Mixed-mode parallel processing systems are capable of executing in either SIMD (synchronous) or MIMD (asynchronous) mode of parallelism. The ability to switch between the two modes at instruction-level granularity allows the parallelism mode to vary for each portion of an algorithm. To fully exploit the capability of intermixing both SIMD and MIMD operations within a single program, one must determine the optimal mapping of an algorithm to the mixed-mode architecture. The phase optimization technique, where the programmer makes an implicit assumption that by combining the best version of each phase the optimal implementation of the entire program will be achieved, generally works in a serial computer environment. The application of this approach to the selection of a mode of parallelism for each phase is investigated by presenting a detailed study of a practical image-processing application, the Edge-Guided Thresholding algorithm, and its mapping to a mixed-mode parallel architecture. The six functional phases of the algorithm, as well as their temporal juxtaposition, are analyzed along with experimental performance measurements obtained from the PASM parallel processing prototype, a mixed-mode system. The results discussed here demonstrate a situation in which the advantages of a mixed-mode approach are limited.

1. INTRODUCTION

One significant issue in the field of massively parallel computing is the mapping of algorithms to parallel computer architectures. To fully exploit the benefit of a parallel processing system, the programmer must understand how to structure an algorithm to take advantage of the target architecture.

Adaptive parallel systems add a new dimension by allowing the user to configure the target machine to best match the algorithm. Although such systems offer potentially higher performance for a wider range of applications, the flexibility places a burden on the programmer to choose for each algorithm those system parameters that will result in the highest performance. Ideally a compiler would remove the burden from the programmer, but usually to achieve peak performance the parameters of the target architecture must be considered throughout the algorithm design process.

In a mixed-mode parallel system, processors are capable of executing instructions in both the SIMD (single-instruction stream–multiple-data stream) and MIMD (multiple-instruction stream–multiple-data stream) modes of parallelism [13] and switching between these two modes at instruction-level granularity. With such flexibility, it is possible for SIMD- and MIMD-mode operations to be intermixed within a single algorithm. Thus, when implementing a given task, the programmer or compiler has an additional parameter to consider. The parallelism mode can be varied on the task, subtask, or even instruction level.

A mixed-mode machine must have a control unit to broadcast instructions, as in an SIMD machine, and each processor must be able to decode instructions into control signals as in an MIMD machine. This research is part of an ongoing series of studies to model and evaluate the worth of the mixed-mode architectural approach to parallelism. Earlier works have demonstrated strengths of this approach, e.g., [6, 11, 12]. This work illustrates a potential weakness.

The optimum mapping of multiphase algorithms to mixed-mode parallel processing systems can be determined through a process of experimentation. One straightforward approach starts by applying phase optimization, a common technique in serial computing. The program is divided into functional phases and, through experimentation, the implementation of each phase is optimized. Several versions of each phase are created, possibly utilizing different parallelism modes, and their performance is empirically measured. The final program consists of the implementation of each phase that had the best performance when executed in isolation.

When applying the phase optimization technique, the programmer makes an implicit assumption that by combining the best version of each phase the optimal implementation of the entire program will have been achieved. In uniprocessor systems, this assumption may not be valid if input/output formats are mismatched (i.e., each phase requires a
processing systems are programmed, even if the input/output formats match, the phase-optimized approach may not yield the optimal solution due to aspects, not found in typical uniprocessor machines, that complicate the interaction among phases.

To demonstrate the process of mapping algorithms to mixed-mode processors, a practical, multiple-phase image-processing application, the edge-guided thresholding (EGT) algorithm, is discussed in detail. After background information is given in Section 2, the EGT algorithm is introduced in Section 3. In Section 4, several implementations for each phase of the EGT algorithm are presented and thoroughly analyzed. The trade-offs among the implementations are studied along with experimental data obtained from a mixed-mode computer, the PASM parallel processing prototype. Section 5 contains an analysis of performance measurements for several versions of the entire algorithm, where each version consists of a different combination of phase implementations. These analyses attempt to concentrate on inherent relative differences among SIMD, MIMD, and mixed-mode processing.

The goal of this paper is to contribute to the development of an overall theory of mixed-mode parallelism, which involves exploring both the strengths and the weaknesses of the approach. The EGT algorithm used in this study was chosen because it is a practical algorithm and has computational characteristics that lead to interesting results in terms of mixed-mode mapping. How well the EGT algorithm compares to other similar image-processing techniques is not the issue here; it is its computational characteristics that are important for this study, not its image-processing performance. By recognizing how aspects of the computational structure of this algorithm interact with mixed-mode parallel architectures, insights that will help further the general study of mapping algorithms onto massively parallel architectures and the development of techniques for automatic compilation of parallel code can be gained. The results discussed here demonstrate a situation in which the advantages of a mixed-mode approach are limited.

2. BACKGROUND

2.1. The Mixed-Mode Architectural Model

There have been at least three mixed-mode parallel prototypes built. TRAC [18], designed and built at The University of Texas at Austin, implemented a mixed-mode architecture through a special SW-baynan interconnection network. OPSILA [1, 2, 9], from Laboratoire de Signaux et Systemes in Nice, France, has the capability to execute instructions in either SIMD or SPMD (single-program-multiple-data stream) mode, a subclass of MIMD.

The mixed-mode architectural model that has been assumed for this research is based on the PASM (partitionable SIMD/MIMD) design for a large-scale parallel processing system [23]. A small-scale 30-processor prototype has been built at Purdue with a computation engine of 16 processing elements (PEs), each consisting of a processor–memory pair [24]. The PEs can communicate with each other through a circuit-switched, extra-stage cube network, a variation of the multistage cube network. The PE processors are MC68000s and perform the actual SIMD and MIMD computation. In SIMD mode, the instructions are broadcast by a control unit (CU) and executed synchronously by all the enabled PEs using data contained in their local memories. There exists a mechanism for the CU to globally disable PEs and for PEs to transmit the results of tests on local data to the CU.

In MIMD mode, PEs execute instructions from their local memory, asynchronously with respect to each other. To change from MIMD parallelism to SIMD parallelism, PEs begin to fetch instructions from a logical address space corresponding to a queue of instructions sent by the CU. Any instruction fetches other than those from this queue return the program to the MIMD mode of parallelism. Thus, any branch statement can change the mode of computation. The CU can also broadcast data to the PEs via the SIMD queue, allowing them to use values computed dynamically by the CU. More details of the PASM prototype are discussed in the text as needed.

The goal of this research is to study the inherent properties of mixed-mode parallel architectures and their computational characteristics. The PASM prototype is used as a tool in the experiments. The results obtained from the experiments are evaluated and analyzed in terms of the properties of the modes of parallelism so the conclusions are as general and implementation-detail-independent as possible.

Experimentation has been performed on a PASM submachine with four PEs. Unless otherwise noted, all the instructions that would be executed in a larger system are executed in the four-PE configuration. Because the execution time of the parallel EGT algorithm is related to the data set size in each PE and not the number of PEs, the results obtained on a larger system would be similar to those presented. This is discussed further in Subsection 5.2.

Several algorithm studies demonstrating the advantages of a mixed-mode architecture have been performed using PASM. Examples include parallel FFTs [6], bitonic sequence sorting [12], and matrix multiplication [11]. Although the previous studies have discovered several important concepts about mixed-mode computing, the work in this paper differs in that the previous studies have concentrated on algorithms that typically would be a single phase in a larger task. In this research, the mapping of multiple-phase algorithms to mixed-mode architectures is explored. The result of the juxtaposition of phase implementations that are optimal in isolation is the focus of this study. General discussions of different aspects of the trade-offs among SIMD-mode, MIMD-mode, and mixed-mode parallelism can be found in [5, 14, 15].
2.2. General SIMD/MIMD Trade-offs

Both the SIMD and MIMD modes of parallelism have their advantages and disadvantages. SIMD is characterized by its single instruction stream, implicit synchronization, and a CU whose operations can be overlapped with PE operations. The single stream of instructions implies that SIMD mode may be inefficient for, and therefore not applicable to, tasks where most processors cannot follow the same single thread of control. In such situations, MIMD mode is preferable.

The implicit synchronization of SIMD mode can be either an advantage or a disadvantage. One advantage is it allows for a simple interprocessor communication protocol. In MIMD, explicit synchronization, polling, or an interrupt-driven scheme along with a higher level protocol must be used to ensure proper data transfers. One disadvantage of the implicit synchronization of SIMD is that it can cause some PEs to remain idle unnecessarily.

Another advantage of SIMD is the capability to overlap the computations being performed by the PEs with the CU's execution of control-flow instructions and/or any instructions common to all the PEs, e.g., address calculations where each PE is accessing the same address relative to its own local memory. The amount of performance gain due to CU overlap depends upon the ratio of work between the PEs and CU. To achieve maximum performance gain, the algorithm must be structured such that one-half of the work is performed by the CU (i.e., neither the CU nor the PEs are ever idle) [16]. Even when only a smaller amount can be overlapped, a significant speedup can occur compared to that of a MIMD implementation, where the PEs must execute all the instructions.

In general, to achieve high performance with a mixed-mode machine, the parallelism modes for a given algorithm must carefully be selected on the basis of the characteristics of the algorithm. As is demonstrated, the impact of the temporal juxtaposition of modes must also be considered.

3. THE EGT ALGORITHM

3.1. Overview of the Algorithm

To study the mapping of algorithms to mixed-mode parallel processors, a detailed discussion of an image-processing algorithm, the EGT algorithm [26], is presented. The paper does not attempt to argue the quality of the EGT algorithm from an image-processing perspective (it is irrelevant), but instead uses the algorithm as a vehicle to examine computational characteristics that might arise in a variety of applications.

The purpose of the EGT algorithm is to determine the optimal threshold to be used for segmentation [20] (re-quantizing the image from pixels (picture elements) with 256 gray levels to one of two levels (zeros and ones)). Any pixel with a value less than the threshold level has a zero in the corresponding position of the segmented image. Any pixel with a value greater than or equal to the threshold level has a corresponding one. The EGT algorithm is designed for processing images of size \( M \times M \) pixels consisting of an object immersed in a background. Ideally, but not in practice, the object is all one gray level and the background is another gray level (forming a "two-level" image). In practice, the image may have variable lighting so that the object, as well as the background, may have slowly varying gray levels. Such a situation arises, for example, in the automated inspection of printed circuit boards.

The goal of the EGT algorithm is to determine the best threshold level that will differentiate the object from the background after segmentation, i.e., the object will be all ones and the background all zeros, or vice versa. The figure of merit for determining the best threshold level is how well the edges of the object in the segmented image match the areas of high gradient in the original image, i.e., areas with a large rate of change in pixel values among neighboring pixels. An edge in the segmented image is defined as a zero-to-one or one-to-zero transition in pixel values. Thus, a pixel in the segmented image is considered an edge pixel if it has at least one neighboring pixel of opposite value. The gradient at each pixel in the original image is found by applying the Sobel operator [10]. The figure of merit is given by the average gradient (Sobel value) per edge pixel generated by the given threshold. A threshold value with a large figure of merit generates edges that highly correspond (on the average) to areas with large gradients in the original image. Thus, the threshold with the largest figure of merit is the threshold used to segment the image. The EGT algorithm consists of four conceptual steps:

1. use the Sobel operator to calculate the gradient for each pixel
2. determine the figure of merit of each possible threshold (1 to 255)
3. determine the threshold level with the largest figure of merit
4. segment the image based on the threshold determined in step (3).

The following subsection discusses these steps in more detail.

3.2. The Basic EGT Algorithm

This subsection describes the basic algorithm from [17] that achieves the desired effect presented in the above conceptual outline. Although a serial algorithm is described, only the slight modifications described in the next subsection are required to create the parallel implementation. The algorithm assumes that the \( M \times M \) image to be processed is contained in the two-dimensional array \( \text{PIXEL}(i, j) \), where \( 0 \leq i, j \leq M - 1 \). After the optimum threshold is found, it is used...
\[ S_z = \frac{1}{4} \left[ \begin{array}{c} \text{PIXEL}(i-1,j-1) + 2 \text{PIXEL}(i,j-1) + \text{PIXEL}(i+1,j-1) \\ \text{PIXEL}(i-1,j) + \text{PIXEL}(i,j) + \text{PIXEL}(i+1,j) \\ \text{PIXEL}(i-1,j+1) + 2 \text{PIXEL}(i,j+1) + \text{PIXEL}(i+1,j+1) \\ \end{array} \right] \]

\[ S_y = \frac{1}{4} \left[ \begin{array}{c} \text{PIXEL}(i-1,j-1) + 2 \text{PIXEL}(i-1,j) + \text{PIXEL}(i-1,j+1) \\ \text{PIXEL}(i,j-1) + \text{PIXEL}(i,j) + \text{PIXEL}(i,j+1) \\ \text{PIXEL}(i+1,j-1) + 2 \text{PIXEL}(i+1,j) + \text{PIXEL}(i+1,j+1) \end{array} \right] \]

\[ \text{GRADIENT} = |S_z| + |S_y| \]

**FIG. 1.** Sobel operation for \( \text{PIXEL}(i,j) \).

To segment the image and the result is placed in the two-dimensional array \( \text{SEG-PIXEL}(i,j) \), where once again \( 0 < i, j < M - 1 \).

To determine the optimum threshold, several quantities are computed for each pixel in the image. Applying the Sobel operator to a given pixel, \( \text{PIXEL}(i,j) \), involves a calculation based on the pixel and its eight nearest neighbors. The Sobel operation first calculates two components of the gradient \( (S_x \text{ and } S_y) \), using the equations given in Fig. 1. The two components of the gradient can be combined into a single value for each pixel and stored in \( \text{GRADIENT} \) by the equation given in Fig. 1. This equation can be implemented with integer arithmetic (as opposed to floating point).

As each pixel is processed, statistics are updated for each possible threshold, \( T (1 \leq T \leq 255) \). The statistics are stored in two arrays, \( \text{EDGE-PIXEL-COUNT}(T) \) and \( \text{EDGE-GRADIENT-SUM}(T) \). The number of edge pixels that would be produced by segmenting the image with a threshold value of \( T \) is stored in \( \text{EDGE-PIXEL-COUNT}(T) \), and the sum of the gradients corresponding to these edge pixels is stored in \( \text{EDGE-GRADIENT-SUM}(T) \). After all the pixels have been processed, the statistics are used to calculate the figure of merit, \( \text{FOM} \), for each \( T \).

The complete algorithm, given in Fig. 2, consists of four separate loops. The first loop initializes the threshold statistic arrays. Next, the doubly nested main loop performs the following four steps for each \( \text{PIXEL}(i,j) \), \( 0 < i, j < M - 1 \).

As each pixel is processed, statistics are updated for each possible threshold, \( T (1 \leq T \leq 255) \). The statistics are stored in two arrays, \( \text{EDGE-PIXEL-COUNT}(T) \) and \( \text{EDGE-GRADIENT-SUM}(T) \). The number of edge pixels that would be produced by segmenting the image with a threshold value of \( T \) is stored in \( \text{EDGE-PIXEL-COUNT}(T) \), and the sum of the gradients corresponding to these edge pixels is stored in \( \text{EDGE-GRADIENT-SUM}(T) \). After all the pixels have been processed, the statistics are used to calculate the figure of merit, \( \text{FOM} \), for each \( T \).

The complete algorithm, given in Fig. 2, consists of four separate loops. The first loop initializes the threshold statistic arrays. Next, the doubly nested main loop performs the following four steps for each \( \text{PIXEL}(i,j) \), \( 0 < i, j < M - 1 \).

**FIG. 2.** Basic EGT algorithm.
imum pixel value, $\text{LOCAL-MIN}$, among the $3 \times 3$ window of pixels around the $\text{PIXEL}(i, j)$ are determined, i.e., the minimum and maximum among $\text{PIXEL}(i, j)$ and its eight nearest neighbors. Only for threshold values between $\text{LOCAL-MIN} + 1$ and $\text{LOCAL-MAX}$ would some pixels in the window be changed to zeros and some to ones, creating an edge (a zero-one or one-zero transition) involving $\text{PIXEL}(i, j)$. The final step in the main loop updates the window be changed to zeros and some to ones, creating $\text{EDGE-GRADIENT-SUM}(T)$, for $\text{LOCAL-MIN} + 1$ to $\text{LOCAL-MAX}$.

Once all the pixels are processed, the statistics collected for the threshold values are used to calculate the figure of merit, for each possible threshold value. The figure of merit is computed as the sum of the edge pixel gradients divided by the number of edge pixels produced from the threshold, giving the average gradient per edge pixel. As the figures of merit are computed, the threshold with the largest figure of merit is determined and stored in the variable $\text{BEST-THRESHOLD}$. That threshold is used in the final loop to segment the image.

For the sake of efficiency, the two-dimensional arrays $\text{PIXEL}(i, j)$ and $\text{SEG-PIXEL}(i, j)$ are implemented as one-dimensional arrays. Pointers are used to access the array.

### 3.3. Parallel Implementation of EGT

The algorithm described above can be easily implemented on a parallel processing system with $N$ PEs. The $N$ PEs are configured logically as a $\sqrt{N} \times \sqrt{N}$ grid. The $M \times M$ image is divided into $N$ subimages, each of size $L \times L$ pixels, where $L = M/\sqrt{N}$. Each subimage is loaded into the local memory of a PE (assuming a distributed-memory system), and each PE processes its own local subimage. The algorithm in Fig. 2 must be altered by reducing the bounds of the index variables $i$ and $j$ so that $0 \leq i, j < L - 1$. The combined segmented subimages will match the segmented image produced by a serial system that generates an individual threshold for each subimage.

Implementation on a distributed-memory system requires an additional step to resolve the problem that arises when the pixels on the subimage borders are processed. Because the Sobel, local minimum, and local maximum operations are based on a $3 \times 3$ window around $\text{PIXEL}(i, j)$, some pixels needed when processing the subimage border points are contained in an adjacent PE's local memory. Inter-PE communication must take place to transfer these subimage border points. Also, each corner point must be transferred to the PE containing the corresponding corner subimage. In general, $4(L + 1)$ pixels must be transferred for each PE. It is assumed that all PEs may transfer corresponding subimage border pixels simultaneously. If all these transfers are performed prior to the main EGT loop, the data will be available when needed.

### 4. PHASE EVALUATION

This section presents several implementations for each phase of the EGT algorithm on a mixed-mode architecture. The phases are transfer of subimage border points, Sobel calculation, local minimum/maximum determination, threshold statistics update, figure of merit calculation, and segmentation. The goal of this discussion is to optimize the implementation of each phase in isolation. Particular attention is paid to the selection of parallelism modes and its effect on performance.

#### 4.1. Transfer of Subimage Border Points

The first phase of the parallel EGT algorithm transfers the subimage border points to PEs containing the neighboring subimages through messages passed via the interconnection network. The interconnection network is assumed to be circuit-switched. Although it is possible for communication paths to be blocked, it is assumed that the interconnection network is robust enough to support the permutations needed for a logical $\sqrt{N} \times \sqrt{N}$ grid of PEs, as is the case for the extra-stage cube network in PASM [22, 24].

It may be possible to eliminate this phase by distributing the border pixels when the image is first loaded into the PEs' local memories. But, in general, the EGT algorithm may be an intermediate step within an image-processing task, and the input data for the EGT algorithm is the output of an earlier step. For example, an image may be smoothed prior to executing the EGT algorithm, so that each PE contains a smoothed subimage, necessitating the transfer of border points. Furthermore, the initial loading of the image is not considered in this study as it equally affects both SIMD and MIMD modes.

During this phase every PE is both a source and a destination, and all the PEs concurrently send and store the data from/to the same relative pixel position within their subimages. For example, consider the case when the PEs are transferring their left subimage border to their left neighbors. When every PE establishes an outgoing path to its left neighbor, an incoming path is established by its right neighbor. For each pixel transferred, all the PEs write their pixel value to their left neighbor and then read the value sent from their right neighbor. PEs that contain subimages that are on the border of the entire image are a special case and must be handled appropriately. Transmitting a message from PE A to PE B consists of the following steps:

1. PE A establishes a communication path to PE B
2. PE A sends pixel value into the network
3. PE B waits for data to be received
(4) PE B reads pixel value from the network and stores it in memory
(5) repeat steps (2) through (4) for each pixel to be sent from PE A to PE B
(6) PE A relinquishes the communication path to PE B.

In addition to exchanging border pixels with its top, bottom, left, and right neighbors, the PE must also transfer the four corner points to the catercorner PEs. An efficient technique transfers the corner pixels to a PE through the four neighboring PEs mentioned above within the messages that transfer their border points. This minimizes the number of network settings needed in this situation. In general, in a circuit-switched network, step (3) above is negligible, while step (1) is not. However, step (1) occurs only four times and this latency could possibly be performed concurrently with other computation.

Three possible implementations of this phase are as follows: the PEs can be in SIMD mode, in MIMD mode using a polling scheme, or in MIMD mode with explicit synchronization between transfers. The key difference among these three implementations is the way in that the PEs know when data are available to be read from the network, i.e., step (3) above.

The SIMD algorithm is very efficient because the PEs are implicitly synchronized. After the source PE sends data into the network, only a small deterministic delay is needed before the destination PE can read the data. Because SIMD programs have only a single instruction stream, this scheme is easily implemented. The PEs do not need to poll the network port to determine whether data have arrived.

An additional advantage of the SIMD version appears because the PEs execute a loop for each border to be transferred. In SIMD mode, the CU can overlap the loop control instructions. The CU can also overlap the calculation of the two pixel addresses needed by the PEs, i.e., the address of the pixel to be transferred and the address of where to store the incoming pixel. The broadcasting of the addresses to the PEs incurs a slight, but noticeable, overhead.

When this phase is implemented in MIMD, PEs are not necessarily synchronized even though they are executing the same sequence of instructions. Therefore, each PE must poll the network to verify that data have arrived before reading its network port. The polling introduces overhead not present in the SIMD version.

The third alternative is to implement the phase in MIMD, but instead of polling the network, the PEs explicitly synchronize after the data have been written into the network. The performance of this implementation depends upon the synchronization overhead. One method of explicitly synchronizing when the PEs are in MIMD mode utilizes the SIMD instruction queue to produce a barrier synchronization [19] among the processors. A barrier synchronization can be achieved by accessing a single word from the logical address space corresponding to the CU instruction queue [8]. This method of synchronization is very efficient and incurs very little overhead, i.e., the time for a single reference to the CU instruction queue.

One final alternative, which was not implemented, involves the use of an interrupt-driven scheme in MIMD, where a PE is interrupted when a data element has been received. The interrupt handler would read the data and place them in the proper buffer. An interrupt would also be generated when the outgoing data buffer is empty, allowing a different interrupt handler to send data appropriately. Thus, for each data element transferred, two interrupts would need to be processed. With an efficient interrupt scheme, this would approach the time for the MIMD polling implementation.

The graph in Fig. 3 shows the performance of the SIMD and the two MIMD implementations of this phase when executed on the PASM prototype. The performance of each algorithm for various subimage sizes is shown. Due to the overhead of polling and the lack of CU overlap, the MIMD version with polling takes approximately twice as long to execute as the SIMD version. The barrier-synchronized MIMD version is better than the polling MIMD version because the polling overhead has been eliminated. But, the barrier-synchronized MIMD version is 25% slower than the SIMD version due to the CU overlap in SIMD.

The asymptotic time complexity of this phase is $O(4L + 4) = O(L)$. As the subimage size grows, the constant factor of establishing and relinquishing the communication paths diminishes in proportion to transfer of the data. As the transfer time becomes dominant over the constant path establishment time, the performance gap between the SIMD and MIMD versions widens.

4.2. Main EGT Loop

The next three phases, Sobel calculation, local minimum/maximum, and the threshold statistics update, are the central
portion of the EGT algorithm. These phases are executed once for each pixel in the subimage and therefore have a time complexity of \(O(L^2)\).

4.2.1. Sobel Calculation

The Sobel phase has four major components, as seen in Fig. 1 and lines 5 through 7 in Fig. 2. First, the loop control must be performed for both the \(i\) and the \(j\) loops. Second, the addresses of the appropriate elements in the PIXEL array must be computed. These addresses are used in the third step to calculate \(S_x\) and \(S_y\). Finally, \(S_x\) and \(S_y\) are combined to form the gradient. Each of these four basic operations can take place in either SIMD or MIMD mode. For example, the loop control can be performed on the PEs in SIMD mode or overlapped on the CU in SIMD mode.

The computation of the PIXEL addresses is independent of the data on each PE and potentially can be executed on the CU in SIMD mode. An efficient technique is to use a pointer to each position in the 3x3 window. Instead of computing the address of the pixels in each new window, the pointers need only be incremented appropriately. There must be nine pointers, one for each position in the window, requiring only nine additions in each iteration to compute the addresses. These additions can be performed on the PEs in MIMD mode, or overlapped on the CU in SIMD mode and the results broadcast to the PEs.

The computation of \(S_x\) and \(S_y\) uses only simple mathematical operations, independent of the actual data values. Thus, once the addresses are calculated for this portion there is no difference in execution time when they are implemented in either parallelism mode. On the contrary, when the two components are combined into a single gradient, data-conditional statements are needed to form the absolute values of \(S_x\) and \(S_y\). In general, data-conditional statements execute differently in each mode. Forming the absolute value in MIMD mode is straightforward using a simple if-then-else statement.

Data-conditional statements can also be executed in SIMD, but the efficiency of such operations is machine implementation dependent. The present implementation of the PASM prototype uses a global masking scheme to disable PEs. With global masking each SIMD instruction has a corresponding PE enable mask consisting of one bit for each PE. For a given instruction, the PEs whose corresponding bit in the PE enable mask is a 1 will execute the instruction. Global masking can be used in conjunction with the condition codes to execute data-conditional statements. The condition codes are used by the PEs to transmit their status to the CU. An if-then-else statement is implemented by performing the “if” test on local data in the PEs, transmitting the results of the tests to the CU via the condition codes, and using the condition codes to set the PE enable mask for the “then” statements. The PE enable mask for the “else” statements can also be derived from the same condition code bits.

In general, the PE enable masking scheme is not very efficient for data-conditional statements because of the synchronization needed between the CU and PE. An improved scheme in which PEs disable themselves with local PE masking, as used in the MPP [3, 4], MasPar [7], and the Connection Machine [27], can be emulated on PASM [21]. With the local PE enable, each PE contains a register that can be loaded with its condition code value. The PE will be enabled or disabled depending on the value of its local enable register. Unlike in global masking, the CU does not read the condition codes to form the mask; consequently the data-conditional statements become much more efficient. Although PASM is not presently implemented with local PE enable bits, the performance of this phase can be emulated on PASM by removing the instructions that perform the PEs’ sending of condition codes to the CU. Although functionally the program computes incorrect results, the timing measurements give a fair estimate of the timing of a PE local enable scheme.

An alternative method of implementing the if-then-else statement in SIMD mode is to use the boolean function. The boolean function takes a conditional test as an argument and returns either 0 (i.e., 000...00 in binary) if the test evaluates to false or -1 (i.e., 111...11 in binary) if the test evaluates to true. The boolean function does not use a branch statement and therefore can be executed by the PEs in SIMD mode. The boolean function can be used to implement the absolute value function as follows, where “boolean \((S_x < 0)\)” returns a value of -1 (11...11 in binary) if \((S_x < 0)\) and it returns a value of 0 if \((S_x \geq 0)\):

\[
B = \text{boolean}(S_x < 0)
\]

\[
S_x = (B \text{ xor } S_x) - B.
\]

The function “xor” performs the bitwise exclusive-or. When \((S_x < 0)\), \(S_x\) is loaded with the two’s complement of its old value. This follows from the facts that two’s complement arithmetic is used, \(B \text{ xor } S_x = \overline{S_x}, -B = \overline{(-1)} = +1\), and \(\overline{S_x} + 1 = -S_x\). This method is well suited to SIMD mode because the operations can be performed independent of the data, thereby eliminating any data-conditional statements.

The graph in Fig. 4 shows the execution times for the five Sobel phase implementations with four PEs on the PASM prototype. The performance of each phase is plotted for different subimage border sizes. The MIMD version performs better than the pure SIMD implementation with global masking. This comes from the inefficiency of the global masking during the absolute-value calculation. By replacing the masking with the boolean function, a version that is about 20% better than the MIMD version is created. The improvement comes from the overlapping of loop indexing opera-
tions in SIMD. Simulating a local PE enable scheme shows a 5% additional improvement because the boolean version contains additional instructions (the exclusive-or and subtract) that are not in the other versions. Finally, a mixed-mode approach with all work performed in SIMD, except the absolute value, gives a result similar to that of the SIMD boolean version. The implementation of C used to code these experiments adds a small amount of overhead when switching modes, i.e., the cost of switching from SIMD to MIMD is a C function call. (A new compiler that avoids the function call is under development.) Although switching modes requires only a small amount of overhead, any additional time is noticeable in this loop. The mixed-mode overhead approximately equals the time to execute the exclusive-or and subtract instructions in the SIMD boolean version.

In summary, the experimentation shows that the best implementation of the Sobel phase is SIMD mode with a local PE enable scheme, if available. Otherwise, the use of the boolean function in SIMD mode or MIMD mode should be employed to compute the absolute value.

4.2.2. Local Minimum/Maximum Determination

The calculation of the local minimum and maximum involves the determination of the smallest and largest values of the nine pixels in a 3 × 3 window. This phase requires eight data-conditional statements for each of the operations.

Three SIMD implementations have been created and tested on the PASM prototype. In the first version, global PE masking is used for each if-then statement. Second, a PE local enable mechanism can be emulated, as described in the previous subsection. Finally, the boolean function can be used in place of each if-then statement, as demonstrated below, where “and” and “not” are bitwise functions:

\[ B = \text{boolean}(\text{PIXEL}(i, j) > \text{LOCAL\_MAX}) \]

\[ \text{LOCAL\_MAX} = (B \text{ and } \text{PIXEL}(i, j)) \]

\[ + ((\text{not } B) \text{ and } \text{LOCAL\_MAX}). \]

The execution times for the three implementations described above and a MIMD implementation are plotted in Fig. 5, assuming that the \texttt{LOCAL\_MIN} and \texttt{LOCAL\_MAX} calculations in Fig. 2 are executed \(L^2\) times. The MIMD version achieved the best performance because in MIMD the PEs are independent, allowing each PE to execute the “then” statement or not, depending on the data in its local memory. Those PEs that do not execute the “then” may continue on to the next “if” statement without waiting for the other PEs to finish their execution of the “then.” SIMD mode is less efficient when a sequence of conditionals is executed because when a PE does not need to execute the “then” statement, it must remain idle until the other PEs have completed it. The time for a “then” statement is incurred if at least one PE needs the instructions.

In a realistic scenario, the MIMD execution time would be lower than that shown in Fig. 5. The data used in the experimentation forced one-half of the “then” statements in this phase to be executed. In a realistic data set, less than one-half of the “then” statements would be taken (on the average), causing the MIMD execution time to be reduced. Also, the experimentation included loop indexing needed to measure \(L^2\) passes. The basic EGT algorithm given in Fig. 2 shows the three phases of the main EGT loop in the same doubly nested loop. When the algorithm is implemented as a whole, the loop index operations and the address calculations needed to access the 3 × 3 window into the subimage are performed only once for each pixel. When experimenting with the phases, in isolation each phase contains its own doubly nested loop. Because the execution time for the loop control was accounted for in the Sobel phase experimentation, this time can be removed from the local min/max experimentation results. In SIMD mode, the additional loop control instructions introduced in the experimentation were overlapped on the CU and therefore did not affect the total execution time. However, the MIMD-mode execution experiment time shown in Fig. 5 was increased by the indexing operations. Thus, in reality, the MIMD time would again be reduced if the “cost” of indexing were shared by multiple subroutines.
The SIMD version that emulated the PE local enable scheme performed considerably better than the global masking version for reasons described in the previous subsection. The implementation using the boolean function has a performance between those of the two other SIMD versions due to the additional instructions needed in the boolean implementation, i.e., the "and" and "not" instructions.

4.2.3. Threshold Statistics Update

The threshold statistics update (TSU) routine is the most important phase of the EGT algorithm because it contains the innermost loop and may execute for more time than the other phases (see Fig. 2). Because its flow of control depends upon the PE data, it is shown that this phase is best implemented in MIMD.

During each invocation of the TSU phase, the $\text{EDGE\_GRADIENT\_SUM(T)}$ and $\text{EDGE\_PIXEL\_COUNT(T)}$ arrays must be updated for each $T (\text{LOCAL\_MIN} < T \leq \text{LOCAL\_MAX})$. This requires each PE to loop through the arrays with the index beginning at the local minimum plus one and finishing at its local maximum.

The MIMD implementation is given in lines 15 through 18 of Fig. 2. Because the PEs are independent, each PE is able to begin and end its loop on the basis of its local data.

This phase is not well suited to SIMD because each PE may need to perform a different number of iterations on the basis of its own $\text{LOCAL\_MIN}$ and $\text{LOCAL\_MAX}$ values. The CU must issue a single instruction stream that will allow each PE to iterate the proper number of times.

Let $D_p(i, j) = (\text{LOCAL\_MAX} - \text{LOCAL\_MIN})$ when $\text{PIXEL}(i, j)$ is processed on PE $P$. In MIMD mode, PE $P$ performs $D_p(i, j)$ iterations of the TSU loop and then continues processing the next pixel. In SIMD mode, the CU must broadcast the loop body at least $\max_p[D_p(i, j)]$ times, $0 < P < N$, resulting in two performance penalties. First, some PEs may be idle, waiting for other PEs to execute this required number of iterations. Second, the CU must somehow determine $\max_p[D_p(i, j)]$, $0 < P < N$.

One way to implement this in SIMD is to have the CU loop until all the PEs are done, as shown in Fig. 6. Each PE contains its own local counter, $T$, that is initialized to the value of $\text{LOCAL\_MIN}$. During each iteration, if a PE's local counter is less than its $\text{LOCAL\_MAX}$, the corresponding threshold's statistics are updated. Otherwise, the PE becomes disabled and will not execute the remaining iterations. The CU continues to broadcast the loop body until no more PEs are enabled. Performing the data-conditional statement with a PE local enable mask or boolean function mechanism will not be an advantage over a global mask in this situation because the CU must receive the condition code bits for the "while" statement before broadcasting the loop body.

An alternative SIMD implementation, shown in Fig. 7, removes the need to transfer the condition code bits from the PEs to the CU. Instead, the CU always iterates 255 times, i.e., the maximum number of iterations. The PEs can then use the boolean function to update the threshold statistics appropriately. On PEs where $\text{LOCAL\_MIN} < T \leq \text{LOCAL\_MAX}$, the boolean function sets $B = 11 ... 111 = -1$, so that with two's complement arithmetic, $-B = +1$. On PEs where $T$ is outside this range, $B$ is set to zero.

The third SIMD implementation, shown in Fig. 8, uses the interconnection network to find the maximum iteration count among the PEs. This iteration count is transferred to the CU, which uses it as the loop index bound. The PEs use the boolean function to update their threshold statistics, similar to the previous SIMD implementation described. This method is potentially beneficial because only a single data transfer is needed from the PEs and CU, and only the number of iterations needed is executed. A penalty is incurred by the need to determine the maximum iteration count among the PEs via the interconnection network. Using a recursive doubling [25] scheme, this requires $\log_2 N$ inter-PE data transfers (for $N$ PEs) that are not needed in the other SIMD implementations.

The graph in Fig. 9 shows the performance of each of the implementations discussed above. The execution time of each version is plotted versus $D = D_p(i, j)$. These data were collected by executing the TSU phase for a single pixel $D = D_p(i, j)$ in all PEs $P$. As predicted, the MIMD implementation performs best for all values of $D$. The SIMD implementation that always iterates 255 times requires a constant amount of time, independent of the data. Compared to this version, the implementation with the "while" statement in SIMD mode attains a better performance for $D < 120$. As the value of $D$ increases, the overhead of testing the condition codes in the "while" implementation is larger than the time to perform the additional $255 - D$ iterations executed in the constant loop of 255 iterations.

The recursive doubling version is much slower because of
LIMITATIONS ON MIXED-MODE DUE TO JUXTAPOSITION

\[ \text{LOCAL}_D = \text{LOCAL}_{\text{MAX}} - \text{LOCAL}_{\text{MIN}} \]

\[ \text{MAX}_D = \max_{\text{among PE}}(\text{LOCAL}_D) \quad \text{/* use recursive doubling */} \]

\[ \text{transfer from PE to CU}(\text{MAX}_D) \]

\[ \text{LOCAL}_T = \text{LOCAL}_{\text{MIN}} + 1 \]

for \( T = 1 \) to \( \text{MAX}_D \) {
  \[ B = \text{boolean}(\text{LOCAL}_T < \text{LOCAL}_{\text{MAX}}) \]
  \[ \text{EDGE}_4 \text{GRADIENT}_4 \text{SUM}(\text{LOCAL}_T) += (\text{GRADIENT} \text{ and } B) \]
  \[ \text{EDGE}_4 \text{PIXEL}_4 \text{COUNT}(\text{LOCAL}_T) += -B \]
  \[ \text{LOCAL}_T += 1 \]
}

FIG. 8. SIMD algorithm with recursive doubling for TSU phase.

the PASM prototype’s inefficient mechanism used to transfer data from the PEs to the CU. The dotted curve in Fig. 9 labeled “SIMD(ret daub)**” corresponds to the recursive doubling implementation without the approximately 65,000 clock cycles needed for the PE-to-CU data transfer and is included in an attempt to circumvent these artifacts of the prototype implementation details. Because the execution time of the recursive doubling function is \( O(\log_2 N) \), the \( N = 4 \) curve representing this scheme would be raised for larger \( N \), while the other curves remain constant. Each doubling in the number of PEs requires an additional step in the recursive doubling algorithm. To account for the extra network setup and data transfers, an additional 2000 clock cycles (approximately) are required for each doubling of the PE count above 4. For example, the curve for a 1024-PE configuration would be raised by 16,000 clock cycles (independent of \( D \)).

In summary, the TSU phase is best implemented in MIMD. The juxtaposition of this phase with the remaining program is analyzed in detail in Section 5.

4.3. Figure of Merit Calculation

The figure of merit phase determines the threshold that achieved the highest average gradient per edge pixel. Each PE calculates a \( \text{BEST}_4\text{THRESHOLD} \) for its subimage. This is a simple routine that contains a loop with 255 iterations independent of the image size or number of PEs (see Fig. 2). This phase has asymptotic time complexity of \( O(1) \) and does not greatly influence the overall execution time of the EGT algorithm and therefore is not discussed in detail.

4.4. Segmentation

The segmentation phase involves setting each pixel in the segmented image to either zero or one, depending on the value of the corresponding pixel in the original image in comparison to the threshold level. The PEs can perform this operation on their subimages concurrently, so the asymptotic time complexity is \( O(L^2) \). The straightforward MIMD implementation is given in lines 27 through 32 of Fig. 2.

The MIMD algorithm can be implemented in SIMD, but the performance would be poor because the if-then-else statement would be serialized. A better implementation, given in Fig. 10, removes the conditional statement. A segmentation table, \( \text{SEG}_4\text{TABLE}(V) \), that has an entry for each possible pixel value \( V, 0 < V < 255 \), is created. The entry for \( V \) in the segmentation table is the corresponding segmentation value of a pixel with value \( V \); i.e., if \( V < \text{BEST}_4\text{THRESHOLD} \) then \( \text{SEG}_4\text{TABLE}(V) = 0 \), and otherwise \( \text{SEG}_4\text{TABLE}(V) = 1 \). The table is first created using the boolean function, and then each pixel in the image is segmented by indexing into the table (a very quick operation).

The graph in Fig. 11 displays the performance measurements of the straightforward algorithm implemented in MIMD and the table method implemented in MIMD and SIMD. Their relative performance is influenced by three factors. First, there is \( O(1) \) overhead to initialize the table when the table method is used. Second, the number of instructions in the body of the segmentation loop is slightly less in the table method, causing the straightforward MIMD imple-

\[
\text{for } V = 0 \text{ to 255} \quad \text{SEG}_4\text{TABLE}(V) = -\text{boolean}(V \geq \text{BEST}_4\text{THRESHOLD}) \\
\text{for } i = 0 \text{ to } L-1 \{ \\
\text{for } j = 0 \text{ to } L-1 \{ \\
\quad \text{SEG}_4\text{PIXEL}(i,j) = \text{SEG}_4\text{TABLE}4\text{PIXEL}(i,j) \\
\} \} \]

FIG. 10. Table method of segmentation.

\[
[\text{LOCAL}_{\text{MAX}} - \text{LOCAL}_{\text{MIN}}] \text{ in all PEs} \\
\]

FIG. 9. Execution time for TSU phase.
5. EVALUATION OF MODE SELECTIONS

In this section, both experimental and analytical results are given. The analyses are based on the interaction of algorithm characteristics with inherent properties of SIMD, MIMD, and mixed-mode parallelism. These analyses attempt to present the relative performance of these three modes in a machine-independent way to as great an extent as possible. The goal is to abstract away implementation details of the PASM prototype and concentrate on inherent properties of the modes.

5.1. Phase-Optimized Implementation

This section discusses several implementations of the complete EGT algorithm. A phase-optimized version based on the results in Section 4 is compared to a pure MIMD and several pure SIMD implementations.

The results discussed in Section 4 determine the mode selection for each phase of the EGT algorithm that produces the best performance of that phase in isolation. This phase-optimized implementation has the transfer of subimage border points, Sobel, and segmentation phases in SIMD. The local minimum/maximum determination, TSU, and figure of merit phases are implemented in MIMD. Because the loop control for the main EGT loop is considered part of the Sobel phase, it is executed on the CU in SIMD and can be overlapped with the PEs' execution of the loop body (i.e., the Sobel, local minimum/maximum, and TSU phases).

5.2. Experimental Data Sets

The experimental data presented in this paper have been collected by measuring execution times on the PASM prototype. Three test images were selected to evaluate the performance of the different EGT implementations. These images are obviously atypical; however, they were selected to provide a variety of testing conditions that can be precisely analyzed.

Test Image I, shown in Fig. 12, consists of a square object immersed in a background. The difference between the gray levels of the object and the background is given by the parameter $D(\text{edge})$ ($0 < D(\text{edge}) < 255$). For an $M \times M$ image, the object has dimensions $M/2 \times M/2$ and is centered within the background. The data are distributed such that each PE processes $L \times L$ pixels, including a $L \times 2 \times L/2$ corner of the object.

Test Image II, shown in Fig. 13, consists of a series of stripes. Each stripe is two pixels wide and has a gray level of $D(\text{edge})$, and the background has a gray level of zero. Each subimage has a series of $L/16$ stripes separated by 14 rows of zero-valued pixels. There is a one-pixel wide border of zero-valued pixels on all sides of each subimage. The stripes are organized such that when divided, the four subimages are identical. Test Image III, in Fig. 14, is similar to Test Image II, except the stripes in each subimage are skewed by
LIMITATIONS ON MIXED-MODE DUE TO JUXTAPOSITION

four pixels so that no two subimages contain a stripe in the same relative position. If the four subimages were to be laid over one another, no stripes would overlap and two rows of zero-valued pixels would separate each stripe.

Test Images II and III were selected to study the effect of the parameter $D_p(i, j)$, the TSU iteration count defined in Section 4. In Test Image II, for each $PIXEL(i, j)$, $D_p(i, j)$ is the same for all $P$. Test Image III contains the same number of nonzero pixels as the second test image, but for each pixel position, $i, j$ in the subimage, exactly one subimage has a nonzero $D_p(i, j)$. As demonstrated in the following subsections, performance varies considerably over these three test images.

Tests were performed using four sizes of each test image. Image sizes of $64 \times 64$, $128 \times 128$, $256 \times 256$, and $512 \times 512$ pixels were partitioned into $32 \times 32$, $64 \times 64$, $128 \times 128$, and $256 \times 256$ subimages for four PEs, respectively. Although the experimentation presented here involved four PEs, the same conclusions would be obtained from a 1024-PE or larger machine. Because of the nature of the EGT algorithm, the execution time is dependent upon the subimage data distribution and subimage size and, in general, not the total number of PEs (e.g., in a 1024-PE system each PE still communicates only with its nearest neighbors).

5.3. Phase-Optimized Mixed-Mode vs Pure MIMD

5.3.1. Analysis of Test Image I Performance

The graph in Fig. 15 plots the execution times of two complete EGT algorithm implementations when various sizes of Test Image I are processed. This subsection compares and analyzes the performance of the phase-optimized and pure MIMD implementations. As seen from the graph, the phase-optimized version for Test Image I is slower than the pure MIMD version for all subimage sizes tested. For reasons discussed below, as the image size increases the margin between the execution times of the two implementations diminishes.

There are several opposing factors that influence the relative performance of the programs. Figure 16 shows the breakdown by phases for Test Image I of size $M = 64$ ($L = 32$). The CU overlaps the pixel pointer incrementing in the phase-optimized version, while that time is included in the Sobel section of the pure MIMD version. The transferring of subimage border points, Sobel, and segmentation phases are shown to be more efficient in SIMD, an advantage for the phase-optimized program. This advantage does not outweigh the relatively large amount of time lost during the TSU phase. The TSU phase is implemented in MIMD mode in both programs, so why does its execution appear to take longer in the phase-optimized version? The answer lies in what happens after the TSU phase.

The phase-optimized program executes the TSU phase for a given $PIXEL(i, j)$ in SIMD mode and then returns to SIMD mode to execute the Sobel phase for the next pixel. As discussed in Section 4, when PE $P$ processes $PIXEL(i, j)$ in MIMD mode, the time to execute the TSU phase is proportional to $D_p(i, j)$, the number of iterations PE $P$ executes in the TSU loop when processing $PIXEL(i, j)$. Because entering SIMD mode implies synchronization among all of the PEs, when $D_p(i, j)$ differs among the PEs for a given pixel position, some PEs remain idle, waiting for the PE with the largest $D_p(i, j)$ to finish the TSU phase. The effective iteration count for all the PEs becomes $\max_P[D_p(i, j)]$, $0 \leq P < N$. For a given $PIXEL(i, j)$, the TSU phase consists of incrementing two arrays of variables $D_p(i, j)$ times. Let $T_{inc}$ be the time to increment a single variable. In general, when the $L \times L$ subimages are processed using the phase-optimized implementation, the total time spent executing the TSU phase, $T_{PO}(TSU)$, is given as

$$T_{PO}(TSU) = \sum_{i=0}^{L-1} \sum_{j=0}^{L-1} \left[ \max_P[2T_{inc} \times D_p(i, j)] \right], \quad 0 \leq P < N.$$
phase in the pure MIMD implementation, $T_M(TSU)$, is the maximum among the PEs to independently process the entire subimage, or

$$T_M(TSU) = \max \left\{ \sum_{i=0}^{L-1} \sum_{j=0}^{L-1} [2T_{inc} \times D_p(i,j)] \right\}, \quad 0 \leq P < N.$$  

$T_{PO}(TSU)$ is said to have a "sum of max's" execution time for the TSU phase, whereas the $T_M(TSU)$ has a "max of sum's" execution time. The "sum of max's" is always greater than or equal to the "max of sum's," meaning $T_{PO}(TSU) \geq T_M(TSU)$. $T_{PO}(TSU)$ equals $T_M(TSU)$ only when, for each pixel, all PEs execute the identical number of iterations in the TSU phase, i.e., for any fixed $i$ and $j$, $D_p(i,j) = D_0(i,j), 0 \leq P < N$.

This phenomenon is analogous to the situation analyzed in [11] when the execution time for a sequence of SIMD instructions depends upon the data, e.g., the PASM prototype's integer multiply instruction. Because synchronization occurs after every instruction, similar to the synchronization after the TSU phase, a "sum of max's" execution time occurs for the sequence of SIMD instructions.

Most pixels in Test Image I have neighbors of the same value; i.e., $D_p(i,j) = 0$. The exception occurs when a PE is processing one of the $L + 1$ outside edge pixels of the object or one of the $L - 1$ inside edge pixels. These $2L$ pixels in each subimage have $D(edge) = 80$, the value of $D_p(i,j)$, where PIXEL$(i,j)$ is on an edge of the object. In Test Image I, $D(edge)$ is the same for all edge pixels and all PEs.

When the PEs are executing the phase-optimized program, they are all concurrently processing the same pixel position. For Test Image I, the edge pixels are processed in pairs. When a PE is processing an edge pixel, at least one other PE is also processing an edge pixel. For most pixel positions, only two subimages contain an edge. For an example, refer to Fig. 17. When logical PE 0 and PE 1 are processing the edge pixel in the positions denoted by X's, PE 2 and PE 3 are not processing an edge. Likewise, when PE 0 and PE 2 are processing an edge pixel pointed to by the arrow, PE 1 and PE 3 are not.

In Test Image I, there is a difference in $D_p(i,j)$ among the PEs for all pixel positions on the edges of the object, except for the four pixels surrounding the corner in each subimage, labeled A, B, C, and D in the insert to Fig. 17. When a PE is processing one of these four pixels, the other PEs are processing a corresponding corner pixel in their subimages. There is a total of $4I$ pixel positions, where $D_p(i,j) = D(edge)$ for at least one PE, i.e., $\max_p[D_p(i,j)] = D(edge), 0 \leq P < N$, for $4L$ values of $i$ and $j$. The other pixel positions have $D_p(i,j) = 0$ for all $P$. Recall from above that for any single PE, only $2L$ pixels have $D_p(i,j) = D(edge)$, and the rest have $D_p(i,j) = 0$. Therefore $\left[ \sum_{i=0}^{L-1} \sum_{j=0}^{L-1} [2T_{inc} \times D_p(i,j)] \right] = 2L \times D_p(edge)$ for all PEs.

When the above information is combined with the equations for $T_{PO}(TSU)$ and $T_M(TSU)$, the execution time for their entire TSU phase can be given as

$$T_{PO}(TSU) = 4L \times 2T_{inc} \times D(edge)$$  
$$= 8L \times T_{inc} \times D(edge)$$  
$$T_M(TSU) = 2L \times 2T_{inc} \times D(edge)$$  
$$= 4L \times T_{inc} \times D(edge).$$  

The pure MIMD program executes the TSU phase for Test Image I approximately twice as fast as the phase-optimized version. From the breakdown of the phase execution times in Fig. 16, there are other phases that execute faster in the phase-optimized program. That particular figure has

FIG. 17. Example of processing edge pixels.
$D(\text{edge}) = 80$ and therefore shows the TSU phase dominating the execution time. But if $D(\text{edge})$ is decreased, the TSU phase has less effect on the total execution time. Figure 18 shows the effect of different $D(\text{edge})$ values on the total execution time for Test Image I of size $L = 32$. For large values of $D(\text{edge})$, the MIMD implementation executes substantially faster because the TSU phase dominates the execution time. But, for small values of $D(\text{edge})$, i.e., 1 or 2, the phase-optimized version performs best. With smaller values of $D(\text{edge})$, the TSU phase has less impact on the total execution time. As the other phases begin to dominate, the relative performance of the phase-optimized program increases.

The subimage size also changes the relative performance. The number of edge pixels in Test Image I is $O(L)$, so that as the subimage size grows at $O(L^2)$, the proportion of edge pixels to nonedge pixels decreases. As the image size increases, the other phases begin to dominate over the TSU phase in determining the total execution time. This causes the phase-optimized performance to approach that of the MIMD implementation as the image size increases, as shown in Fig. 15.

### 5.3.2. Analysis of Test Image II Performance

This subsection applies the analysis developed in the previous subsection to Test Image II. Every subimage of Test Image II consists of $L/16$ stripes, each 2 pixels wide. As with Test Image I, $D_p(i, j) = D(\text{edge})$ for all pixels, where $D_p(i, j) > 0$; however, for Test Image II, $D_p(\text{edge}) = 255$. Each stripe has $4L + 4$ edge pixels, giving a total of $(L/16)(4L + 4) = (L^2 + L)/4$ pixels in each subimage, where $D_p(i, j) = D(\text{edge})$.

The positioning of the stripes in Test Image II is a “best case” for the phase-optimized implementation because each subimage is exactly the same, making $T_{\text{po}}(\text{TSU}) = T_M(\text{TSU})$. Although some of the other phases are slightly faster in SIMD mode, the phase-optimized implementation has only negligible advantage because the TSU phase dominates the execution time in both programs. Figure 19 plots the execution time for various sizes of Test Image II, with $D(\text{edge}) = 255$. Experimentation with smaller values of $D(\text{edge})$ shows that the phase-optimized version can achieve a 10% performance advantage.

### 5.3.3. Analysis of Test Image III Performance

Test Image III contains the same number of edge pixels as Test Image II, but they are skewed such that only one PE is processing an edge at any given time. This represents the “worst-case” scenario for the phase-optimized program because all $L^2$ pixel positions have an edge pixel in only one subimage. When the phase-optimized program processes a given pixel position, only one PE executes $D(\text{edge})$ iterations in the TSU phase. The other PEs remain idle, effectively serializing this phase across the PEs. The total amount of time spent executing the TSU phase is the same as that in a serial program. The pure MIMD version processes Test Images II and III at the same rate because the execution time is independent of the relative position of edge pixels. As seen in Fig. 20, using four PEs to process Test Image III results in the phase-optimized version, which is about one-third the speed of the pure MIMD implementation.

### 5.4. Phase-Optimized Mixed-Mode vs Pure SIMD

Two pure SIMD implementations have been created and tested on the PASM prototype. The two versions have the best SIMD implementation of each phase with different TSU phases. One version has the TSU phase implemented with the constant 255-iteration loop given in Fig. 7. The other version contains the “while” version given in Fig. 6. For this test image, with $D_p(\text{edge}) = 255$, $\text{MAX}_D = 255$ for all pixels, and the Fig. 8 version would always be longer than the Fig. 7 version.

As seen in Figs. 15, 19, and 20, the phase-optimized implementation performs better than either of the pure SIMD versions for all three test images. These results occur because,
as shown in Subsection 4.2.3, the TSU phase is more efficiently implemented in MIMD. The SIMD versions also suffer from the "sum of max's" phenomenon in that, for a given \( \text{PIXEL}(i, j) \), the CU must broadcast the TSU loop body \( \max_p \{ D_p(i, j) \} \), \( 0 \leq p < N \), times for the Fig. 6 version and all 255 times for the Fig. 7 version.

5.5. Analysis

This section has shown that the relative performance of the phase-optimized and pure MIMD implementations of the EGT algorithm is data dependent. Both versions have their disadvantages. The main weakness of the pure MIMD version is its lack of CU overlap in the Sobel phase. Although the phase-optimized version exploits CU overlap, this requires the PEs to enter SIMD mode, forcing implicit synchronization. Synchronizing after every invocation of the data-dependent TSU phase results in the "sums of max's" phenomenon. This phenomenon has less impact as the variation of \( D_p(i, j) \) among the PEs decreases. The results were based on the use of both experiments and algorithm complexity analyses.

A hybrid of these two versions can be created by implementing the transfer of border points and segmentation phases in SIMD mode, and all three phases in the main loop, as well as the figure of merit loop, in MIMD mode. This version is essentially the pure MIMD version with the transfers and segmentation in SIMD. Because these phases play only a minor role in the total execution time of the algorithm, the new hybrid program performs less than 1% better than the pure MIMD version for the test images.

On the PASM prototype, fetching SIMD instructions is slightly faster than fetching MIMD instructions due to differences in the implementation technology of the PE memory and the instruction queue for PEs that is in the CU. This has only limited impact because the instruction execution and data fetching occur at the same speed in both modes. Furthermore, the fetching of the next instruction is often partially overlapped with the execution of the current instruction. Because the results presented in this section show that a pure MIMD implementation is better than a phase-optimized approach, taking into account the difference in instruction fetch time would only substantiate this conclusion.

Many other options for implementing the entire algorithm were considered, but performance was not improved over that of the phase-optimized and MIMD approaches. For example, separating the Sobel, local min max, and TSU phases into their own loops results in a lowered performance due to the overhead of storing and retrieving the data multiple times.

If the TSU phase were such that processors were required to communicate or synchronize between each successive pair of iterations, the choice of modes would be different. This inter-iteration data independence of the TSU is a key feature of this algorithm. Without this independence, mixed-mode performance would most likely be best.

6. SUMMARY AND CONCLUSIONS

While experiments involving other computational situations have shown the advantages of the mixed-mode capability (e.g., [12]), in the case of the EGT algorithm, it provides little improvement. The results presented here demonstrate the need for study of the problem of mapping algorithms to mixed-mode architectures. Clearly, there exists an inherent problem in the use of the phase optimization approach. When each phase of the EGT algorithm was optimized in isolation, an implementation of the entire algorithm that was less than optimal was created. This can potentially be true for any algorithm whose execution times are data dependent and whose subtasks can be performed on different processors without synchronization. In these cases, the most efficient mapping can be determined only by applying a more global approach that considers the impact of the temporal juxtaposition of code executed in different modes of execution. In addition, statistical information about the characteristics of the data may need to be incorporated into the mapping decision process.

As stated earlier, the purpose of this paper is to contribute to the understanding of mixed-mode parallelism. This involves exploring both the strengths and the weaknesses of this approach to massively parallel computation. The long-term goal of these studies is to develop an overall theory of mixed-mode computation that can be used to determine when and how mixed-mode parallelism can best be exploited.

ACKNOWLEDGMENTS

The authors thank Professor Thomas L. Casavant of the University of Iowa, Wayne G. Nation of Purdue University, and the reviewers for their many useful comments. A preliminary version of portions of this work was presented at the 1991 International Conference on Parallel Processing.
REFERENCES


