig. 10. The details of the algorithm for this example are given in Section IV-C. In this example, each multiply requires two time units (or computational cycles) and each addition requires one time unit, and communication time per word c is one time unit. Feedback paths (connecting $q_{1,3}(n_1, n_2)$ and $q_{1,3}(n_1 - 1, n_2)$, for instance) are omitted for simplicity. Our results show that a PM with seven processors can solve all of the required equations using eight computations cycles per pixel. Our results further show that this can be done with the processors connected as specified for the BDPA. We have used our tool for mapping a variety of algorithms to the BDPA, including a fourth-

[4] Our method is more flexible than this example indicates. The process of partitioning a linear extension can begin with *any* node in the extension, not just the first one. Partitioning proceeds from left to right, with wraparound from the end of the extension to the start of the extension [47].

[5] In addition, if processors execute synchronously the inputs for a partition must arrive at the correct time. This can also be checked [47].

order Jaumann wave digital filter, a fourth-order all-pole lattice filter, a 16-point FIR filter, and a 1-D second-order IIR filter. The running time for the tool is generally on the order of a few seconds of CPU time.

In summary, the order graph method can be used to automatically partition and schedule a SFG, using fine-grained parallel processing. It considers the effects of the network topology, and minimizes the communication overhead to the extent possible. It is possible to optimize the sampling rate or the number of processors needed, or to meet user-specified bounds on one or both. The method balances the workload of processors heuristically. Although potentially expensive to use for very large signal flow graphs or large numbers of processors, we have found it to be quite practical for actual use on important signal and image processing problems.

V. Methods of Performance Evaluation

Our goal in mapping algorithms to the BDFP and scheduling them on the BDPA is to achieve a high performance as determined by maximum possible input data rate, speedup, and efficiency as we add more and more processors. In our early work, we used analytical methods to determine performance [9], [49]–[53]. The drawbacks of this approach are that it can be quite difficult to develop a good analytical model, and there is always the possibility that assumptions and oversights will cause important results to be overlooked. We have developed several tools to enable us to do a more detailed analysis and verification of our ideas. These tools are the following:

- `sogae` [54]—a dataflow tool for doing fast, approximate analysis of proposed parallel algorithms;
- `erg` [55]—a detailed simulator for doing precise analysis of proposed algorithms, and validating the correctness of the mapping; and
- *a prototype hardware system*—for final demonstration of a proposed parallel algorithm, in actual operation.

Each of these methods has something to offer, in terms of ease of use, speed of execution, or degree of precision. We now describe each one in turn.

`sogae` [54] models a parallel architecture as a set of processors connected by point-to-point communication links (as in the BDPA). Each processor has a set of named queues (FIFO's); a queue corresponds to an input, an output, or a register used in the computation of an image or signal processing algorithm. For each processor there is an input matrix whose columns specify the order in which data values are needed from the queues, and whose rows show which data values are used on a particular clock cycle. As an example, for a processor with two queues, where queue #1's data is needed on cycles one and three, and queue #2's data is needed on cycles two and three, the corresponding matrix R would be

$$R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}. \tag{17}$$

Fig. 11. Creating the `Erg` simulator from the descriptions of the algorithm and architecture.

An output vector for each processor specifies where (i.e., which queue) the output produced on a particular cycle is intended to go. The set of queues, and the input matrix and output vector, are determined directly from the SFG and the parallel algorithm mapping for the flow graph. An assumption made in this simulator is that data values are produced and stored in the queues in the order in which they will be needed by the processor.

A simulation run models the consumption and production of data items in the way specified by the input matrices and output vectors. If a processor's input matrix indicates the use of a data item from a queue which is empty, the simulated execution of that processor's task blocks until a data item has been put into the queue. `sogae` outputs several pieces of useful information to the user, including statistics on queue lengths and total number of simulated cycles to complete a parallel computation. In addition, a graphical interface allows the user to watch and interactively query the state of the processors during the progress of the simulation. Note that `sogae` only models the timing of computations, and does not produce the actual results that would be generated by those computations.

`erg` [55] is intended to be a more precise (but slower) simulator for verifying both the detailed performance of an algorithm, and the correctness of the results produced by that algorithm. The user of `erg` describes both the architecture of the parallel computer, and the program executed by each processor. These descriptions can both be written in the C programming language for convenience. The descriptions are compiled, along with the simulation event queue manager, into a simulator which can be directly executed (i.e., not interpreted) on the host machine. A picture of this process is shown in Fig. 11. Other parallel computer simulators which implement some of the same ideas are described in [56] and [57].

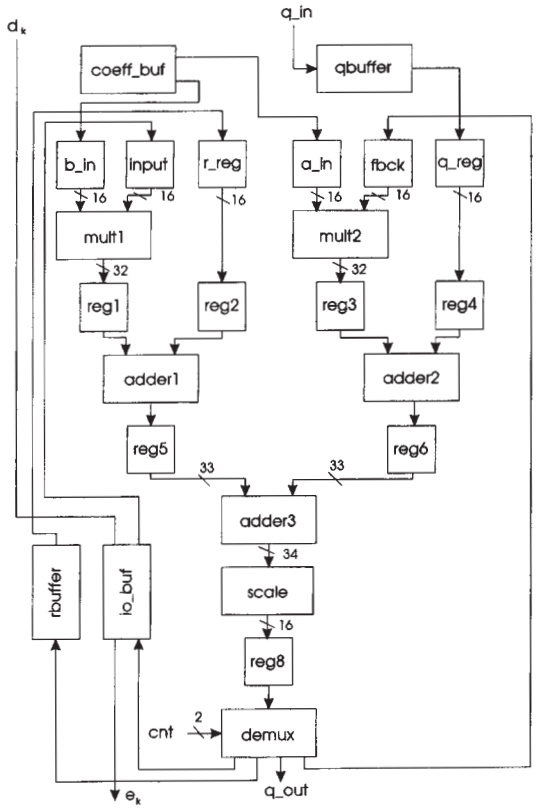


Fig. 12. Application specific processor for the 2-D IIR digital filter.

The architecture description specifies the time required for each type of operator (addition, multiplication, condition checking, etc.), the network topology, and the sizes of all queues. The algorithm description specifies the function (the actual C code) executed by each processor, the function of input and output devices, and the function of the IM and the OM.

The `erg` simulation produces a set of performance statistics and the output that would be produced by an actual system. The output is used to verify that parallel execution produces the same result that would be produced by execution of the algorithm on a single processor system. The statistics include processor utilization, maximum queue lengths, and total processing time. `erg` is a valuable analysis tool, and has helped to clarify many of our ideas about algorithm mapping. Most of the performance results described in the next section have been produced using `erg`.

A final method of performance analysis and verification is to map algorithms to a real system and measure their performance on real data. For this purpose, we are constructing a prototype parallel computer consisting of high-performance DSP chips (the Texas Instruments TMS320C40). This computer, which currently has nine processors, will allow us to determine maximum processing rates and the practical factors which limit those rates. Algorithms are manually coded for execution on the parallel system, although a future goal is to generate parallel code directly from our mapping tools.

Table 1 Computational Table for a Second Order IIR Digital Filter

cycle	b_in	input	a_in	fdback	r_reg	q_reg	reg8
0	0	0	0	0	$q_{1,1}$	$q_{2,1}$	$q_{1,5}$
1	$\widehat{b_{0,0}}$	f	0	0	$q_{1,1}$	$q_{2,1}$	$q_{1,6}$
2	0	f	0	0	0	0	y
3	$\widehat{b_{1,1}}$	f	$\widehat{a_{1,1}}$	y	$q_{1,2}$	0	g
4	$\widehat{b_{1,2}}$	f	$\widehat{a_{1,2}}$	y	0	0	0
5	$\widehat{b_{2,1}}$	f	$\widehat{a_{2,1}}$	y	$q_{1,3}$	$q_{2,2}$	$q_{1,1}$
6	$\widehat{b_{1,3}}$	f	$\widehat{a_{1,3}}$	y	$q_{1,4}$	0	$q_{1,2}$
7	$\widehat{b_{1,4}}$	f	$\widehat{a_{1,4}}$	y	0	0	$q_{2,1}$
8	$\widehat{b_{2,2}}$	f	$\widehat{a_{2,2}}$	y	$q_{1,5}$	0	$q_{1,3}$
9	$\widehat{b_{1,5}}$	f	$\widehat{a_{1,5}}$	y	$q_{1,6}$	0	$q_{1,4}$
10	$\widehat{b_{1,6}}$	f	$\widehat{a_{1,6}}$	y	0	0	$q_{2,2}$

VI. Applications

We have mapped a large number of image processing algorithms onto the BDPA. We briefly discuss the mapping of some of these algorithms in this section. This includes the 2-D FIR and IIR digital filters, the 2-D DCT, Cholesky factorization, QR decomposition, and the back substitution algorithm to solve a system of equations after factorization. We have included matrix operations in this list because many image processing applications can be expressed in matrix form. As an example, the popular algorithms for beam-forming involve finding the eigenvalues and eigenvectors of a matrix, and solving a system of equations [58], [59].

We programmed these applications and simulated their execution on the BDPA using the `erg` simulator [55] to evaluate their performance. For each application, we provide performance results for a sample sequence of images or matrices. These results are summarized in a graph for each application. In these graphs, *speedup* for N processors is defined as the time (total number of simulated clock cycles) to execute the program on a single-processor BDPA, divided by the time to execute the program on an N-processor BDPA. The tables showing number of cycles per output are computed as the total time to execute the program, divided by the number of outputs produced by the program (where an output is one element of a matrix, or one pixel of an image). *The number of processors N does not need to be a power of two, but it must be even.* In spite of this, for most applications, the number of processors we chose to simulate happens to be a power of two.

For each application, we also computed the correct output values using a standard symbolic algebra program (Matlab) running on a workstation. The outputs produced by `erg` were verified to be correct by comparing them with the expected output values.

A. 2-D IIR Filtering

Our early experience in developing parallel algorithms for digital signal and image processing was obtained during our efforts to develop a real-time system for filtering images

Table 2 Modified Computational Table for a Second Order IIR Digital Filter

cycle	b_in	input	a_in	fdback	r_reg	q_reg	reg8
0	b(0,0)	f	0	0	$q_{1,1}$	$q_{2,1}$	$q_{1,5}$
1	0	f	0	0	0	0	$q_{1,6}$
2	0	f	0	0	0	0	g
3	b(1,0)	f	-a(1,0)	g	$q_{1,2}$	0	0
4	b(2,0)	f	-a(2,0)	g	0	0	0
5	b(0,1)	f	-a(0,1)	g	$q_{1,3}$	$q_{2,2}$	$q_{1,1}$
6	b(1,1)	f	-a(1,1)	g	$q_{1,4}$	0	$q_{1,2}$
7	b(2,1)	f	-a(2,1)	g	0	0	$q_{2,1}$
8	b(0,2)	f	-a(0,2)	g	$q_{1,5}$	0	$q_{1,3}$
9	b(1,1)	f	-a(1,2)	g	$q_{1,6}$	0	$q_{1,4}$
10	b(2,2)	f	-a(2,2)	g	0	0	$q_{2,1}$

at TV frame rates. As a part of this effort, we designed an application specific processor for efficiently executing the state space model for the 2-D IIR digital filter [9], [49]. This processor has a three-stage pipeline using two multipliers and three adders, as shown in Fig. 12. The required equations can be solved by applying the appropriate inputs to the input registers. The output corresponding to these inputs is available in $reg8$ after two computational cycles. Table 1 gives the inputs required for the solution of the equations for each pixel with a second order IIR digital filter.

In Table 1, the previous horizontal state variable $q_{1,1}(n_1 - 1, n_2)$ and the previous vertical state variable $q_{2,1}(n_1, n_2 - 1)$ are used as inputs for the first two cycles to compute $y(n_1, n_2)$ and $g(n_1, n_2)$. This can be provided for in the control sequence for the processor. However, we found that we could simplify the control sequence without penalty if we used the current output $g(n_1, n_2)$ for the feedback instead of using the temporary variable $y(n_1, n_2)$. The corresponding equations for the second order example are given in (9)–(11). Table 2 shows the inputs required for this approach. We chose this approach for the application specific processor because of the simpler control sequence. The state variables are used only once during each sequence (11 cycles for the second order IIR filter). Since the state variables are computed in the same order they are used, the controller only needs to control whether a new input should be provided or if the input should be a null.

Both tables show that the output for a particular set of inputs to the input registers is available in $reg8$ two cycles later. Thus the output $g(n_1, n_2)$ is available on cycle two where the inputs are applied on cycle zero. The output $q_{1,5}$ on cycle zero refers to the output $q_{1,5}(n_1 - 1, n_2)$ and the output $q_{1,6}$ on cycle one refers to the output $q_{1,6}(n_1 - 1, n_2)$. All other outputs are related to the current input $f(n_1, n_2)$. Thus, the tables refer to the timing for a loop to solve all of the equations for each individual input.

We developed a hardware simulator (a predecessor to `erg`) to evaluate the use of this application specific processor in the BDPA for doing 2-D IIR filtering. The number of state equations to be solved for the 2-D IIR digital filters depends on the order of the filter. For example, the second order filter requires six horizontal state variables, and two vertical state variables. The fourth order filter requires 20 horizontal state variables and four vertical state variables.

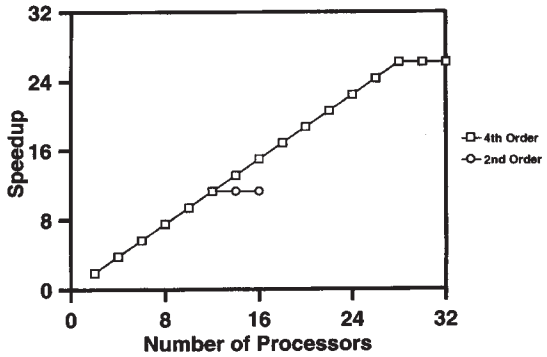


Fig. 13. Parallel speedup for 2-D IIR filtering of eight (64 × 64) images using the application specific processor.

The application specific processor requires 11 clock cycles per input to compute all of the state variables and the output for the second order filter and 27 cycles per input to compute all of the state variables and the output for the fourth order filter.

Fig. 13 shows the parallel speedup when using up to 16 processors to filter a sequence of eight images with a second order filter and up to 32 processors to filter a sequence of eight images with a fourth order filter. The system reaches its maximum output rate of almost one output per cycle when 12 processors are used for the second order filter and when 28 processors are used for the fourth order filter. Adding additional processors after this time does not significantly affect the output rate since the system can keep up with the input of almost one input per cycle. However, there is some overhead associated with handshaking so it is not possible to achieve a throughput rate of one output per cycle. The input image size was 64 pixels by 64 rows.

Table 3 shows the number of clock cycles needed to generate a single output pixel for the second order filter as a function of the number of processors used. Table 4 shows the number of clock cycles needed to generate a single output pixel for the fourth order filter as a function of the number of processors used. In computing the time required for a single processor in the above tables, we assumed that processing overlaps input but it does not for output. We also assumed that six cycles are used for handshaking per block of input data. Thus if the block size is given as M, and the number of cycles per input is given as K_c, then the

Table 3 Clock Cycles per Output for a Second Order IIR Digital Filter

Number of Processors	2	4	6	8
Cycles/Output	6.42	3.22	2.14	1.61
Number or Processors	10	12	14	16
Cycles/Output	1.29	1.07	1.07	1.07

Table 4 Clock Cycles per Output for a Fourth Order IIR Digital Filter

Number of Processors	2	4	6	8
Cycles/Output	14.97	7.49	5.00	3.75
Number or Processors	10	12	14	16
Cycles/Output	3.00	2.50	2.14	1.88
Number of Processors	18	20	22	24
Cycles/Output	1.67	1.52	1.37	1.25
Number of Processors	26	28	30	32
Cycles/Output	1.16	1.07	1.07	1.07

Table 5 Clock Cycles per Output for a Standard DSP

Number of Processors	2	4	8	16	32
Cycles/Output	296	149	75	38	19

computation time per block required for a single processor is given by

$$T(1) = (K_c + 1) * M + 6. \tag{18}$$

For example, there are 8×64 data blocks (number of rows) in a sequence of eight 64×64 images. In addition, $K_c = 11$ for a second order 2-D IIR filter. Thus $T(1) = 396\,288$ for this case.

Except for the case above for the application specific processor, the BDPA performance evaluations results for the applications presented in this paper were based upon the use of a standard DSP chip (like the TMS320C40). The speedup tables for these applications were normalized to the time for two processors. We did this because of the difficulty in obtaining an appropriate time for the single processor. The BDPA requires a minimum of two processors because of its architecture. Thus we used $T(1) = 0.5 * T(2)$. This means that the speedup for two processors is always two in these figures. The speedup for higher numbers of processors is therefore relative to the time required for two processors rather than for a single processor.

Fig. 14 shows the resulting speedup for a second order 2-D IIR digital filter for the standard DSP when filtering six frames, each of size 64×64. The number of clock cycles needed to produce each output is shown in Table 5. These results indicate that the speedup is excellent for parallel execution on the BDPA using either processor. The number of processors can be increased until the output rate matches the requirements of the application. The application specific processor achieves a performance several times better than the standard DSP chip.

B. A 2-D FIR Filter

The mapping of 2-D FIR digital filters to the BDFP is very similar to the mapping for the 2-D IIR filter, except all of the $a(j_1, j_2)$ coefficients are equal to zero. Making this

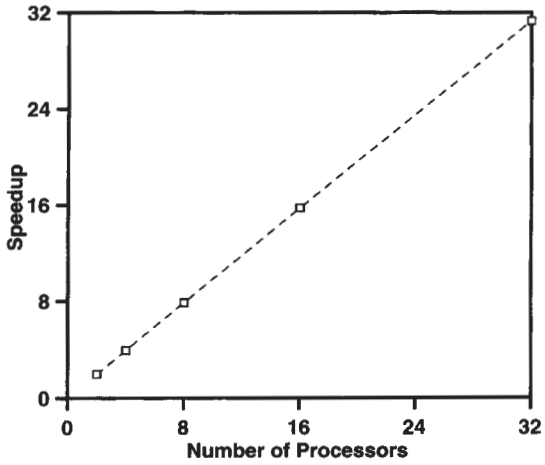


Fig. 14. Parallel speedup for 2-D IIR filtering of six (64×64) images using a standard DSP.

Table 6 Clock Cycles per Output for the Standard DSP for the 2-D Filter

Number of Processors	2	4	8	16	32
Cycles/Output	296	129	65	33	17

modification to the state equations, we obtain the following equation for the vertical state variables.

$$\begin{aligned} q_{2,j_2}(n_1, n_2) &= b(0, j_2) f(n_1, n_2) + q_{1,I_1}(n_1 - 1, n_2) \\ &\quad + q_{2,j_2+1}(n_1, n_2 - 1); \quad 1 \le j_2 \le L_2 \\ I_1 &= j_2 L_1 + 1 \\ q_{2,L_2+1}(n_1, n_2 - 1) &= 0. \end{aligned} \tag{19}$$

This equation requires one multiplication and two additions (or subtractions).

The equations for the horizontal state variables are

$$\begin{aligned} q_{1,I_1}(n_1, n_2) &= b(j_1, j_2) f(n_1, n_2) + q_{1,I_1+1}(n_1 - 1, n_2); \\ &\quad 1 \le j_1 \le L_1 - 1; \quad 0 \le j_2 \le L_2 \\ I_1 &= j_2 L_1 + j_1. \end{aligned} \tag{20}$$

$$\begin{aligned} q_{1,I_1}(n_1, n_2) &= b(j_1, j_2) f(n_1, n_2) \\ j_1 &= L_1; \quad 0 \le j_2 \le L_2. \\ I_1 &= j_2 L_1 + j_1. \end{aligned} \tag{21}$$

These equations require one multiplication and one or two additions (or subtractions).

The output equation is given by

$$\begin{aligned} g(n_1, n_2) &= b(0, 0) f(n_1, n_2) + q_{1,1}(n_1 - 1, n_2) \\ &\quad + q_{2,1}(n_1, n_2 - 1). \end{aligned} \tag{22}$$

This equation requires one multiplication and two additions (or subtractions).

Simulation results for this algorithm are shown in Fig. 15. Speedup for a standard DSP was essentially linear for up to 32 processors. The number of cycles required per output is in Table 6.

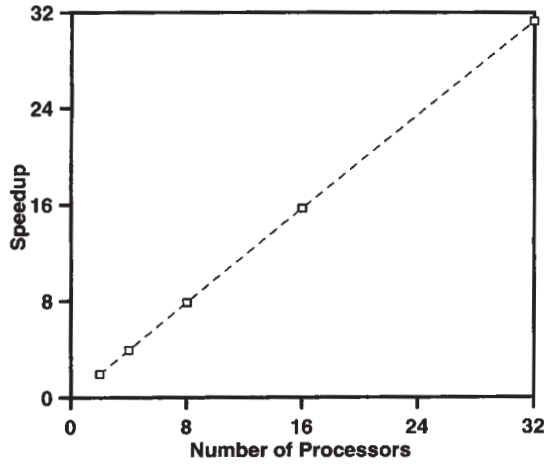


Fig. 15. Parallel speedup for 2-D FIR filtering of six (64 × 64) images.

C. A 2-D Discrete Cosine Transform

The discrete cosine transform has been widely used for data compression and for the implementation of filter banks. It has been adopted as a part of several international standards, including the high-definition television (HDTV) standard [60], [61]. The discrete cosine transform is a computationally intensive transform which is well suited for implementation using a multiprocessor system. An $n \times n$ point 2-D DCT is defined as

$$\mathbf{Y} = \mathbf{C}^{\mathbf{t}}\mathbf{X}\mathbf{C} \tag{23}$$

where $\mathbf{C}$ is the coefficient matrix, X is the input image, and $\mathbf{C}^{\mathbf{t}}$ is the transpose of the coefficient matrix. (All matrices are of dimension $n \times n$.) The elements of the matrix $\mathbf{C}$ are the coefficients for the DCT as given by

$$c_{i,j} = \sqrt{\frac{2}{n}} \cos\left[\frac{(2i-1)(j-1)\pi}{2n}\right] \tag{24}$$

where $i = 1, 2, \cdots, n$ and $j = 2, 3, \cdots, n$, and

$$c_{i,1} = n^{-1/2}. \tag{25}$$

Using the above equations, direct computation of the DCT for a $n \times n$ matrix requires on the order of $2n^3$ multiply-accumulate operations. When the size of the image is large, computation of the DCT for the entire image is prohibitively expensive. Therefore, in our research we divide the image into subimages of size $m \times m$, where $n = km$ and k is a positive integer. Given an $n \times n$ image, we compute the $m \times m$ point DCT of k^2 subimages. Fig. 16 presents an $n \times n$ image partitioned into k^2 subimages. The figure also illustrates the data partitioning used when mapping the DCT onto the BDFP. A block of data, defined as k subimages, is sent to each PM. Processors within a PM divide this $m \times n$ block of data evenly to balance workload. Each PM then produces a single $m \times n$ block of the output image.

We use a hierarchical BDPA for this application. We utilize a linear array of PM's, where each PM consists of a linear array of processors. Fig. 17 shows an example of a

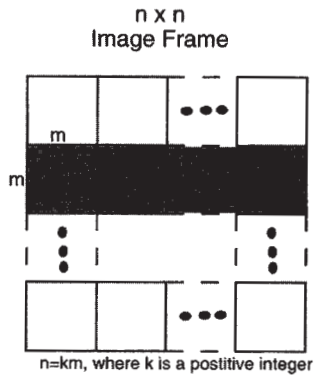


Fig. 16. Data partitioning of the DCT for the BDPA.

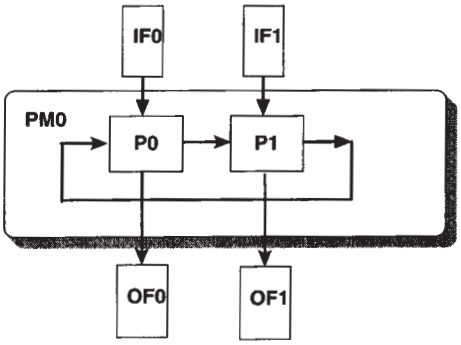


Fig. 17. A processor module containing two processors.

Table 7 Clock Cycles per Output for the 2-D DCT

Number of Processors	2	4	8	16	32
Cycles/Output	525	263	132	66	33

PM that consists of two processors. Our implementation of the DCT uses both data partitioning at the high level, and algorithm partitioning at the PM or low level.

Fig. 18 presents performance results obtained from the simulation of a 2-D DCT for which $m = 8$, $n = 64$, and $k = 8$. The number of cycles per output is shown in Table 7. In these experiments, when the number of processors was eight or less, all processors were included in a single PM. In all other cases, each PM contained eight processors requiring each processor to compute the DCT of a single 8×8 DCT subimage. Speedup was almost perfectly linear in this experiment.

D. Cholesky Factorization

Given an $n \times n$ symmetric, positive-definite matrix $\mathbf{A}$, Cholesky factorization is the determination of a triangular matrix $\mathbf{L}$ such that

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^{\mathbf{T}} \tag{26}$$

where $\mathbf{L}$ is also an $n \times n$ matrix with nonzero elements only on and below the main diagonal, i.e.,

$$\mathbf{L} = \begin{bmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ \vdots & & & \vdots & \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn} \end{bmatrix}. \tag{27}$$

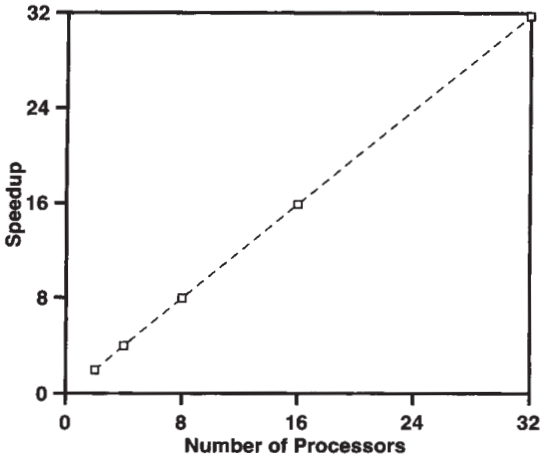


Fig. 18. Parallel speedup for the 8 × 8 2-D discrete cosine transform of a sequence of six (64 × 64) images.

Table 8 Clock Cycles per Output for a 64 × 64 Cholesky Factorization

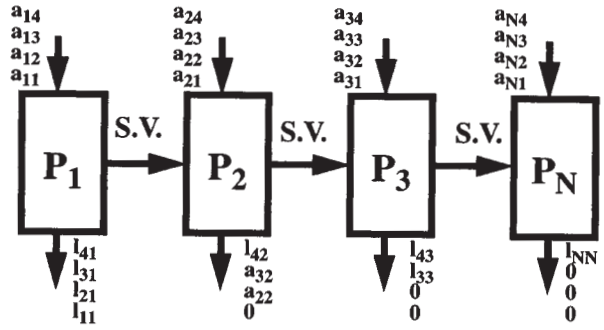
Number of Processors	2	4	8	16	32
Cycles/Output	229	114	51	20	10

This factorization originates from a special case of the well known LU factorization which is associated with Gaussian elimination [62]. Similar to the LU decomposition, Cholesky factorization decomposes a square matrix into the product of a lower triangular matrix and an upper triangular matrix. Adding the constraint on the input matrix **A** that it is symmetric and positive-definite, Cholesky factorization becomes even less computationally intensive by utilizing the relationship between the upper triangular matrix and the lower triangular matrix (i.e., $\mathbf{U} = \mathbf{L}^{\mathbf{T}}$ to reduce the total number of computations needed by a factor of $\frac{1}{2}$ as compared to LU factorization). Cholesky factorization is used in many computationally intensive applications such as data modeling, line fitting, data smoothing, and maximum likelihood estimation. However, the most prevalent application is in the solution of the linear least squares problem. Additional parallel implementations of Cholesky decomposition can be found in [63], [64].

Fig. 19 is a graphical depiction of the Cholesky Factorization on the BDPA. Each processor accepts as input a column of the input matrix as its block of data. One column of output, corresponding to the input block and the state variables received from the previous processor, is created by each processor. The code for each PM is identical, except for slight modifications for the first and last columns (boundary conditions). Figs. 20 and 21 summarize the performance of the Cholesky algorithm on the BDPA, obtained using `erg`. Parallel speedup on the BDPA for this example is excellent. For the 64 × 64 factorization, the number of cycles needed per output element is shown in Table 8.

E. QR Decomposition

Many digital signal or image processing applications require the computation of a few of the eigenvalues (and their



Processor Pseudo Code

```
Compute l_ii
  for j = i to N
  Do
     Compute l_ji
  End
  for k = 1 to i
  Do
     Compute & Send State Variable
  End
     Send outputs l_ii & lji
```

Fig. 19. Illustration of Cholesky factorization on the BDPA.

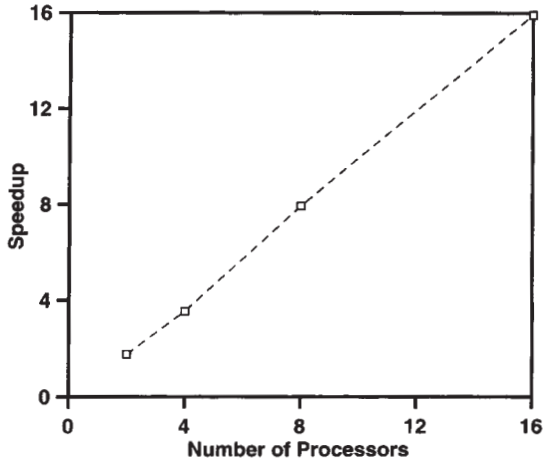


Fig. 20. Parallel speedup for Cholesky factorization of a sequence of 50 (32 × 32) matrices.

corresponding eigenvectors) for a large matrix. Examples of such applications include array signal processing [65], system identification [66], [59], image processing [67], spectrum estimation [68], and filter design [69]. The first step of the partial eigenvalue solution algorithm is the QR decomposition of a covariance matrix **E** to solve the set of linear equations $\mathbf{EU} = \mathbf{S}$, where $\mathbf{E} = \mathbf{QR}$ and $\mathbf{B} = \mathbf{Q}^{\mathbf{t}}\mathbf{S}$. We now discuss the mapping of the required QR decomposition to the BDFP. This requires the QR decomposition of **E** to find **R** and **B**.

$$\begin{aligned} \mathbf{EU} &= \mathbf{S} \\ \mathbf{Q}^{\mathbf{T}}\mathbf{EU} &= \mathbf{Q}^{\mathbf{T}}\mathbf{S} \\ \mathbf{RU} &= \mathbf{B}. \end{aligned} \tag{28}$$

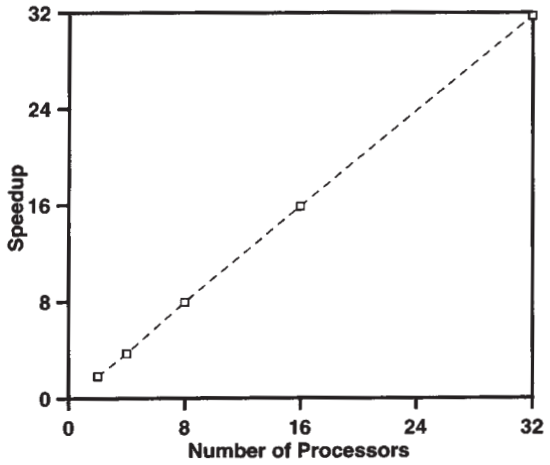


Fig. 21. Parallel speedup for Cholesky factorization of a sequence of six (64 × 64) matrices.

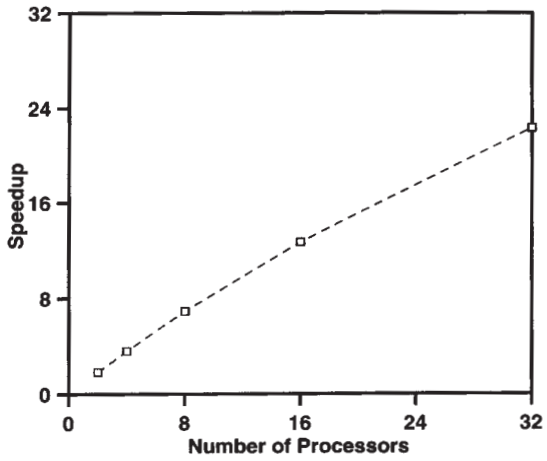


Fig. 22. Parallel speedup for QR decomposition of a sequence of six (64 × 64) matrices.

Table 9 Clock Cycles per Output for QR Decomposition

Number of Processors	2	4	8	16	32
Cycles/Output	685	352	184	100	57

We used Given's rotations for the QR decomposition. The purpose of the Given's rotation is to annihilate the subdiagonal elements of matrix **E** and reduce it to upper triangular form. In Given's rotation, the subdiagonal elements of the first column are annihilated first, then the elements of the second column, and so forth until an upper triangular form is eventually reached. The matrix **Q** is a product of matrices used to zero elements of **E**. Each element of **E** (e_{ij}) is zeroed by multiplying **E** by an orthogonal matrix $\mathbf{Q_{ij}}$. The matrix $\mathbf{Q_{ij}}$ is formed from the identity matrix by replacing the diagonal (i,i) and (j,j) elements by c_{ij}, the (i,j) element by s_{ij}, and the (j,i) element by $-s_{ij}$ where

$$c_{ij} = \frac{e_{jj}}{\sqrt{e_{jj}^2 + e_{ij}^2}}$$
$$s_{ij} = \frac{e_{ij}}{\sqrt{e_{jj}^2 + e_{ij}^2}}. \tag{29}$$

Multiplying **E** by $\mathbf{Q_{ij}}$ updates row i $(\mathbf{e}_i)$ and row j $(\mathbf{e}_j)$ of **E**

$$\mathbf{e}_i \leftarrow -s_{ij}\mathbf{e}_i + c_{ij}\mathbf{e}_j$$
$$\mathbf{e}_j \leftarrow c_{ij}\mathbf{e}_i + s_{ij}\mathbf{e}_j. \tag{30}$$

The element e_{ij} becomes

$$\hat{e}_{ij} = -\frac{e_{ij}e_{jj}}{\sqrt{e_{jj}^2 + e_{ij}^2}} + \frac{e_{jj}e_{ij}}{\sqrt{e_{jj}^2 + e_{ij}^2}} = 0. \tag{31}$$

Therefore, a subdiagonal element, e_{ij}, is zeroed in this way, and the two rows $\mathbf{e}_i$ and $\mathbf{e}_j$ are modified for each element e_{ij} that is zeroed.

We partitioned the matrix **E** by columns. A column of **E** and a row of **S** are sent to each PM as a block of data. Each PM receives previously computed c_{ij} and s_{ij} pairs and the updated **B** from the previous PM as state variables. The PM annihilates the subdiagonal elements for its assigned column and updates **B**. It then sends the updated **B**, its newly computed c_{ij} and s_{ij} values and all previously computed c_{ij} and s_{ij} values to the next PM as state variables. Finally, it sends the results for its assigned column of **R** to the output (OM).

The above algorithm is a modified version of the QR decomposition algorithm, in which we update **B** while computing **Q** and **R**. This is a more efficient approach when the final goal is to compute **B**. **B** does not need to be updated and communicated to the next PM for the normal QR decomposition. Fig. 22 presents simulation results for the QR decomposition algorithm that computes **Q** and **R**. The number of cycles per output is shown in Table 9. Fig. 23 presents the simulation results for the modified QR decomposition algorithm that computes **R** and updates **B** during each pass. In this experiment, the **B** matrix has dimensions 64 × 4. Speedup is acceptable for both of these, although less than ideal in this case.

F. Back Substitution

The QR decomposition algorithm described above would be followed by the use of back substitution to obtain the solution of a set of equations. The goal of this algorithm is to find **U** for

$$\mathbf{RU} = \mathbf{B} \tag{32}$$

given the upper triangular matrix **R** and the matrix **B**. Since **R** is a $m \times m$ matrix and **U** and **B** are $m \times k$ matrices, this algorithm solves for k elements of U simultaneously. Since **R** is upper triangular, the linear equations corresponding to the rows of the matrix equation are solved in reverse order, starting with the last row. This procedure is repeated until the equation corresponding to the first row is solved.

We partitioned this problem by row, with each PM receiving a row of **R** and a row of **B** as a block of data.

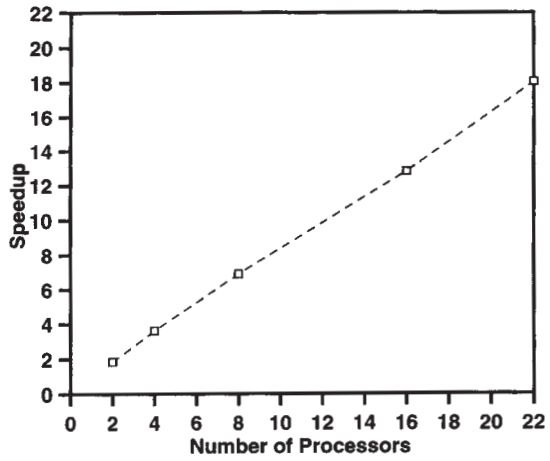


Fig. 23. Parallel speedup for modified QR decomposition of a sequence of 50 (64 × 64) matrices.

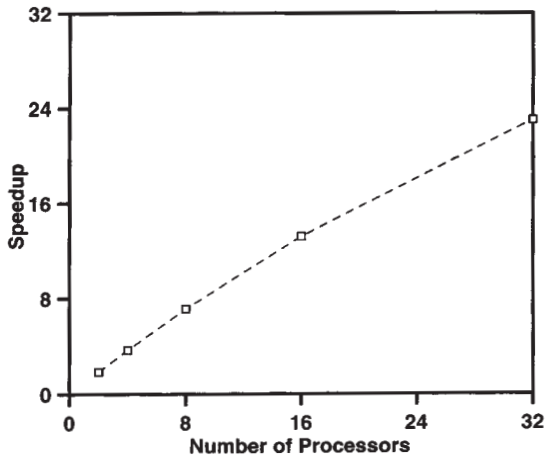


Fig. 24. Parallel speedup for back substitution of a sequence of six (64 × 64) matrices.

Table 10 Clock Cycles per Output for Back Substitution

Number of Processors	2	4	8	16	32
Cycles/Output	65	33	17	10	6

The PM receives previously computed values of U from the previous PM as state variables. It then updates a row of U corresponding to its assigned row and sends the updated values of U and all previously computed values of U to the next PM as state variables. The PM also sends the updated values of U corresponding to its assigned row to the output (OM).

Simulation results for this algorithm are shown in Fig. 24. Again, speedup is acceptable although not ideal. The number of cycles per output is shown in Table 10.

G. Summary

The examples of applications we have given show the wide range of applicability of the BDFP. Our simulated experimental results indicate that the architecture and our method of partitioning effectively exploits the parallelism inherent in these applications.

VII. Conclusion

We feel that the BDPA provides a solution to many of the problems associated with high performance digital signal and image processing. The use of block processing reduces the requirement for data communication. The use of data flow computing permits the processors to operate asynchronously. We developed the BDFP as a paradigm for mapping digital signal and image processing applications on a multiprocessor system and we presented methods for partitioning applications for the BDFP. By restricting the data communication at the algorithm level, we can require all communication to be local (nearest neighbor) and in one direction (normally up to down and left to right). This allows the processors to skew operations which makes it easier to keep all processors operating at the same time.

The BDFP evolved from our research on developing a multiprocessor architecture for implementing 2-D digital filters in real-time. Through extensive simulation, we were able to verify that this approach could achieve almost linear speedup and very high efficiency as the number of processors increases until the processing rate equals the input data rate. The BDFP can be used to implement any algorithm that can be implemented using block data processing. Thus we expect the BDFP to be widely applicable to a wide variety of high performance digital signal and image processing applications. We have mapped a large number of image processing algorithms onto the BDPA and we presented performance evaluation results that are consistent with our claim that the BDPA can provide almost linear speedup for many image processing algorithms.

We feel that the BDPA is flexible and adaptable to changes in technology. We have explored the use of application specific processors, DSP chips, and general purpose processors as node processors in the BDPA. Either of these can be used although the highest performance can be provided with the use of application specific processors. However, special purpose processors such as DSP chips offer a good compromise between high performance and programmability. On the other hand, the greatest flexibility and lowest cost can be provided with general purpose processors as node processors in the BDFA.

We are experimenting with the implementation of the DCT using Pentek 4284 and 4270 boards in a VME bus extender connected to the ISA bus of a PC. One goal of this experiment is to determine if commercial boards with multiple DSP's such as the Pentek 4270 can be used to implement the BDFP. We use the Pentek 4284 in our setup to emulate the IM and the OM and one or more Pentek 4270 boards to emulate the PMA. Our early experiments show that the bottleneck for this configuration is the I/O. We are currently working on optimizing the I/O so that we can take advantage of the processing power of the multiple DSP's on the 4270 boards.

Our work presented in this paper involves the use of the BDFP and the BDPA on low level image processing applications. However, we feel that the BDFP is appropriate

for many medium level and high level image processing applications as well. Since the communication between PM's is asynchronous and computation is based upon data flow, a multiprocessor system based upon the BDFP can dynamically adjust work load based upon the requirements of the application. In addition, our concept of the BDPA involves the use of a stored program in each PM rather than central control. Thus a PM can run different program segments based upon the intermediate data it receives, etc. We are exploring some of these ideas. We are also exploring the use of hierarchical BDPA systems where the PM is replaced by another BDPA. However, the BDFP does not support global communication in general. Our approach is to avoid this problem by remapping the algorithm so that only local communication is required.

As we continue our research, we will continue to focus on developing a systematic approach to mapping image processing applications to the BDFP. Our goal is to provide software tools to automate the whole algorithm partitioning and scheduling process. Our experience in using the state space model and the order graph method will provide the base for the development of these tools. We will continue to improve our tools for performance evaluation because we need feedback from these to evaluate our research results on actual image processing applications. Finally, we plan to continue to explore opportunities to test our methods on actual hardware. This includes exploring the use of the BDFP to map applications to commercial multiprocessor systems as well as a small scale effort to develop our own hardware prototype.

ACKNOWLEDGMENT

The authors would like to thank the following former and current students who have contributed to this project: J. H. Kim, S. M. Park, H. Xu, J. G. Jeong, M. Dabbagh, S. Howard, S. Alexandre, G. Cato, K. Ellis, H. Ko, S. H. Yoon, P. R. Meyer, and V. Wilburn.

REFERENCES

[1] W. Sung, S. K. Mitra, and B. Jeren, "Multiprocessor implementation of digital filtering algorithms using a parallel block processing model," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 3, 1992.

[2] T. P. Barnwell III, V. K. Madisetti, and S. J. McGrath, "The Georgia Tech digital signal multiprocessor," *IEEE Trans. Signal Process.*, vol. 41, no. 7, 1993.

[3] S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[4] S. Sunder and V. Ramachandran, "Systolic implementation of multidimensional nonrecursive digital filters," *IEEE Trans. Circ. and Syst. for Video Technol.*, vol. 3, pp. 399–407, 1993.

[5] C. W. Wu, "Bit-level pipelined 2-D digital filters for real-time image processing," *IEEE Trans. Circ. and Syst. for Video Technol.*, vol. 1, pp. 22–34, 1991.

[6] D. Dudgeon and R. Mersereau, *Multidimensional Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[7] D. A. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn, "A second opinion on data flow machines and languages," *Computer*, vol. 15, pp. 58–69, 1982.

[8] B. C. Kuo, *Digital Control Systems*. New York: Holt, Rinehart and Winston, 1980.

[9] J. H. Kim and W. E. Alexander, "A multiprocessor architecture for two-dimensional digital filters," *IEEE Trans. Comput.*, vol. C-36, pp. 876–884, 1987.

[10] W. E. Alexander and C. J. Ju, "Considerations for the block state implementation of multidimensional digital filters," in *Proc. Asilomar Conf. on Signals, Syst. and Comput.*, 1987.

[11] R. Roesser, "A discrete state space model for linear image processing," *IEEE Trans. Autom. Contr.*, vol. AC-20, pp. 1–10, 1975.

[12] E. Fornasini and G. Maresini, "State space realization theory for two dimensional filters," *IEEE Trans. Autom. Contr.*, vol. AC-21, pp. 484–492, 1976.

[13] M. R. Azimi–Sadjadi and R. A. King, "Two-dimensional block processor-structures and implementations," *IEEE Trans. Circ. and Syst.*, vol. CAS-33, pp. 42–50, 1986.

[14] C. J. Ju and W. E. Alexander, "Block realization of multidimensional IIR digital filters and its finite word length effects," *IEEE Trans. Circ. and Syst.*, vol. CAS-34, pp. 1030–1044, 1987.

[15] S. Y. Kung, S. C. Lo, S. N. Jean, and J. N. Hwang, "Wavefront array processor—Concept to implementation," *Computer*, vol. 20, no. 7, pp. 18–33, 1987.

[16] M. C. Chen, Y. Choo, and J. Li, "Compiling parallel programs by optimizing performance," *J. Supercomput.*, vol. 2, pp. 171–207, Oct. 1988.

[17] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 3, pp. 179–193, Mar. 1992.

[18] P. H. Hartel, H. Glaser, and J. M. Wild, "Compilation of functional languages using flow graph analysis," *Software-Practice and Experience*, vol. 24, no. 2, pp. 127–173, Feb. 1994.

[19] J. Ramanujam, "Compile-time techniques for data distribution in distributed memory machines," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 2, pp. 472–482, Oct. 1991.

[20] F. Gasperoni and U. Schwiegelshohn, "Scheduling loops on parallel processors: A simple algorithm with close to optimum performance," *Lecture Notes in Computer Sci.*, no. 634, pp. 625–636, 1992.

[21] K. Hwang, *Advanced Computer Architecture With Parallel Programming*. San Francisco: McGraw-Hill, 1993.

[22] K. K. Parhi and D. G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Trans. Comput.*, vol. 40, pp. 178–195, Feb. 1991.

[23] C. M. Wang and S. D. Wang, "Efficient processor assignment algorithms and loop transformations for executing nested parallel loops on multiprocessors," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 3, pp. 71–82, Jan. 1992.

[24] M. E. Wolfe and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 2, pp. 452–471, Oct. 1991.

[25] M. C. Chen and C. E. Mead, "Concurrent algorithm as space-time recursion equations," in *Proc. USC Workshop VLSI Modern Signal Process.*, Nov. 1982, pp. 31–52.

[26] V. Van Dongen and P. Quinton, "The mapping of linear recurrence equations on regular arrays," *J. VLSI Signal Process.*, vol. 1, pp. 95–113, 1989.

[27] L. Johnsson and D. Cohen, "A mathematical approach to modeling the flow of data and control in computational networks," in *CMU Conf. VLSI Syst. and Computers*, Oct. 1981, pp. 226–234.

[28] W. L. Miranker and A. Winkler, "Spacetime representations of computational structures," *Computing*, vol. 32, pp. 93–114, 1984.

[29] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrence equations," in *Proc. 11th Annu. Symp. Comput. Architecture*, 1984, pp. 208–214.

[30] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree, and graph problems," in *Proc. 27th Annu. Symp. on Found. of Computer Sci.*, New York, 1986, pp. 478–491.

[31] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Computer*, pp. 50–56, June 1982.

[32] S. H. Huang and J. M. Rabaey, "Maximizing the throughput of high performance DSP applications using behavioral transformations," in *Proc. Europe. Design and Test Conf.*, 1994, pp. 25–30.

[33] W. Shen and D. Sweeting, "Heuristic algorithms for task assignment and scheduling in a processor network," *Parallel Computing*, vol. 20, no. 1, pp. 1–14, Jan. 1994.

[34] B. A. Curtis and V. K. Madisetti, "Rapid prototyping on the Georgia Tech digital signal multiprocessor," *IEEE Trans. Signal Process.*, vol. 42, pp. 649–662, Mar. 1994.
[35] P. R. Gelabert and T. P. Barnwell III, "Optimal automatic periodic multiprocessor scheduler for fully specified flow graphs," *IEEE Trans. Signal Process.*, vol. 41, pp. 858–888, Feb. 1993.
[36] S. M. H. de Groot, S. H. Gerez, and O. E. Herrmann, "Range-chart-guided iterative data-flow graph scheduling," *IEEE Trans. Circ. and Syst.*, vol. 39, pp. 351–364, May 1992.
[37] M. Renfors and Y. Neuvo, "The maximum sampling rate of digital filters under speed constraints," *IEEE Trans. Circ. and Syst.*, vol. 28, pp. 196–202, 1981.
[38] K. Hwang, Y. H. Cheng, F. D. Angers, and C. Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication," *SIAM J. Computing*, vol. 18, pp. 244–257, 1989.
[39] M. Potkonjak and J. M. Rabaey, "Scheduling algorithms for hierarchical data control flow graphs," *Int. J. Circ. Theory and Applicat.*, vol. 20, pp. 217–233, 1992.
[40] S. Dandamudi, "A comparison of task scheduling strategies for multiprocessor systems," in *Proc. 3rd IEEE Symp. on Parallel and Distrib. Process.*, 1991, pp. 423–426.
[41] K. Konstantinides, R. T. Kaneshiro, and J. R. Tani, "Task allocation and scheduling models for multiprocessor digital signal processing," *IEEE Trans. Signal Process.*, vol: 38, Dec. 1990.
[42] C. S. R. Krishnan, D. A. L. Piriyakumar, and C. S. R. Murthy, "Task allocation and scheduling models for multiprocessor digital signal processing," *IEEE Trans. Signal Process.*, vol. 43, no. 3, pp. 802–805, Mar. 1995.
[43] S. Manoharan and N. P. Topham, "An assessment of assignment schemes for dependency graphs," *Parallel Computing*, vol. 21, no. 1, pp. 85–107, Jan. 1995.
[44] C. L. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle, "A comparison of heuristics for scheduling dags on multiprocessors," in *Proc. 8th Int. Parallel Process. Symp.*, 1994, pp. 446–451.
[45] C. E. Leiserson and J. B. Saxe, *Retiming Synchronous Circuitry*. Palo Alto, CA: Syst. Res. Ctr., 1989.
[46] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.
[47] G. R. Cato and D. S. Reeves, "Parallel task scheduling using the order graph method," in *Proc. 1995 IEEE 14th Annu. Int. Pheonix Conf. on Computers and Commun.*, Mar. 1995, pp. 69–75.
[48] Y. L. Varol and D. Rotem, "An algorithm to generate all topological sorting arrangements," *Computer J.*, vol. 24, pp. 83–84, 1981.
[49] S. M. Park *et al.*, "A novel VLSI architecture for the real-time implementation of 2-D signal processing systems," in *Proc. IEEE Int. Conf. on Compter Design: VLSI in Computers and Processors*, 1988.
[50] M. Y. Dabbagh and W. E. Alexander, "Frequency domain implementation of block state space 2-D digital filters," in *Proc. 31st Midwest Symp. on Circ. and Syst.*, 1988.
[51] ——, "VLSI multiprocessor implementation of block state-space digital filters," in *Proc. Int. Symp. on Circ. and Syst.*, 1989.
[52] ——, "Multiprocessor implementation of 2-D denominator-separable digital filters for real-time processing," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 37, pp. 872–881, June 1989.
[53] H. Xu and W. E. Alexander, "A high performance architecture for real-time signal processing and matrix operations," in *Proc. 1992 IEEE Int. Symp. on Circ. and Syst.*, 1992, pp. 1057–1060.
[54] S. A. Howard and W. E. Alexander, "Simulation and performance evaluation of a parallel architecture for signal processing," in *Proc. Southeast. Symp. on Syst. Theory*, Mar. 1994, pp. 82–85.
[55] S. Alexandre, W. Alexander, and D. Reeves, "A programmable simulator for analyzing the block data flow architecture," in *Proc. 2nd Workshop on Modeling, Analysis, and Simulation of Computer and Telecommun. Syst. (MASCOTS'94)*, 1994, IEEE Computer Soc. Press.
[56] P. Beadle, C. Pommerell, and M. Annaratone, "K9: A simulator of distributed-memory parallel processors," in *IEEE Proc. Supercomputing'89*, 1989, pp. 765–774.
[57] S. Mehrotra, "Developing a simulator for the USC orthogonal multiprocessor," in *Proc. 1990 Winter Simulation Conf.*, 1990, pp. 857–862.
[58] C. S. Lee and R. J. Evans, "A new eigenvector weighting method for stable high resolution array processing," *IEEE Trans. Signal Process.*, vol. 40, no. 4, pp. 999–1004, 1992.
[59] C. L. Nikias and M. I. Guerlli, "An eigenvector-based algorithm for multichannel blind deconvolution of input colored signals," *IEEE Trans. Signal Process.*, vol. 43, pp. 134–149, 1995.
[60] Y. T. Chang and J. J. Leou, "New sytstolic array implementation of the 2-D discrete cosine transform and its inverse," *IEEE Trans. Circ. and Syst. for Video Technol.*, vol. 5, pp. 150–157, 1995.
[61] A. Madisetti and A. N. Wilson, "A 100 MHz 2-D 8×8 DCT/IDCT processor for HDTV applications," *IEEE Trans. Circ. and Syst. for Video Technol.*, vol. 5, pp. 158–165, 1995.
[62] R. S. Martin, G. Peters, and J. H. Wilkinson, "Symmetric decomposition of a positive definite matrix," *Numerical Math*, pp. 362–383, 1965.
[63] G. H. Golub and C. F. Van Loan, *Matrix Computations*. Baltimore: Johns Hopkins, 1989, ch. 4, pp. 139–149.
[64] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, Soc. Indust. and Appl. Math., chs. 5 and 6, pp. 75–142, 1991.
[65] M. Lu, "A toeplitz-induces mapping technique in sensor array processing," *IEEE Trans. Signal Process.*, vol. 43, pp. 1128–1139, 1995.
[66] C. E. Davila and H. Chiang, "An algorithm for ploe-zero system model order estimation," *IEEE Trans. Signal Process.*, vol. 43, pp. 1013–1017, 1995.
[67] W. K. Pratt, *Digital Image Processing*. New York: Wiley Interscience, 1978.
[68] Z. Fu and E. M. Dowling, "Conjugate gradient eigenstructure tracking for adaptive spectral estimation," *IEEE Trans. Signal Process.*, vol. 43, pp. 1151–1160, 1995.
[69] W. Lu, H. Wang, and A. Antoniou, "Design of two-dimensional FIR digital filters using the singular-value decomposition," *IEEE Trans. Circ. and Syst.*, vol. 37, pp. 35–45, 1990.

The authors

Winser E. Alexander *Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina 27695 (winser@eos.ncsu.edu).* Dr. Alexander received a B.S. in electrical engineering from North Carolina Agricultural and Technical State University in 1964 and an M.S. in engineering and a Ph.D. in electrical engineering from the University of New Mexico in 1966 and 1974, respectively. He is currently a Professor in the Electrical and Computer Engineering Department at North Carolina State University and has been a faculty member there since 1982. He was Chair of the Department of Electrical Engineering at North Carolina Agricultural and Technical State University from 1976 to 1982. Professor Alexander has also been employed as a Member of Technical Staff at Sandia Laboratories, Albuquerque, New Mexico, and has served as an officer in the United States Air Force. His research interests are in the areas of parallel algorithms, special-purpose multiprocessor architectures for digital signal and image processing, and multidimensional digital signal and image processing. He received the Presidential Award for Excellence in Science, Mathematics and Engineering Mentoring from the White House Office of Science and Technology and the National Science Foundation in 1998. He also received the American Society for Engineering Education (ASEE) Minorities in Engineering Award for Leadership in the Conception, Organization, and Operation of Precollege and College Activities to Increase Participation of Minorities in Engineering in 1993.

Douglas S. Reeves *Department of Computer Science and Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina 27695 (reeves@eos.ncsu.edu).* Dr. Reeves received a Ph.D. degree in computer science from Pennsylvania State University in 1987. Since then he has been a member of the faculty of the Computer Science Department and the Electrical and Computer Engineering Department at North Carolina State University. His research interests include network security and quality of service in computer networks.

Clay S. Gloster, Jr. *Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina 27695 (gloster@eos.ncsu.edu).* Dr. Gloster is currently an Associate Professor in the Department of Electrical and Computer Engineering at North Carolina State University. He received his B.S. and M.S. degrees in electrical engineering from North Carolina Agricultural and Technical State University, and a Ph.D. degree in computer engineering from North Carolina State University. He has also been employed with IBM, the Department of Defense, and the Microelectronics Center of North Carolina. Professor Gloster's research currently focuses on the identification of applications that have the potential for significant speedup when they are implemented on reconfigurable computing systems—and on the development of automated tools that assist scientists/engineers in mapping those applications onto such systems. He is also actively conducting research in the area of technology-based curriculum development and distance education. Dr. Gloster is a member of Eta Kappa Nu, Tau Beta Pi, and ACM, and is a registered professional engineer.

Publication of this paper

This paper was originally published on pages 947–968 of the *Proceedings of the IEEE,* Volume 84, Number 7 (1996) It was produced by scanning the original version and contains updated biographical sketches of its authors. We gratefully acknowledge permission to include it in this issue.