FEATURE ARTICLE

Distributed Edge Detection: Issues and Implementations

XIAODONG ZHANG AND SANDRA G. DYKES
University of Texas at San Antonio
HONG DENG
Diversified Technology

Experiments in parallelizing an edge detection algorithm on three representative message-passing architectures—a low-cost, heterogeneous PVM network, an Intel iPSC/860 hypercube, and a CM-5 massively parallel multicomputer—provide insight into implementation and performance issues for image-processing applications.

For computer vision and image-processing systems to successfully interpret an image, they must first be able to detect the edges of each object in the image. Such systems start by segmenting the image into regions, in a process called image segmentation. In the region-growing approach to segmentation, contiguous pixels with similar characteristics such as intensity or color are grouped together. Conversely, region-splitting segmentation methods locate region boundaries, or edges, where pixel values change abruptly. Edge detection, a region-splitting approach, produces an edge map that contains important information about the image. The memory space required for storage is relatively small, and the original image can be restored easily from its edge map. This method has proved both effective and powerful and is widely used in applications ranging from satellite imaging to medical radiology.

The importance of edge detection has led to the development of several algorithms for use on sequential computers in attempts to improve computational speed and accuracy.[1-3] Our research project—to take a complete, sequential edge-focusing program and parallelize it in different ways on various message-passing architectures—was motivated by two problems in improving the computational efficiency of edge detection processing. First, edge detection is computationally intensive. The computation is conducted pixel by pixel, with several dozen arithmetic operations for each pixel. For example, edge detection using the edge-focusing technique requires multiple processing iterations from coarse to fine levels of resolution. When we ran a small edge detection problem (the image shown in Figure 1) on various powerful workstations, even the fastest—a 40-MHz Intel iPSC/860—required 221 seconds for the 10-iteration process.

The second major problem is edge detection accuracy. Bergholm's edge-focusing method[4] combines high positional accuracy with good noise reduction, so it forms a good foundation for developing more efficient edge detection algorithms.

Potentially, using direct and iterative methods in parallel can solve many image-processing problems. The direct method is a divide-and-conquer approach, partitioning a problem into smaller parts and combining individual solutions into a solution of the whole. The iterative method is applied after partitioning has distributed the data among processors. During each iteration, each processor independently updates its image data using a numerical method; this computational phase requires no interprocessor data communication. At the end of each iteration, each processor acquires updated image data from the other processors until a solution tolerance is satisfied. A synchronization barrier guarantees that the

1070-9924/97/$10.00 © 1997 IEEE

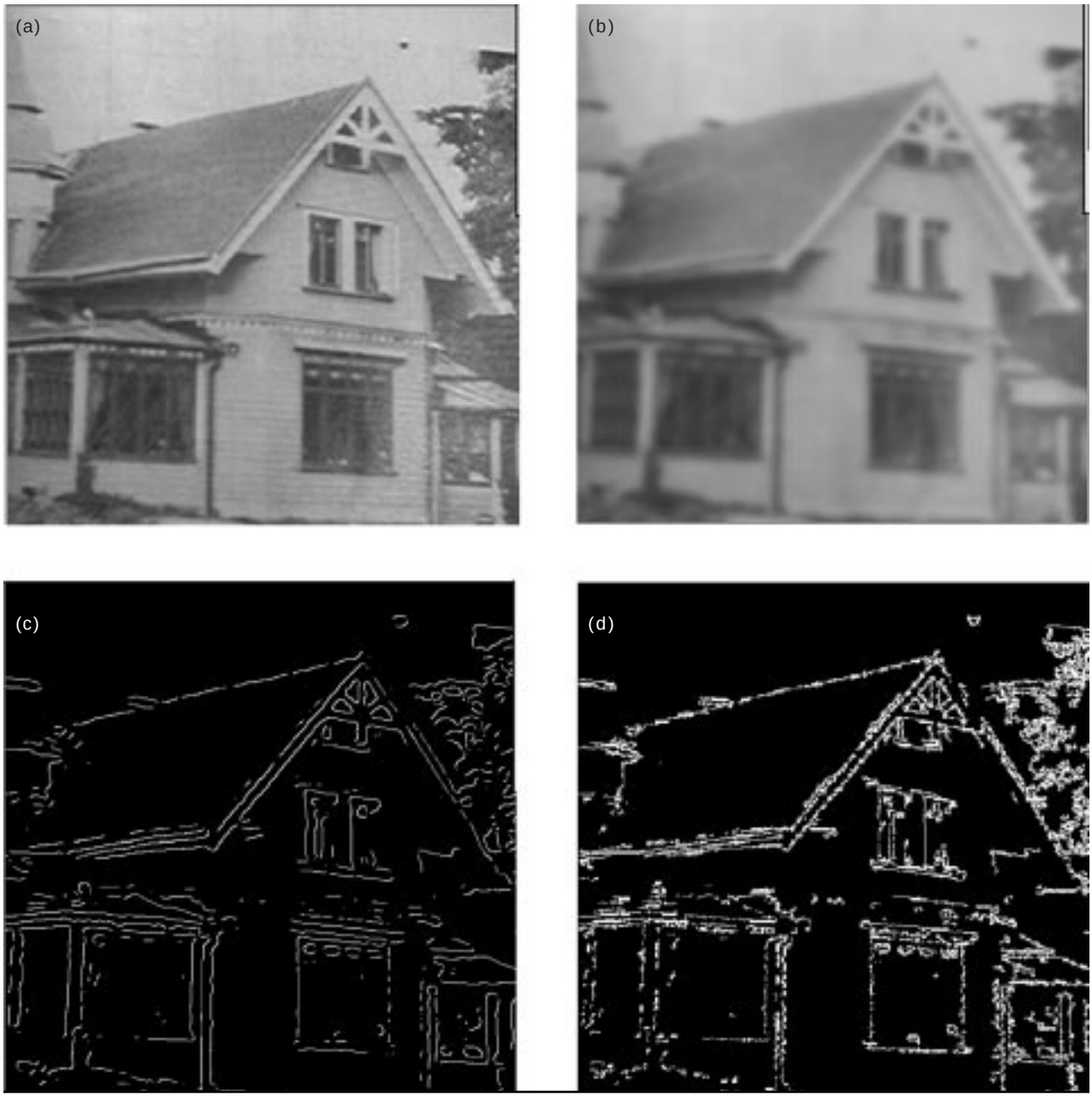

Figure 1. Edge-focusing application: (a) original 512 × 512-pixel image; (b) initial Gaussian blurred image, $\sigma = 3.8$; (c) edge map of initial Gaussian blur; and (d) final edge map.

next iteration cannot start until all processors finish computation and data acquisition. Like other image-processing applications, edge detection is a good candidate for parallel implementation.

Our work has three objectives:

♦ to gain insight into implementation and performance of edge detection algorithms on general-purpose message-passing and data-parallel architectures,

♦ to investigate the effects of network variations on these algorithms, and

♦ to evaluate computing scalabilities on three representative network systems by examining edge detection execution and communication patterns.

Implementation issues

Edge detection shares a characteristic of many image-processing problems: algorithms for solv-

ing them can be decomposed at a high level into fairly large segments that can be calculated concurrently and require relatively little interprocessor communication. In addition, edge detection has a one- or two-dimensional data domain structure, so solving such problems requires large local memory space. Thus, they are well suited for distributed-memory multicomputers and data-parallel architectures.

An important issue for parallel implementation on multicomputers is the choice between message-passing and data-parallel languages. Most message-passing languages are standard sequential languages such as Fortran or C extended with a library of interprocessor communication functions. In message-passing languages the application programmer must explicitly include statements to perform data distribution and processor synchronization. In contrast, data-parallel compilers automatically insert processor synchronization barriers before communication commands, and data-parallel runtime systems automatically distribute data across processors. Relinquishing control to the compiler and runtime system relieves the programmer of tedious, error-prone details but can result in less efficient code. Data-parallel languages are best suited for applications that perform the same computations on large amounts of data, as in low-level image-processing functions.

We have included in our study both message-passing and data-parallel versions of the edge-focusing application to compare their ease of program development and performance. We implemented the edge detection algorithms on three representative message-passing architectures: a low-cost, heterogeneous PVM (Parallel Virtual Machine) network, an Intel iPSC/860 hypercube, and a CM-5 massively parallel multicomputer. The CM-5 studies included both message-passing and data-parallel versions.

Edge focusing

Edge detection algorithms rely on derivative operators that produce large values at pixels where intensity or some other characteristic is changing rapidly. Although any derivative operator can be used in edge detection, the most popular operators are digital approximations to the gradient, such as the Roberts operator, and digital approximations of the second derivative, such as the Laplacian operator.[5,6] The Laplacian operator has general applicability because it is orientation-insensitive, so we have chosen it for our work.

As we have said, edge detection methods locate pixels at points where there is an abrupt change in some measured image value, typically image intensity. If we search for edges by considering intensity changes between adjacent pixels or within a small local region, the inherent random noise and digitization effects in the image will produce artificial edges that do not correspond to true structure boundaries. On the other hand, if we identify edges as transitions between larger, averaged regions, the determined edge locations lose precision because the averaging process lowers image resolution. Thus, the basic conflict in edge detection is ignoring spurious edges caused by random signal noise without distorting the shape of true edges.

One method of identifying essential image structures is to strongly blur the original image to filter away noise and unnecessary detail, thus producing a coarse-resolution edge map. Then, continuously increasing the resolution gradually focuses the edges. The edge-focusing algorithm consists of the following steps (see Figure 1):

1. Blurring—We use a 2D Gaussian average operator to blur the original image. Blurring eliminates high-frequency noise by averaging, or smoothing, pixel intensities. Using more pixels to obtain an average intensity produces a lower-resolution, more strongly blurred image, but one with less random noise.
2. Registering—Since object edges appear as discontinuities in intensity, the edge detection program locates them by searching for pixels with the maximum changes in intensity. Then it registers their locations in computer memory.
3. Matching—Edge matching compares the edge map of the previous iteration with the map of the current iteration. Because the previous map was obtained from a lower-resolution (more blurred) image, it has less noise. However, the current map is obtained from a higher-resolution image and consequently has more precise edge locations. The matching process accepts a higher-resolution edge only if its location is within a certain distance from a lower-resolution edge. In our algorithm, this distance is one pixel.
4. Repeating or halting—If the blurring scale at the current step is sufficiently large, the edge-focusing process continues by reducing the blurring scale and beginning another iteration. Otherwise, the final edge map is formed and the process terminates. When the blurring scale becomes small enough, the blurring operator cannot produce any significant blurring and the edge locations are precise.

Numerical methods

An edge map, denoted $E(i, j, \sigma)$, is a binary image represented by 0's and 1's with a resolution parameter σ. A point $E(i, j, \sigma) = 1$ if the pixel (i, j) is an edge point; otherwise, $E(i, j, \sigma) = 0$.

For the image-blurring operation in the edge-focusing algorithm, let $f(i, j)$ denote the gray-level image and $g(i, j)$ the blurred image. The blurred image is computed from the discrete convolution shown in Figure 2, where

$$\text{Gauss}(i, j, \sigma) = e^{-(i^2+j^2)/2\sigma^2}$$

is the Gaussian operator and w is the size of the convolution window.

Figure 3 shows the Gaussian function we used for our initial blurring of the image in Figure 1a, where $\sigma = 3.8$. This operator has good properties in both the time and frequency domains and does not produce false edges in the blurring process.

We compute a digital form of the convolution shown in Figure 2 using a convolution window—a 2D matrix consisting of a Gaussian weight in each entry of the window, denoted $W(i, j)$. The weights are calculated by the Gaussian operator. The size of a Gaussian window, w, is determined by the blurring scale $w = \lceil 8\sigma \rceil$.

In the next step, the edge-focusing algorithm finds the edge points in the blurred image. An edge contour, denoted $c(i, j)$ in image $g(i, j)$, is defined as a set of points where $g(i, j)$ has the maximum gradient magnitude measured in the direction of the gradient. We define this gradient

$$\nabla [g(i, j)] = g_i x + g_j y, \tag{1}$$

where x and y are unit vectors in the directions of i and j respectively, $g_i = \partial g(i, j)/\partial i$, and $g_j = \partial g(i, j)/\partial j$. The points on the edge contour $c(i, j)$ satisfy

$$g_i^2 g_{ii} + g_j^2 g_{jj} + 2 g_i g_j g_{ij} = 0, \tag{2}$$

where $g_{ii} = \partial^2 g_i/\partial i^2$, $d_{jj} = \partial^2 g_j/\partial j^2$, and $g_{ij} = \partial^2 g(i, j)/\partial j \partial j$. These derivative operators can be obtained in finite difference form:

$$\begin{aligned} g_i = {} & [g(i+1, j-1) + 2g(i+1, j) + g(i+1, j+1)] \\ & - [g(i-1, j-1) + 2g(i-1, j) + g(i-1, j+1)] \end{aligned} \tag{3}$$

and

$$\begin{aligned} g_j = {} & [g(i-1, j+1) + 2g(i, j+1) + g(i+1, j+1)] \\ & - [g(i-1, j-1) + 2g(i, j-1) + g(i+1, j-1)]. \end{aligned} \tag{4}$$

$$g(i, j) = \frac{1}{2\pi\sigma^2} \sum_{x=\left\lceil \frac{-w}{2} \right\rceil}^{\left\lfloor \frac{w}{2} \right\rfloor} \sum_{y=\left\lceil \frac{-w}{2} \right\rceil}^{\left\lfloor \frac{w}{2} \right\rfloor} \text{Gauss}(i-x, j-y, \sigma) f(x, y)$$

Figure 2. This discrete convolution is used to compute the blurred image.

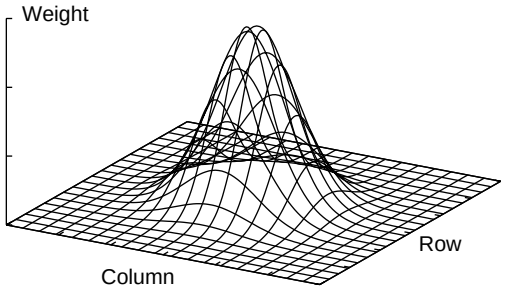

Figure 3. This Gaussian function is used for image blurring ($\sigma = 3.8$).

In image-processing applications, the magnitude of the gradient in Equation 1, denoted $h(i, j)$, is commonly approximated from g_i and g_j by

$$h(i, j) = [g_i^2 + g_j^2]^{1/2} \cong \| g_i \| + \| g_j \| . \tag{5}$$

Therefore, the maximum gradient magnitude measured in the gradient direction is

$$\begin{aligned} & \max_{i,j}(\nabla[g_i^2 + g_j^2]^{1/2}) = \max_{i,j}(\nabla[\| g_i \| + \| g_j \|]) \\ & = \max_{i,j}(\nabla[h(i, j)]). \end{aligned} \tag{6}$$

We can now outline Bergholm's edge-focusing algorithm as follows:

1. Create an initial coarse-level edge image $E(i, j, \sigma_0)$ using an edge detector based on the discrete Gaussian blurring defined in Figure 2 and registration of the maximum gradient in Equation 6.
2. For i=1, 2, …, create the ith-level edge map $E(i, j, \sigma_i)$ by using the same blurring technique, where $\sigma_i = \sigma_{i-1} - \Delta\sigma$. The resolution parameter s is decreased by $\Delta\sigma$ at each blurring step. Unlike step 1, the Gaussian operator blurs only the edge points ($E(i, j, \sigma_{i-1}) = 1$) detected in the previous iteration, and their neighbor areas. By matching the new edge points at the ith and i–1th iterations, we form the new edge map $E(i, j, \sigma_i)$.
3. Continue edge focusing until the blurring scale is too small to blur the image.

In our experiments we blurred the image by convolving it with a 2D, symmetric Gaussian function. We scaled the blurring by varying the parameter σ in the Gaussian function. Informally, σ corresponds to the width or spread of the Gaussian function. Larger σ values result in a wider function, causing a larger blur and lower resolution. Reducing σ narrows the Gaussian and reduces blur. We began the edge-focusing process with σ_0 at 3.8 and reduced σ by 0.5 each iteration for a total of eight iterations. Because the convolution window's size is $\lceil 8\sigma \rceil$, this corresponded to an initial convolution window size of 31×31 pixels and a final window size of 3×3 pixels.

Bergholm's algorithm has four notable features. First, the Gaussian convolution removes noise and unnecessary detail at the expense of smoothing sharp edges. This convolution is an expensive computing process involving a large number of arithmetic operations. Second, the iterative focusing process of decreasing σ at each blurring step makes the edge points sharper and clearer. Third, the two major operations of blurring the image and registering the edge points by calculating the maximum gradient magnitude are conducted pixel by pixel, so the entire process is very time-consuming. Finally, the detection operations applied to each pixel depend only on the original image and the edge points detected in the previous iteration. The operations are independent of the edge points at the same iteration. This feature is the foundation of our parallel implementations of the focusing algorithm.

The first issue in implementing parallel edge focusing is how to partition an image data file.

Parallel edge-focusing algorithms

The first issue we addressed in implementing parallel edge focusing is how to partition an image data file. Partitions can be square blocks, rectangular blocks, row blocks, or column blocks. For our message-passing implementation, we partitioned the image into row blocks. For our data-parallel implementation, we used the default block partitioning assigned by the runtime system. Although edge-focusing operations are independent of the image points at each blurring iteration, there are some dynamic dependencies among these points. Therefore, a straightforward partitioning of the image data does not always work well, mainly because the edge points may move after each blurring iteration. The edge locations at a coarse level may be quite different from the exact edge locations at the final level of resolution. During focusing, edge points gradually move toward the exact edge locations. Bergholm proved that if the blurring parameter σ is reduced slowly enough ($\Delta\sigma \leq 0.5$ for each iteration), the edge locations move no more than one pixel. Therefore, in addition to setting $\Delta\sigma \leq 0.5$ in an image partition, we also need to enlarge the subregions' boundaries to at least one pixel wide to avoid missing edge points that move out of the subregion.

Another important issue in message-passing algorithms is processor scheduling. Our basic computation structure for the message-passing implementation on a distributed-memory multicomputer is the master-and-slave model: the master node initiates, controls, and schedules the computing processes in slave nodes. Scheduling affects how well the computational load is balanced across the processors. Better load balancing improves performance. The parallel edge-focusing algorithm can be distributed and scheduled in three ways: uniform data partitioning, static scheduling, and dynamic scheduling.

Uniform data partitioning. Uniform partitioning is a data decomposition method that equally divides the image data file into multiple subregions and assigns subregions to each processor at load time. This method performs well if equal computation is required for each data region. The parallel edge-focusing algorithm using uniform data partitioning proceeds as follows:

1. The master node partitions the image into equal subregions.
2. Edges are focused iteratively as follows:
   a. The master assigns one or more image subregions to each slave.
   b. Slave nodes perform Gaussian blurring and edge registering in their assigned subregions.
   c. The master collects edge image subregions from slave nodes.
   d. The master decides whether resolution is sufficiently high. If so, edge focusing is complete. Otherwise, the master sends adjusted image subregions to each processor, and iteration continues at step 2a.
3. The master outputs the final detected-edge image.

Static scheduling. In practice, uniform image partitioning may result in unequal computation load distribution and performance degradation. Static scheduling attempts to equally partition the task rather than the image to produce a computation load that is balanced during execution. In edge detection, the number of edge points in the initial image determines task sizes. The master node partitions the image into subregions, each with roughly an equal number of edge points but not necessarily an equal number of image points, and assigns these subregions to the slave nodes.

Dynamic scheduling. Although static scheduling balances the computation load when work is distributed to the slaves, the number of edge points in each subregion may change during edge focusing. Therefore, the computation load may become unbalanced during execution. Dynamic scheduling divides the image into multiple subregions with approximately equal numbers of edge pixels. The number of subregions is significantly larger than the number of processors. These subregions are put into a task queue. During execution, a task scheduler dynamically assigns tasks in the queue to available processors. When a processor finishes its current task, it looks for the next task. This scheme produces better load balancing, but the runtime scheduling adds overhead.

Results. We implemented uniform, static, and dynamic scheduling on the iPSC/860 to observe their effects on the edge-focusing algorithm for three different images. In these experiments, uniform data partitioning gave the best performance and dynamic scheduling the worst. Although the test images appeared substantially different, the overall distribution of edge locations was uniform enough that static- and dynamic-scheduling overheads outweighed the savings from balancing the computation. Consequently, we implemented our message-passing algorithm with uniform data partitioning.

Message-passing implementation

The message-passing implementation requires several global operations, including reduction and searching for maximum and minimum values. The CM-5 has a separate control network for global operations; global and node-to-node message-passing operations share the same data communication links on the iPSC/860. In addition to global operations, we addressed three other implementation issues for the message-passing versions.

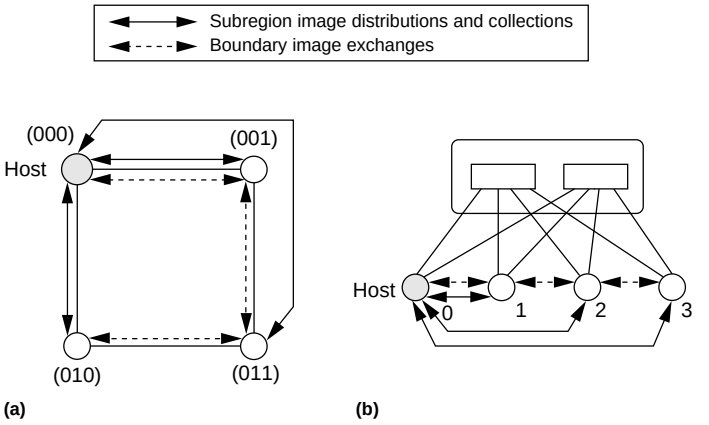

Figure 4. Communication arrangements of the edge-focusing algorithm on (a) the iPSC/860 four-node hypercube and (b) the CM-5 four-node fat-tree network.

Distributed or centralized window weight calculation. Each iteration requires calculation of convolution window weights. Either the master must calculate and broadcast the weights, or each slave program must calculate the weights locally. We performed local calculations to reduce communication cost.

Communication modes between master and slaves. Both systems support blocking and nonblocking communication protocols. The master program uses nonblocking protocols to distribute the image subregions among the processors, allowing communications and computation in the master to be overlapped. Slave programs use blocking protocols to send back intermediate processing results to the master.

Pixel exchanges between boundary regions. At the end of each iteration, updated pixels in boundary regions must be exchanged. For such exchanges, we arranged direct-link neighbor communications on the iPSC/860 and same-switch-level communications on the CM-5. In addition, we designed the subregion image distributions and collections to be conducted through optimal paths. Figure 4 shows examples of communication patterns in subregion image distributions and collections and boundary image exchanges on a four-node hypercube and a four-node CM-5 fat-tree network (which we describe later).

Data-parallel implementation

Data-parallel languages allow application programmers to define a matrix of virtual processors. For example, an image-processing application would assign a virtual processor to each pixel

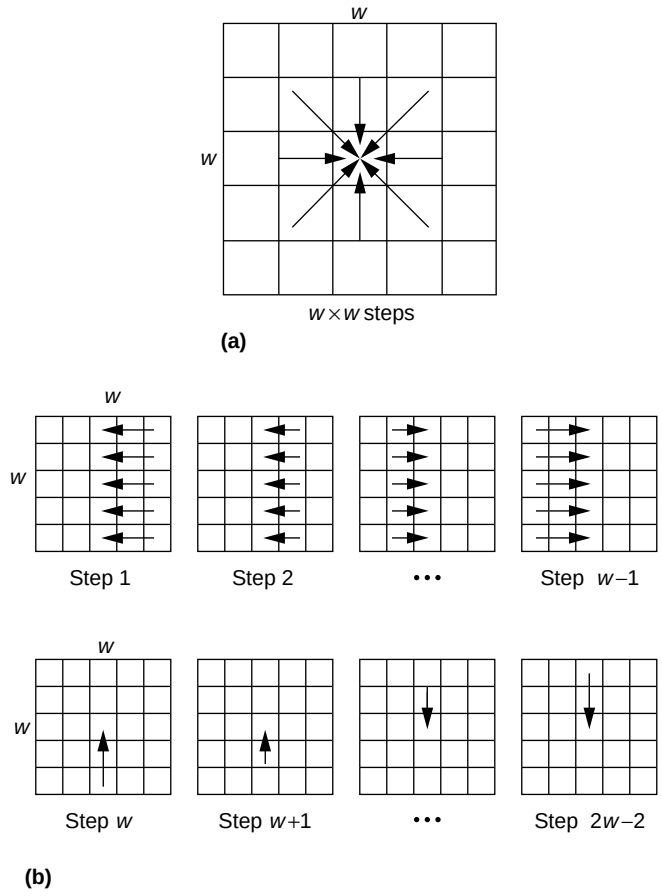

Figure 5. Virtual communication in (a) naive and (b) folded data-parallel convolution algorithms.

Table 1. Execution times for the edge detection of the house image on a Sun Sparcstation PVM network.

No. of nodes	Node types	Time (seconds)
1	Sparc ELC	2,145
2	Sparc ELCs	1,222
4	Sparc ELCs	595
8	4 Sparc ELCs and 4 Sparc 1+'s	578
16	5 Sparc ELCs, 4 Sparc 1+'s, 5 Sparc SLCs, 1 Sparc 10/30,1 Sparc 10/40	482

ELC: enhanced low cost; SLC: super low cost

location; that is, for a 512 × 512–pixel image, the program would define a matrix of 512 × 512 virtual processors. The runtime system automatically maps virtual processors onto physical processors and converts virtual-processor communication statements into either local memory references or remote message-passing calls. An application program need not specify nor even be aware of the number of physical processors and data layout. The runtime system handles communication details, including source and destination addresses, freeing the programmer to focus on a natural representation of the problem. A second important feature of data-parallel languages is the automatic synchronization provided by the compiler prior to communication. This eliminates race conditions that often cause major debugging problems in message-passing languages.

For the CM-5 multicomputer, there is a third important difference between data-parallel and message-passing languages. CM-5 processor nodes contain a standard RISC (reduced instruction set computing) processor and four vector units. However, compilers for message-passing languages on the CM-5 produce only Sparc RISC code and do not use the vector units. A programmer can write functions in vector assembler code and call these from a message-passing program, but this substantially increases development time and makes the code nonportable. In contrast, data-parallel language compilers on the CM-5 produce vector code, effectively increasing the number of processors and memory bandwidth.

Because data-parallel layout and communication are so easily molded into the application domain, data-parallel languages typically have shorter learning curves, faster development times, and less debugging than message-passing languages. A consequence of this automation and data abstraction is system dependency; a data-parallel program's performance heavily depends on the compiler and runtime system. By relying on system software in the critical area of data layout and processor communication, an inexperienced programmer can trade performance for ease of use. Thus, data-parallel languages are most successful when identical operations are performed on large amounts of data, and when the mapping of virtual to physical processors matches the application's communication pattern. Both these conditions hold in low-level image-processing applications.

Our data-parallel version of edge focusing uses the CM-5 C* language with default block data mapping. We implemented a folded convolution algorithm[7] that reduces the communication costs of a naive algorithm. The naive algorithm directly implements the convolution formula without considering execution efficiency (see Figure 5). The folded method packs convolution results for one dimension (rows, for example) before

communicating along the second dimension (columns). This algorithm does not change the amount of data transferred, but it does reduce the number of messages from $\Theta(w^2)$ to $\Theta(w)$, where Θ represents the lower bound, for a $w \times w$ convolution window. Because message start-up costs are expensive, reducing the number of messages significantly lowers communication costs.

A PVM network is an inexpensive solution for edge detection because it uses existing workstation resources.

PVM performance results

We first implemented the message-passing algorithm in PVM[8] and performed experiments across a network of workstations. This is an inexpensive solution for edge detection because it uses existing workstation resources. Table 1 reports the execution times of processing the house image (Figure 1) on Sun Sparcstation PVM networks of various sizes. These results show that a homogeneous multicomputing environment with a small number of nodes is highly effective for the edge detection program. However, unbalanced processor power in a heterogeneous environment degrades computing performance significantly. By first implementing the algorithm across a PVM network, we reduced costly development time on large-scale parallel systems while still producing efficient distributed algorithms. In addition, a cluster of workstations is a useful and practical environment for medium-size edge detection and for other image-processing applications that are too intensive for a single workstation but do not justify the use of a large-scale parallel system.

iPSC/860 and CM-5 performance results

The Intel iPSC/860 hypercube[9] is a distributed-memory, MIMD (multiple-instruction, multiple-data) multicomputer with up to 128 nodes. Each node contains the i860 40-MHz processor, 8 Mbytes of memory, an 8-Kbyte data cache, and a 4-Kbyte instruction cache. The data link bandwidth is 2.8 Mbytes per second. The processor speed is extremely fast compared to the network bandwidth.

The CM-5,[10] also a distributed-memory, MIMD multicomputer, can be configured with up to 16K (16,384) processing nodes. The CM-5 used in our work was configured with 1K (1,024) nodes. CM-5 processor nodes contain a 33-MHz Sparc RISC processor with a 64-Kbyte combined data and instruction cache, a 32-Mbyte dynamic RAM, and an interface to control and data interconnection networks. Nodes can be augmented with four vector units, each of which has a high-speed path to its associated memory bank. Two separate networks connect the nodes: a data network for concurrent message transfers between pairs of processors, and a control network for low-latency broadcasts and processor synchronization. In the absence of contention, bandwidth between nodes on the data network is 40 Mbytes per second.

Message-passing results

A major problem with sequential image convolution is its heavy demand for memory access.[11] When image and window data sizes exceed cache size, execution time increases due to cache misses. Distributed-memory multicomputers can reduce the effect of this problem because each processor has its own data cache; thus, aggregate cache size and bandwidth increase as processors are added. This increase in cache size significantly improves the edge-focusing algorithm's parallel-computing performance.

As the number of processors increases, the number of pixels in an image subregion assigned to a processor decreases by the following formula: $S_{image}(p) = (N \times M)/p$, where N and M are the numbers of columns and rows respectively of the image data, and p is the number of processors. For single-precision computations, each pixel is represented by 4 bytes in the input data and 4 bytes in the output data. With four processors and a 512×512–pixel image, each processor's memory requirement exceeds 500 Kbytes. With 64 processors, the memory requirement diminishes to 32 Kbytes. Recall that iPSC/860 processor nodes have an 8-Kbyte data cache, and CM-5 processor nodes have a 64-Kbyte combined cache. The CM-5's larger cache size proves effective in the edge-focusing application. Figure 6 (next page) compares computation and communication times on the iPSC/860 and the CM-5 for the Figure 1 image. The CM-5 outperforms the iPSC/860 despite the fact that the CM-5 Sparc processor is slower than the iPSC/860 processor.

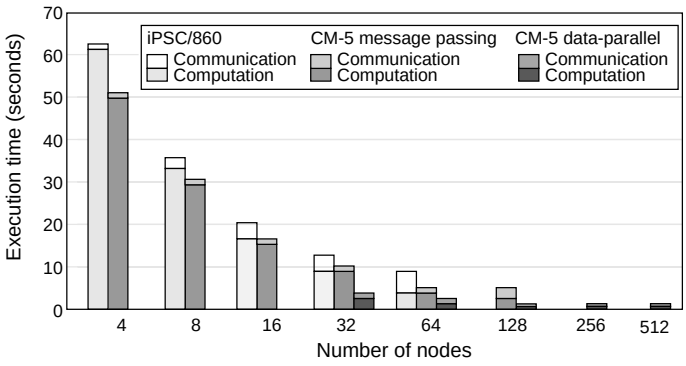

Figure 6. Performance results of the edge-focusing algorithm on the iPSC/860 and the CM-5.

Table 2. Execution and communication times for message-passing (MP) and data-parallel (DP) versions of the edge-focusing algorithm. Data-parallel communication time is reported for virtual communication, which includes both on-node memory references and off-node communication.

Number of nodes	4	8	16	32	64	128	256	512
Total execution time (seconds)								
iPSC/860 (MP)	61	34	20	12	9			
CM-5 (MP)	50	30	16	10	5	5		
CM-5 (DP)				3.3	1.9	1.1	0.7	0.6
Computation time (seconds)								
iPSC/860 (MP)	60	32	18	10	6			
CM-5 (MP)	49	29	15	9	4	2		
CM-5 (DP)				1.9	1	0.5	0.2	0.2
Communication time (seconds)								
iPSC/860 (MP)	1	1.5	2	2.4	3.3			
CM-5 (MP)	0.9	0.8	0.8	0.9	0.9	2.8		
CM-5 (DP)				1.4	0.9	0.6	0.5	0.4

Table 3. Overall network performance of the iPSC/860 and the CM-5.

Communication parameters	α (μs)	β(μs/bytes)	γ (μs)	α / γ	β / γ
iPSC/860	470	0.357	0.025	18,800	14.28
CM-5	64	0.025	0.030	2,133.33	1.20

Communication between nodes is also critical to performance. Figure 6 and Table 2 show that as the systems scale up to 64 processors, communication cost for this application proportionally increases on the iPSC/860 but stays almost constant on the CM-5. However, at 128 processors on the CM-5, communication overhead increases to a point exceeding computation time. This occurs because the problem size is not large enough for such a large system partition. In addition, communications for subregion image distributions and collections are performed across switches in CM-5. When the number of nodes is sufficiently large, communication latency significantly increases for a fixed-size problem.

CM-5 data-parallel results

Not only was the data-parallel version easier to develop and debug, it produced shorter code and, as shown in Figure 6 and Table 2, executed much faster than the corresponding message-passing implementation on the CM-5. However, these results should be viewed with caution: the main reason for the data-parallel version's better performance is that the C* data-parallel compiler used vector units, whereas the message-passing version did not.

As the number of CM-5 nodes increased from 32 to 512, the computation time scaled almost linearly. Our reported computation times include parallel computation plus execution time for nonparallelizable (sequential) code segments and parallel overhead. This overhead is incurred whenever a parallel code block is initialized and distributed to the nodes, and upon vector code start-up and loop control. The almost linear scaling of computation times up to 512 processors indicates the Gaussian convolution contains predominantly parallelizable code in the data-parallel implementation, with relatively low overhead costs.

Our reported communication times for the data-parallel version are measurements of virtual communication, which includes on-node memory access as well as off-node message passing. Virtual-communication times dropped as processor nodes increased from 32 to 512, although the scaling is less linear than that observed for computation time. This decrease in virtual-communication time occurs because fewer virtual processors map to each node as nodes are added. That is, each node is responsible for fewer pixels, resulting in fewer memory accesses and less message data per node. Because memory accesses occur concurrently on all nodes, reducing the number of accesses per node reduces overall virtual-communication time. Likewise, node-to-node communication occurs concurrently for independent node pairs, so reducing the amount of message data per node reduces communication time. In image-processing applications, the communication pattern is very regular, which means node-to-node communication is highly concur-

rent. Compared to computation, communication times do not scale as well because of the difference in overheads: start-up cost on the CM-5 for message passing is much greater than for initiating a parallel code block.

In general, the data-parallel nature of low-level image-processing applications matches the vector-architecture and data-parallel programming models. As a result, for these applications we can benefit from the higher level of abstraction offered by data-parallel languages, without giving up performance.

Performance implications of network architectures

Network latency forms a major obstacle to improving parallel-computing performance and scalability on multicomputers. Therefore, we studied and compared latency-changing patterns of the iPSC/860 and CM-5 network systems from the architectural point of view.

An important interprocessor communication measure is network latency in transferring K bytes between neighboring node processors:

$$T_{lat}(K) = \alpha + \beta K$$

where α is the start-up time in μs (microseconds) for sending a packet, and β is the bandwidth of the data link (μs/bytes). If we use the processor node cycle time, denoted as γ(μs), to characterize the system's computational speed, the ratios α/γ and β/γ give the communication efficiency of a network system.

Table 3 compares the overall network performance of the iPSC/860 and the CM-5. Our distributed edge-focusing experiments demonstrated the difference between the two systems' communication efficiencies. The CM-5 is significantly better-balanced in terms of the computation and communication ratio. For example, the iPSC/860's β/γ is about 12 times higher than the CM-5's.

The bandwidths of communication network channels change differently as the number of processors changes in multicomputer networks. For example, we can make a hypercube of arbitrary dimension using a linear arrangement with connecting wires. We obtain the hypercube of each dimension by replicating the hypercube of the next-lower dimension and then connecting corresponding nodes. We construct a higher-dimension hypercube by further connecting the bisections of the current-dimension hypercube. A channel is a physical link between two directly connected nodes. It consists of a bundle of wires for data bits and any necessary control bits. The number of channels across the bisection needed to construct a hypercube is $N/2$, where N is the number of nodes in the hypercube. Therefore, the bandwidth of each channel proportionally decreases as the number of processors increases. This example shows that bisection bandwidth is an important limitation in a multicomputer system.[12]

The CM-5 data network is a modified 4D fat tree. A fat-tree network is a treelike structure in which bandwidth increases at each level nearer the root. This increasing bandwidth counteracts contention effects on a loaded network. As a result, each level has a guaranteed bandwidth, and the network is scalable. In the CM-5 data network, bandwidth doubles between the first and second and the second and third levels and then remains constant.

The network consists of router chips, each with an 8-bit-wide bidirectional link to each of its four child chips lower in the fat tree, and four 8-bit-wide bidirectional links to its parent chips higher in the fat tree. To route a message from one processor to another, the network sends the message up the tree to the least common ancestor of the two processors, and then down to the destination. The network design provides several comparable paths for a message to take from a source processor to a destination processor. As it goes up the tree, a message may have several choices as to which parent connection to take. In addition. the CM-5 network has a constant bisection bandwidth. Thus, it should have lower latency and higher throughput than variant network bisection structures such as high-dimensional hypercubes. This architectural advantage improves the efficiency of large-scale image processing.

Our message-passing models of the edge-focusing algorithm turned out to be highly effective and scalable. In addition, we found that a data-parallel implementation can outperform a message-passing implementation on the CM-5, primarily due to the data-parallel compiler's automatic use of vector units. The data-parallel version was easier to develop than the message-passing version because the runtime system handles data distribution and communication details. Despite this automation, the data-parallel communication costs remained low compared to the message-passing version.

In terms of network architectures, network latency increased proportionally as the PVM cluster and the iPSC/860 architecture were scaled; on the other hand, the CM-5's network latency decreased as that system was scaled. Our study also indicates that the choice of network architectures directly affects computation scalability. A major design and performance evaluation issue for a scalable multicomputer system is the network latency limit and its changing patterns in a computation as the system is scaled. Our current work includes designing and testing additional communication variations for the message-passing and data-parallel models. ♦

Acknowledgments

Y. Zhou provided the sequential edge-focusing program for this project. We thank N. Wagner for carefully reading this article and contributing many valuable suggestions. This work was supported in part by National Science Foundation grants CCR-9102854 and CCR-9400719, by the US Air Force under agreement FD-204092-64157, by Air Force Office of Scientific Research grant AFOSR-95-1-0215, and by a grant from Cray Research. The parallel experiments were performed on Intel iPSC/860 machines at the Rice University Center for Research on Parallel Computation and at the Intel Supercomputer Systems Division in Portland, Oregon, and on CM-5 machines at Los Alamos National Laboratory and at the University of Illinois National Center for Supercomputing Applications.

References

1. J.F. Canny, "A Computational Approach to Edge Detection," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Nov. 1986, pp. 679–698.
2. T. Pavlidis and Y. Liow, "Integrating Region Growing and Edge Detection," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Mar. 1990, pp. 225–233.
3. P. Perona and J. Malik, "Scale-Space and Edge Detection Using Anisotropic Diffusion," *IEEE Trans. Pattern Analysis and Machine Intelligence*, July 1990, pp. 629–639.
4. F. Bergholm, "Edge Focusing," *IEEE Trans. Pattern Analysis and Machine Intelligence*, June 1987, pp. 726–741.
5. A. Rosenfeld and A.C. Kak, *Digital Picture Processing*, 2nd ed., Academic Press, New York, 1982.
6. R.C. Gonzalez and R.E. Woods, *Digital Image Processing*, Addison-Wesley, Reading, Mass., 1993.
7. S.G. Dykes and X. Zhang, "Folding Spatial Image Filters on the CM-5," *Proc. Ninth Int'l Parallel Processing Symp.*, IEEE Computer Soc. Press, Los Alamitos, Calif., Apr. 1995, pp. 451–456.
8. A. Geist et. al., *PVM3 User's Guide and Reference Manual*, Oak Ridge Nat'l Laboratory, ORNL/TM-12187, Oak Ridge, Tenn., May 1993.
9. *iPSC/860 User's Guide*, Order No. 311532-007, Intel Corp., Portland, Ore., 1991.
10. *The Connection Machine CM-5 Technical Summary*, Thinking Machines Corp., Cambridge, Mass., 1993.
11. S.G. Dykes et al., "Computation and Communication Patterns of Large-Scale Image Convolutions on Parallel Architectures," *Proc. Eighth Int'l Parallel Processing Symp.*, IEEE Computer Soc. Press, Los Alamitos, Calif., Apr. 1994, pp. 926–931.
12. X. Zhang, "System Effects of Interprocessor Communication Latency in Multicomputers," *IEEE Micro*, Vol. 11, No. 2, 1991, pp. 12–15, 52–55.

Xiaodong Zhang is an associate professor of computer science at the University of Texas at San Antonio, where he directs the High-Performance Computing and Software Laboratory. His primary research interests are parallel and distributed computation, computer system performance evaluation, and scientific computing. Zhang received his PhD in computer science from the University of Colorado at Boulder in 1989. He is a senior member of the IEEE, and he chairs the Computer Society's Technical Committee on Supercomputing Applications.

Sandra G. Dykes is a PhD candidate at the University of Texas at San Antonio, studying parallel and distributed computing, and is the author of several publications on parallel image processing. She holds a BS in chemistry from the University of Texas at Austin and MS degrees in chemistry and computer science from the University of Texas at San Antonio. She is a member of the IEEE Computer Society and the ACM.

Hong Deng is a systems analyst at Diversified Technology. His research interests are database development and scientific computing. Previously, he was a research assistant in the High-Performance Computing and Software Lab at the University of Texas at San Antonio. Deng received his BS in computer science from Nanjing University, China, in 1984, and his MS in computer science from the University of Texas at San Antonio in 1994. He is a member of the IEEE Computer Society.

Contact the authors in care of Xiaodong Zhang, High-Performance Computing and Software Laboratory, Division of Computer Science, Univ. of Texas, San Antonio, TX 78249; e-mail, zhang@cs.utsa.edu, http://www.hpcs.utsa.edu.