TRC94-23

December 1, 1994

Static Global Scheduling for Optimal Computer Vision and Image Processing Operations on Distributed-Memory Multiprocessors

Cheolwhan Lee **Yuan-Fang Wang** **Tao Yang**

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{chlee, yfwang, tyang}@cs.ucsb.edu

Abstract

In this paper, we develop a static global scheduling scheme for mapping computer vision and image processing (CVIP) operations on distributed-memory multiprocessors. Unlike most current parallel image processing research which focuses on parallelizing individual processing algorithms on a particular parallel architecture, our scheduler is for optimizing processor assignment and data partition for an entire image processing pipeline.

The scheduler operates on task graphs specified by conventional visual languages such as `Khoros` and `Explorer`. A task graph is assumed to be a linear chain of operations with any number of nested loops. The task chain is first decomposed into simpler subchains; each a linear sequence of tasks without loops. The communication and computation costs of the component tasks in the subchains are determined by a taxonomy of CVIP operations. Data redistribution overheads in between tasks can also be tabulated in advance for many popular data partitioning schemes. The scheduler then employs a shortest path algorithm to optimize the parallel time, taking into consideration possible variation in the task and resource parameters (such as the image size and number of processors used), and both the *intra*-operation and the *inter*-operation computation and communication times. In this paper, we present the scheduling scheme, and provide analyses and experimental results to verify our approach.

Keywords: parallel processing, image processing, computer vision, schedule, mapping

1 Introduction

Computer Vision and Image Processing (CVIP) algorithms possess characteristics which are ideally suited for implementation on a variety of parallel architectures [20, 22]. Many CVIP problems are inherently parallel because, often times, similar or identical operations are applied to all pixels in an image without conditional branching. Program structures, by a large extent, comprise nested loops over image data, and a number of loop parallelization schemes are potentially applicable [25]. Furthermore, CVIP problems can be computationally intensive, and hence, can benefit from the parallel computation support. For example, many robotic servoing tasks using visual feedback [16] require image sequences processed at the video frame rate (30 frames per second). In remote-sensing applications, the enormous size of the image databases (typically in the gigabyte or terabyte range) [24] makes image processing a major limiting factor in digesting and cataloging image data.

So far, parallel image processing research has focused on implementing *individual* algorithms on a *particular* parallel architecture [1, 3, 11, 19]. However, optimizing individual tasks in a processing pipeline comprising many such tasks does not guarantee the optimal performance of the whole operation. For distributed memory architectures, each task may employ different data partition and processor allocation schemes; this necessitates data migration in between operations, and such overheads must be considered. Furthermore, a compile-time scheduler need be aware of possible run-time variation of task and resource parameters (e.g., input image size and number of processors available). This research thus aims at developing a global scheduler–which takes into consideration the data shuffling overheads in between processing stages and the run-time variation of the task and resource parameters–to generate data partition schemes for optimizing an entire operation, instead of individual tasks in the operation.

The scheduler operates on the task graph representations of CVIP algorithms, which can be specified using commercial visual languages such as `Explorer` [5] and `Khoros` [17]. A task graph is assumed to be a linear chain of operations with any number of nested loops. The task chain is first decomposed into a number of subsequences based on the loop structures and the types of component operations in the graph. And each subsequence corresponds to a maximal loop-free linear succession of operations. Each subsequence is then expanded into a scheduling graph, where nodes represent tasks and edges delineate the task execution order and data dependence. Each node in a scheduling graph has a weight corresponding to the communication and computation costs of the task in terms of the particular data partitioning scheme used. And each edge assumes a weight equaling the *inter-operation* communication cost, which is the cost of resolving data location inconsistency in between adjacent tasks.

We develop a taxonomy for CVIP operations which facilitates formulating the computation and communication requirements of many CVIP operations in a multiprocessor environment. Furthermore, we tabulate the data redistribution costs for many popular data partitioning schemes. The scheduler then employs a shortest-path algorithm to find the optimal data partitions and processor schedules by minimizing a cost function with both *intra-operation* and *inter-operation* cost components.

To allow variation in the task parameters (e.g., input image size), an aggregated scheduling graph is generated by summing the costs of nodes and edges in scheduling graphs assuming different values of a task parameter. Another short path search is performed in the aggregated scheduling graph. The resulting schedules give the minimum performance variation among different settings of task parameters and provide satisfactory processor schedules and data partitions over a predefined parameter range.

The design goal of the scheduler is to facilitate the migration from a single-processor paradigm to a multiprocessor one in a *transparent* manner. Though it has long been recognized that CVIP operations are ideally suited for parallel implementation, it is a daunting task for most vision researchers to learn a parallel programming language, and to solve problems such as processor allocation, data partition, and load balancing in order to achieve a reasonable speedup with a parallel implementation. With the proposed scheduler, vision researchers can still use the visual programming environments, such as `Khoros` and `Explorer` which they are accustomed to, to design their algorithms. The scheduler will then analyze the corresponding task graphs automatically and determine suitable processor allocation and data partitioning schemes. Our ultimate goal is to use the scheduling results for generating parallel codes for a processing algorithm automatically by employing a library of existing parallel routines.

Related Research Work There have been a number of projects aimed at developing software environments, including compilers and schedulers [6, 13, 14, 21, 26, 27], for parallel processing. However, automated compilation without taking advantage of specific domain knowledge is still quite difficult. Several researchers presented preliminary designs of schedulers for parallel CVIP operations. For example, [15] discussed static and dynamic design/scheduling strategies for image processing tasks comprising a linear sequence of tasks. Although this research provided a useful design framework for parallel image processing, it was not applicable to message-passing architectures and did not consider the communication overheads for redistributing image data.

DISC [23] developed at Purdue University is a *dynamic* scheduler aimed at generating data partitioning and processor assignment schemes at the run time for operations involving many data-dependent tasks and conditional branches. For such operations the program behaviors are in general hard to predict at the compilation time, and hence DISC is employed at the program run time on a dynamic reconfigurable and repartionable machine (PASM). Compared to DISC, our scheduler is a static one, which generates the optimal schedule if the given task does not possess too many dynamic factors that cannot be determined at the compilation time.

We argue that for CVIP operations, a static scheduler is practical because (1) it is debatable whether typical CVIP operations involve an excessive number of conditional statements. This is witnessed by the fact that some popular visual programming environment for image processing, such as the `Explorer` from SGI, does not even provide an `if-then-else` construct, and (2) a dynamic scheduler is more flexible in handling data dependent operations. However, it requires runtime task migration and/or processor reconfiguration as it is done in DISC [23]. Runtime scheduling thus suffers a high overhead in a message-passing architecture. It is still an open problem how to use runtime scheduling for better load balancing without undue overhead. A static scheduler also enjoys the following advantages: a static scheduler does not incur run-time scheduling overhead, is simpler to design, and is applicable to a large class of image operations.

Library-based parallel system development tools were discussed in [9, 10, 18]. [18] presented a parallel library of CVIP routines, but it did not provide scheduling support for parallel tasks. [9, 10] illustrated a user-friendly interface of a parallel library, which stored several implementations of a single algorithm, each optimized for a different parallel architecture. This system was able to select the most efficient implementation of an algorithm by considering various system parameters. It, however, did not address the issues of optimizing the parallel time of a sequence of tasks. We believe that these systems can benefit from the proposed research as schedule optimization, albeit hard and tedious, should be an integral part of any parallel development system.

The remainder of this paper is organized as follows: Section 2 gives an overview of the proposed scheduling scheme. Section 3 develops a CVIP taxonomy and uses it to establish the computation and communication costs of many CVIP operations. In Section 4, various types of processor assignments and data redistribution costs are studied. In Section 5, the optimal scheduling algorithm and the minimal performance variation scheduling algorithm are described. Section 6 presents some experimental results and, finally, Section 7 contains a concluding remark.

2 Overview of the Scheduling Scheme

We first discuss several important system design parameters. We will assume the target architecture is one where processors are connected in a 2D square mesh, with processors located on an edge of the mesh connected to processors on the opposite edge in a wrapped-around manner. Many commercially available parallel architectures support such a topology (e.g., Cray T3D, nCube, Intel iPSC/860). Each processor has its own local memory, and processors communicate by sending messages among them. Message passing incurs a nonzero cost including the startup time, α, which is the latency in preparing a message and arranging a communication channel, and time for transmitting the message. The message transmission rate is characterized by another constant β, which is the time for transmitting a single word from the source to the destination and is inversely proportional to the channel bandwidth. The cost of sending a message is therefore $\alpha + \beta \times M$, where M is the size of the message. We denote the processor speed as ω, which is the time for a float point multiplication. In our simulation, we used the system parameters (processor speed, communication bandwidth, etc.) from Intel iPSC/860 and Paragon, and from nCube which support mesh connection.

A task may employ one of the following six data partitioning schemes, which are also provided by the High-Performance Fortran [12]: row, column, row-cyclic, column-cyclic, block, and block-cyclic partitions. These data partitioning schemes are defined below assuming that the size of the image is $n \times n$, the number of processors is p, and the p processors are arranged in a 2D mesh (Figure 1):

`Row partition`: the ith row of the image data is allocated to processor $P_{f_{row}(i)}$ where $f_{row}(i)$ is $\lfloor i/(n/p) \rfloor$, $i = 0, \cdots, n-1$.

`Column partition`: the jth column of the image data is allocated to processor $P_{f_{column}(j)}$ where $f_{column}(j)$ is $\lfloor j/(n/p) \rfloor$, $j = 0, \cdots, n-1$.

`Row-cyclic partition`: the ith row of the image data is allocated to processor $P_{f_{row-cyclic}(i)}$ and $f_{row-cyclic}(i)$ is $\lfloor i/b \rfloor \bmod p$, where $i = 0, \cdots, n-1$, and b is a blocking factor dictating the number of consecutive rows to be packed together in a processor. We assume $n/b = k_1 p$, where k_1 is a positive integer (i.e., the number of row blocks is divisible by p). If $k_1 = 1$, this

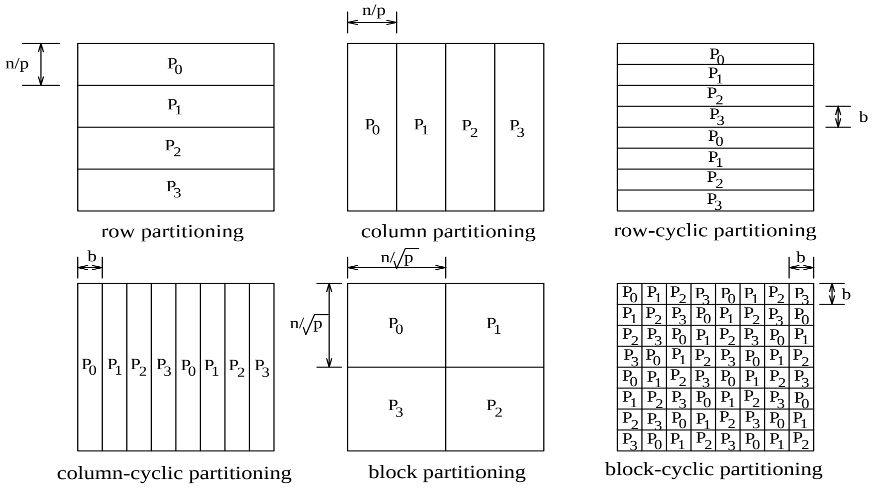


Figure 1: Six types of data partitions of an $n \times n$ image.

partition reduces to the row partition.

`Column-cyclic partition`: the jth column of the image data is allocated to processor $P_{f_{column-cyclic}(j)}$ and $f_{column-cyclic}(j)$ is $\lfloor j/b \rfloor \bmod p$, where $j = 0, \cdots, n-1$, and b is a blocking factor. Again, we assume $n/b = k_2 p$, where k_2 is a positive integer (i.e., the number of column blocks is divisible by p). If $k_2 = 1$, this partition becomes the column partition.

`Block partition`: pixel (i, j) is allocated to processor $P_{f_{block}(i,j)}$ where if $\lfloor j/(n/\sqrt{p}) \rfloor$ is even, then $f_{block}(i,j) = \lfloor j/(n/\sqrt{p}) \rfloor \sqrt{p} + \lfloor i/(n/\sqrt{p}) \rfloor$, and if it is odd, then $f_{block}(i,j) = \lfloor j/(n/\sqrt{p}) \rfloor \sqrt{p} + \lceil j/(n/\sqrt{p}) \rceil - \lfloor i/(n/\sqrt{p}) \rfloor$.

`Block-cyclic partition`: pixel (i, j) is allocated to processor $P_{f_{block-cyclic}(i,j)}$ where $f_{block-cyclic}(i,j)$ is $(\lfloor i/b \rfloor + \lfloor j/b \rfloor) \bmod p$, where b is a blocking factor indicating the number of elements to be packed in a block. We assume that $n/b = k_3 p\sqrt{p}$, where k_3 is a positive integer. This is to ensure that interprocessor communication can be easily established for this partitioning pattern.

Many researchers have proposed parallel versions of various popular CVIP algorithms, assuming different data partitioning schemes and processor architectures (e.g., [9, 10, 13, 14, 15, 18, 21, 22, 23]). The difficulty here, as mentioned before, is that optimizing individual local operations does not guarantee a globally optimal schedule. This is because a CVIP algorithm usually involves a sequence of tasks, and processor schedules and data distributions can change from one task to the next. Furthermore, CVIP algorithms may have loops and data-dependent operations, making it difficult to predict the program behaviors and data distribution patterns at the compilation time. A schedule should also tolerate some variation in the task and resource parameters. For example, a parallel schedule should achieve a reasonable speed up over several input image sizes and numbers of processors used.

In this research, we propose a static scheduling scheme for optimizing the parallel time of CVIP operations involving multiple stages, loops, and data dependent operations. A static global scheduler is a program, employed at the compilation time, that determines how available processors are allocated to different tasks, and how data are partitioned among processors to minimize the overall parallel execution time. Below we give an overview of the scheduling scheme. Details can be found in Sections 3 to 5.

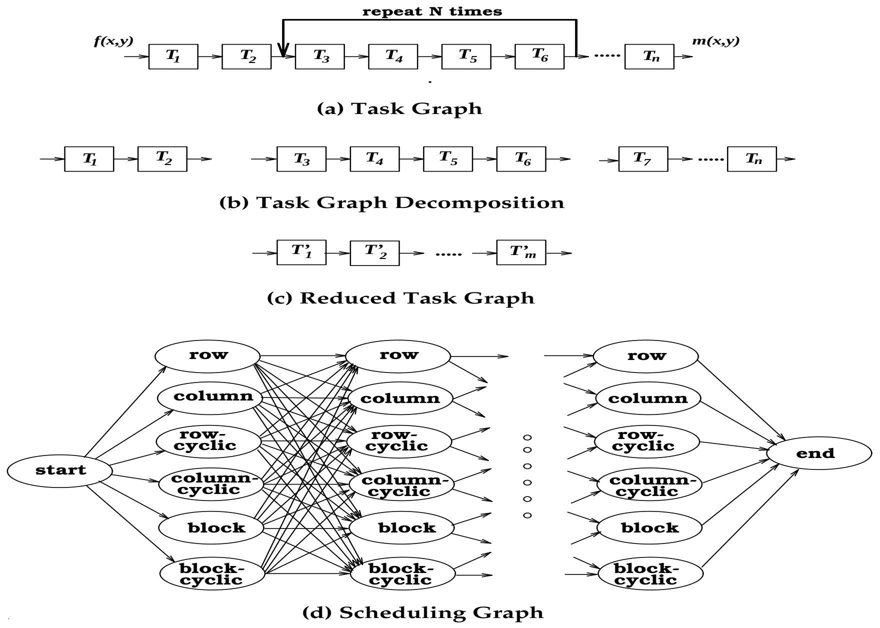


Figure 2: (a) A task graph, (b) the task graph decomposed into three independent task graphs, (c) a reduced task graph, and (d) a scheduling graph.

Input to the scheduler is assumed to be the task graph representation of a CVIP algorithm, which is a linear chain of operations with any number of nested loops. Mathematically speaking, a task chain with loops is a directed graph, $TG = (V_{TG}, E_{TG})$, where V_{TG} represents a set of nodes, each corresponding to an image processing task, and E_{TG} is a set of edges delineating the data dependency of these tasks (e.g., Figure 2.a).

The task graph is first decomposed into subsequences based on the loop structures and the types of component operations in the graph (Figure 2.b). The resulting subsequences are again task graphs, but now corresponding to maximal loop-free/data-independent linear successions of operations.

Task graphs resulting from the previous decomposition may still comprise many stages. To reduce the complexity in finding the optimal schedules for these component task graphs, we construct the corresponding reduced task graphs (Figure 2.c). A reduced task graph, $RTG = (V_{RTG}, E_{RTG})$, is a directed graph where V_{RTG} represents a set of nodes, each corresponding to one or *several* adjacent image processing tasks, and E_{RTG} is a set of edges representing the execution order and data dependency.

Finally, an RTG is expanded into a scheduling graph, $SG = (V_{SG}, E_{SG})$, which is again a directed graph (Figure 2.d). Each stage in the corresponding RTG is expanded into several nodes in an SG, and each node assumes a particular data partioning scheme that can potentially be employed by the task at that particular stage. Furthermore, each node in an SG has a weight that is proportional to the communication time and computation time of the task at that stage, assuming that particular data allocation scheme. Each edge in E_{SG} has an edge weight which represents the data redistribution cost from the previous data partitioning scheme to the ensuing one.

The scheduler operates by minimizing a cost expression involving the computation times (T_{cp}'s) and the communication times (T_{cm}'s) of all the tasks, as well as the data redistribution

	local operators		global operators
	pixel	mask	
data-independent	Arithmetic -multiple operand single operand complex arithmetic nonlinear functions trigonometry bitwise operators comparison, logic Data conversion Format conversion classification	2D convolution	FFT,DCT,Hadmad statistics size operators histogram shape analysis fractal analysis
data-dependent	feature-analysis edge-linking eigenvalues/eigenvectors		

Figure 3: A taxonomy of computer vision and image processing operations.

times (DT's) to resolve data location inconsistency between adjacent tasks in a task graph. Nodes along the shortest path give the data partitioning schemes to be used for each task.

To allow variation in the task and resource parameters such as the input image size and number of processors, schedules with a moderately good performance over a pre-defined range of parameters are needed. This is accomplished by finding the optimal schedules of various settings of a parameter as discussed above, and then constructing a new scheduling graph by aggregating the costs of nodes and edges in scheduling graphs assuming different parameter values. The shortest-path in the new scheduling graph then gives the minimum performance variation schedule.

For efficiency consideration, we develop a taxonomy of CVIP operations, which helps us determine in advance the intra-operation costs of many CVIP operations. Inter-operation data redistribution costs are computed and tabulated in advance as well. Therefore, the scheduling algorithm can perform efficient table lookup procedures to fetch some or all of the T_{cp}'s, T_{cm}'s, and DT's in scheduling. The CVIP taxonomy will be discussed in Section 3 and the data redistribution costs in Section 4. Finally, in Section 5, we will describe the scheduling scheme in more details.

3 Classification of Image Operations

In Figure 3, a taxonomy of several image processing operations in `Khoros` is presented. The taxonomy is based on the data requirement of different image processing algorithms. A *local* operation can be either a pixel operation or a masking operation. A pixel operator operates on a pixel-by-pixel basis while a masking operator effects an area of an image which is usually a small $r \times s$ rectangular tile. If an output pixel value of an operator generally depends upon the values of the whole input image, the operator is classified as a *global operator* [2].

The parallel execution time of a task, PT_i, is the sum of its computation time, $T_{cp}(i)$, and communication time, $T_{cm}(i)$. If a task employs a pixel operator, the computation time for the task is proportional to the size of the input image (n^2), the number of operations applied to each pixel (l), and time of each unit operation (ω). It is inversely proportional to the number of processors used (p), and hence $T_{cp}(i) = n^2 l\omega/p$, assuming that image data have been partitioned in such a way that each processor operates on the same number of image pixels. There is no need for a processor performing pixel operations to communicate with other processors, as all needed information resides in the processor, or $T_{cm}(i) = 0$.

For a masking operator using an $r \times s$ mask (assuming that both r and s are odd numbers), the computation time is proportional to the size of the input image, the size of the mask, and the unit operation time, hence $T_{cp}(i) = n^2 rsl\omega/p$. The communication cost may not be zero however, as pixels needed in a masking operation may not be stored in the same processor. The communication cost depends on the size of the mask, the size of the input image, and the way image data are partitioned. For example, if data are partitioned row-wise with no overlap, a processor will lack some "border" stripes to perform the masking operation for pixels on the border of a partition.

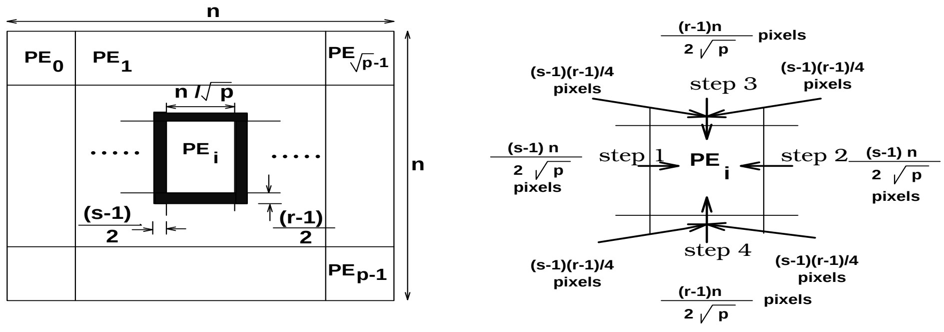
On a 2D mesh, this difficulty can be remedied by a two-phase processor communication: Assume that processors in a 2D mesh are strung together in a linear ring such that processor P_i is connected to processors P_{i-1} and P_{i+1} to its left and right respectively. Note that processor P_0 is connected to processor P_{p-1} in a wrapped-around manner. In a row-wise data distribution, each processor is missing the image data along the top and bottom borders of its data partition, and the size of the missing data is $(r-1)n/2$ along each border. Since each processor keeps n/p rows of image data in its own local memory, the missing data along the top border are found in approximately $\frac{r-1}{2(n/p)}$ left processors, and in that many right processors for data along the bottom border.

In the first phase, each processor packs the border data its right neighbor needs in a message of size n^2/p and sends it off to its right neighboring processor. A processor also receives a message of the same size from it left neighbor. The above step repeats $\frac{r-1}{2(n/p)}$ times. In the second phase, each processor sends a message to its left neighbor and receives a message from its right neighbor $\frac{r-1}{2(n/p)}$ times to fill in the border data on the right side. The two steps replenish the missing borders for all processors, and thus $T_{cm}(i) = 2(\lceil\frac{(r-1)p}{2n}\rceil\alpha + \frac{(r-1)p}{2n}\beta n^2/p) \approx \frac{(r-1)p}{n}(\alpha + \beta n^2/p)$, where α is the communication startup time, and β the transmission speed. If image data are partitioned column-wise, the communication steps are identical with the row-partitioned case except that the number of processors to exchange data with on each side of a processor is $\frac{s-1}{2/(n/p)}$.

With a row cyclic data partition, neighboring pixels are stored in $\lceil(r-1)/(2b)\rceil$ left processors and $\lceil(r-1)/(2b)\rceil$ right processors in the linear ring order, where b is the blocking factor (defined in Section 2). To place data in the processors that need them, messages of size nb are first shifted to the right $\lfloor(r-1)/(2b)\rfloor$ times and picked up by processors along the way. A message of size $(\frac{r-1}{2} \mathtt{~mod~} b)n$, corresponding to pixels on the leftmost edge of a mask, is sent from each processor to its right neighbor $\lceil(r-1)/(2b)\rceil$ distance away. The same number of messages of the same size are then packed and relayed to the left neighbors. Thus $T_{cm}(i) = 2(\lfloor\frac{r-1}{2b}\rfloor(\alpha + \beta bn) + (\alpha + \beta(\frac{r-1}{2} \mathtt{~mod~} b)n)) \approx \frac{r-1}{b}(\alpha + \beta bn)$. Similarly, the communication time for the column cyclic partitioning is $2(\lfloor\frac{s-1}{2b}\rfloor(\alpha+\beta bn)+(\alpha+\beta(\frac{s-1}{2} \mathtt{~mod~} b)n)) \approx \frac{s-1}{b}(\alpha+\beta bn)$.

With a block data partition, pixels needed at a processor may be stored in its neighbors in all directions. Figure 4 shows a case where border pixels are stored in the eight adjacent neighbors. Generally speaking, border pixels used in a masking operation can be found in processors to the west and east within $(s-1)/2$ distance away, and in processors to the north and south within $(r-1)/2$ distance away.

Neighboring pixels are transmitted to the correct processors using four communication



(a) Data allocation of the block partitioning **(b) Four steps neighbor pixels transmission**

Figure 4: Distribution of neighboring pixels in a masking operation using the block data partition where the neighboring pixels may be stored in the eight adjacent processors. The mask size is $r \times s$.

steps. In the first phase, every processor sends a message of size $(s-1)n/(2\sqrt{p})$, which is the chunk of pixel data on its left border needed by its right neighbor, to its right neighboring processor. If the border data are stored in more than one neighboring processor to the left, then every processor sends a message of size n^2/p to its right neighboring processor and has it relayed to processors up to $(s-1)\sqrt{p}/(2n)$ distance away. In the second phase, every processor sends the same size message, which is now the chunk of image data on its right border, to its left neighbor. In the third phase, every processor sends a message of size $(r-1)(n/\sqrt{p}+s-1)/2$, which is the chunk of data on its bottom border, to its neighbor on the south. Again, if data are stored in more than one neighboring processor, then every processor relays a message of size $(n/\sqrt{p}+s-1)n/\sqrt{p}$ to its neighbor on the south up to $(r-1)\sqrt{p}/(2n)$ distance away. In the last phase, every processor sends a message, now data on its top border, to its neighbor on the north. The total communication cost in a block partition is thus $4\alpha + \beta((s-1)n/\sqrt{p} + (r-1)(n/\sqrt{p}+s-1))$, when border pixels are stored only in eight adjacent neighboring processors. If border pixels are spread over more than the eight adjacent neighbors, the communication time is $2\frac{(s-1)\sqrt{p}}{2n}(\alpha + \beta n^2/p) + 2\frac{(r-1)\sqrt{p}}{2n}(\alpha + \beta(n/\sqrt{p}+s-1)n/\sqrt{p})$.

With a block-cyclic data partition, we assume the mask size is smaller than the blocking factor, i.e., $b \geq \mathtt{max}((r-1)/2, (s-1)/2)$. Pixels needed in a masking operation may be found in four adjacent processors on the 1D processor ring, and in the processor itself. Pixels on the top-right and bottom-left corners of the mask are stored in the processor itself. Pixels on the left and top borders of the mask are found in the previous processor, and pixels on the top-left corners are found in the processor two distance backward in the ring. Similarly, pixels on the right and bottom borders are stored in the next processor, while pixels on the bottom-right corner are found in the second processor forward. Neighboring pixels are exchanged in four steps by shifting data two distances forward and backward. The communication cost can be shown to be $4\alpha + 2\beta(\frac{n}{bp})^2(\frac{b(s+r-2)}{2} + \frac{(s-1)(r-1)}{4})$. These costs are summarized in Table 1.

The communication and computation times of global operations can be quite complicated and do not assume any fixed pattern like they do for pixel and masking operations. We found no easy way to tabulate them in advance. Generally speaking, a library of existing parallel codes for global operations must be employed to supply the T_{cp} and T_{cm} used in the analysis.

Operations can also be classified as either *data-independent* or *data-dependent*. In a

Table 1: Communication costs of a masking operation using the six data partitioning schemes. The mask size is assumed to be $r \times s$.

partition	*cost*
row	$(r-1)p(\alpha+\beta n^2/p)/n$
column	$(s-1)p(\alpha+\beta n^2/p)/n$
row-cyclic	$(r-1)(\alpha+\beta bn)/b$
column-cyclic	$(s-1)(\alpha+\beta bn)/b$
block	$\frac{(s-1)\sqrt{p}}{n}(\alpha+\beta n^2/p)+\frac{(r-1)\sqrt{p}}{n}(\alpha+\beta(n/\sqrt{p}+s-1)n/\sqrt{p})$
block-cyclic	$4\alpha+2\beta(\frac{n}{bp})^2(\frac{b(s+r-2)}{2}+\frac{(s-1)(r-1)}{4})$

data-independent operation, complexity of a computation depends only on the size of the input image and processing parameters (e.g., the size of the convolution mask), but not on the content of the input image. The complexity of a data-dependent operation in general depends on the image content. For example, an image feature analysis operation may spend more time on the part of an image cluttered with a lot features than on the part where features are scarce. Our scheduler, to be used at the compilation time, is well suited for tasks involving data-independent operations. Data-dependent operations pose more serious problems, as their behaviors cannot be predicted at the compilation time. The difficulty of mapping data-dependent operations was first recognized in the DISC project [23]. And some worst-case, best-case, or average-case analysis can be employed, but the optimality of such analyses cannot be guaranteed. Under some special conditions, CVIP algorithms involving data dependent operations can still be scheduled at the compilation time, we discuss these cases in Section 5.

We selected the set of frequently-used image processing operators listed in Figure 3 from `Khoros`, and classified them manually based on the above taxonomy. We believe that these operations are representative of typical image processing algorithms and include both local and global operations as well as data-independent and data-dependent ones. Together, they form the test suite of this research.

4 Data Partitioning and Redistribution

Since we assume six data partitioning schemes, there are a total of 36 possible redistribution patterns from one processing stage to the next. If two adjacent processing algorithms assume the same data partitioning scheme, then there should be no redistribution cost incurred. Thus only 30 nontrivial data redistributions assume non-zero costs. We will first illustrate how to compute the redistribution cost using a simple example of row-to-column, row-to-block, and column-to-block redistributions. We will then show that the redistribution cost in general is affected by the size of the image, the number of processors, the particular partitioning schemes used, and hardware communication parameters such as the startup time, α, and the transmission speed, β.

Assume that we have an image processing algorithm T_i using a row partition of image data followed by another algorithm T_j using a column partition, data reshuffling is thus needed in between these two stages. Furthermore, assume a total of four processors are used. Then as depicted in Figure 5 (a), P_0 is allocated one quarter of the image data in a row format during the T_i stage. Before the commencement of the T_j stage, P_0 needs to route portion of the image data stored in its local memory to P_1, P_2, and P_3, and retains only the first quarter of the local data. P_0 will also receive data from P_1, P_2, and P_3 so that one quarter of the image data, now in a column format, is deposited with P_0. Similar send/receive activities are performed by all other processors.

This example illustrates that interprocessor communication depends on the data overlap pattern of two successive data partition schemes. If two processors hold chunks of image data that overlap in two successive processing stages, then the overlapped portion need be exchanged. In a row-to-column redistribution, this overlap is a global phenomenon (i.e., every processor need exchange data with every other processor). However, as illustrated in Figures 5 (d)-(f) which depict a row-to-block redistribution, the overlap is much more local where the four processors can be divided into two groups, with each group communicating within itself and no data are exchanged in between the two groups. For the 30 nontrivial redistributions, our analysis shows that all but four employ a global communication pattern similar to that in the row-to-column redistribution. The two redistributions, row-to-block and block-to-row, assume a localized communication pattern depicted in Figures 5 (d)-(f). Column-to-block and block-to-column redistributions have a communication pattern (Figure 5 (g)-(i)) where data in the ith processor row are exchanged with data in the ith processor column.

The above all-to-all personalized communication pattern for exchanging data among processors has been studied before [14]. However, no specific data distribution was associated with each communication scheme. Our research here presents the detailed cost functions from one data partition to another based on previously developed all-to-all personalized communication for the six popular data allocation schemes.

To realize the all-to-all personalized communication on a 2-D mesh of processors, messages from a processor anywhere in the mesh must be delivered to a processor anywhere else. One well-known communication pattern accomplishes the feast in two separate phases [14]. In the first phase, processors located in the same row communicate with one another by circulating messages (i.e., data) to the left (or right) neighbors. Each processor receives, from processors in the same row, data addressed to itself; it also serves as a temporary repository point for data aimed for its neighbors in the the same column (an accumulation phase). In the second phase, processors in the same column distribute the accumulated data meant for its column neighbors by circulating messages column-wise. This communication scheme guarantees that a message generated anywhere in a 2D mesh can reach any destination in two phases. It is also highly regular, enabling an efficient implementation.

We now derive the communication time formula for the above global two-phase communication scheme used by the majority of 26 redistribution patterns. Each phase of the communication can be treated as a communication on a ring with $\sqrt{p}$ processors. In the first phase, each processor holds n^2/p pixels of image data. Of the original image data, the processor keeps only $1/\sqrt{p}$-th, and sends the rest to the $\sqrt{p}-1$ processors in the same row. The message size is thus $(\sqrt{p}-1)/\sqrt{p} \times n^2/p$. The processor receiving the message retains the portion $(1/\sqrt{p} \times n^2/p)$ addressed to itself and its column neighbors and relays the rest

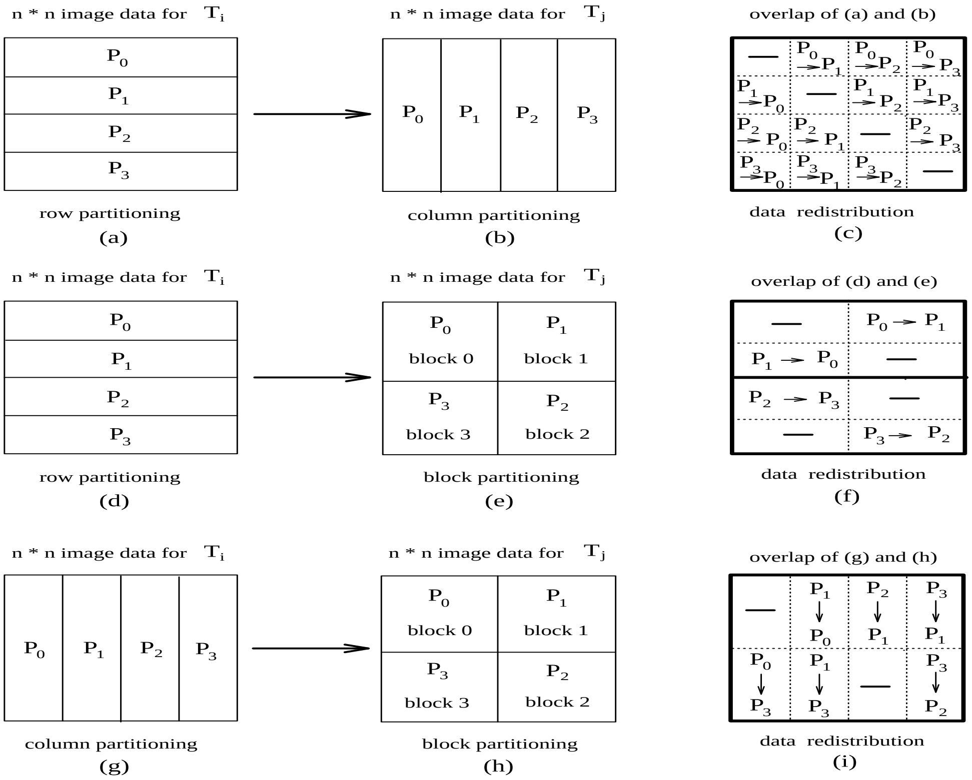


Figure 5: (a), (d), and (g) Task T_i uses four processors to operate on an $n \times n$ image partitioned row-wise. (b) Task T_j uses four processors to operate on an $n \times n$ image partitioned column-wise. (c) Data movement from a row partition to a column partition. Note that every row intersects with every column. (e) and (h) Task T_j now uses four processors to operate on an $n \times n$ image partitioned block-wise. (f) Overlap between a row partition and a block partition is more local with P_0 and P_1 communicating with each other while P_2 and P_3 communicating with each other. (i) Overlap pattern between the column and the block partitions.

$((\sqrt{p}-2)/\sqrt{p} \times n^2/p)$ to the next neighbor. The process repeats until the message circulates the whole ring. Hence the first phase communication time is:

$$\begin{aligned} DT_{single-phase} &= \textstyle\sum_{i=1}^{\sqrt{p}-1}(\alpha + \beta\frac{n^2}{p\sqrt{p}})(\sqrt{p}-1) \\ &= \alpha(\sqrt{p}-1) + \beta\frac{n^2}{p\sqrt{p}}(\sqrt{p}(\sqrt{p}-1) - \frac{\sqrt{p}(\sqrt{p}-1)}{2}) \qquad (1) \\ &= (\alpha + \tfrac{1}{2}\beta\tfrac{n^2}{p})(\sqrt{p}-1)\ . \end{aligned}$$

Phase II communication is identical to phase I except that phase II adopts a column-wise communication pattern. Hence, the total communication time for the two-phase data redistribution, DT, is twice that of a single phase. Therefore

$$
\begin{aligned}
DT &= 2\ DT_{single-phase} \\
&= (2\alpha + \beta\tfrac{n^2}{p})(\sqrt{p}-1).
\end{aligned} \tag{2}
$$

Of the 30 nontrivial redistributions, we verified that 26 satisfy the following conditions: (1) every processor holds a chunk of data in a source partition that intersect with those held by every other processor in the destination partition, (2) the size of the overlap is the same for all pairs of processors, and (3) data can be reshuffled using the two-stage scheme described above. Hence, data redistribution costs for these 26 patterns can be computed using Equation 2.

As mentioned before, two data redistributions involving row-to-block and block-to-row partitions have a more localized communication pattern, and it is not necessary for every processor to communicate with every other processor. As shown in Figure 5 (f), row-to-block and block-to row redistributions are for P_0 and P_1 to communicate with each other and P_2 and P_3 to communicate likewise. This is a pattern in which each processor exchanges data only with processors in the same row, so that the pattern becomes an all-to-all personalized communication on a ring [14]. At the first step, every processor sends a message of size $(\sqrt{p}-1)/\sqrt{p} \times n^2/p$ to the next processor. Each processor then retains $1/\sqrt{p} \times n^2/p$ image data which are addressed to itself, and transmits the remaining to the next processor. This step repeats $\sqrt{p}-1$ times. Therefore, the all-to-all personalized communication time on a ring with $\sqrt{p}$ processors is

$$
DT = (\alpha + \tfrac{1}{2}\beta\tfrac{n^2}{p})(\sqrt{p}-1). \tag{3}
$$

Column-to-block and block-to-column data redistributions use a row-column exchange pattern, where processors in row i exchange data with processors in column i. Data exchange can again be accomplished in two phases. In the first phase, data are accumulated in the processors on the main diagonal of the 2D mesh by shifting data to the left or right in each processor row. After data are accumulated in the processors along the main diagonal, data are redistributed to processors in the same column from the processor on the main diagonal. The row accumulation phase assumes a communication cost of $(\alpha+\beta n^2/p)(\sqrt{p}-1)$. One-way distribution in a column incurs a cost of $(\alpha + \beta n^2/(2\sqrt{p}))(\sqrt{p}-1)$. The total communication cost is thus $(2\alpha+\beta n^2\frac{2+\sqrt{p}}{2p})(\sqrt{p}-1)$. Data redistribution costs are summarized in Table 2.

5 Scheduling Algorithm

We are now ready to describe the scheduling scheme. The main purpose of the scheduling algorithm is to determine the proper data allocation scheme for every processor in each processing stage. This effort is aided by the taxonomy and the data redistribution costs established in the previous sections.

Inputs to the scheduler are a task chain with any number of nested loops, and a set of tunable task and resource parameters. The parameters may include the size of the input image (n) and those of the masks (r and s), number of processors (p), and architectural parameters α, β, and ω. Output of the scheduler is the processor assignment of each indi-

Table 2: Data redistribution costs of the six data partitioning schemes.

from-to	row	column	row-cyclic	column-cyclic	block	block-cyclic
row	-	DT_2	DT_2	DT_2	DT_1	DT_2
column	DT_2	-	DT_2	DT_2	DT_3	DT_2
row-cyclic	DT_2	DT_2	-	DT_2	DT_2	DT_2
column-cyclic	DT_2	DT_2	DT_2	-	DT_2	DT_2
block	DT_1	DT_3	DT_2	DT_2	-	DT_2
block-cyclic	DT_2	DT_2	DT_2	DT_2	DT_2	-

$$DT_1 = (\alpha+\beta\frac{n^2}{p}/2)(\sqrt{p}-1), DT_2 = (2\alpha+\beta\frac{n^2}{p})(\sqrt{p}-1), \text{ and } DT_3 = (2\alpha+\beta n^2\frac{2+\sqrt{p}}{2p})(\sqrt{p}-1).$$

vidual task in the chain and the associated data partitioning scheme. The resulting schedule optimizes the parallel time of the task chain over the pre-defined parameter ranges.

The Scheduling Scheme
`Step 1:` Decompose the task graph into a set of maximal data-independent/loop-free subchains.
`Step 2:` For each maximal subchain, reduce the length of the chain by merging task nodes representing pixel operations into their adjacent task nodes.
`Step 3:` For each reduced task chain, generate the corresponding scheduling graph.
`Step 4:` For each scheduling graph and possible parameter values, derive the optimal processor assignment and data partition using the shortest path algorithm.
`Step 5:` For each subchain, aggregate the edge and node costs of the corresponding edges and nodes in the scheduling graphs assuming different parameter values. For each cost-aggregated subchain, use the shortest path algorithm to find the optimal path, which corresponds to a schedule optimized over the specified range of the parameter.

We first discuss the individual steps in dealing with data dependence, loops, and subchains. Then we put them together and present the detailed algorithm.

Task Graph Decomposition An input task graph is broken into subsequences whenever a data-dependent node or a loop is encountered. This results in maximal data-independent/loop-free subchains, which are linear successions of tasks without loops.
`Data Dependent Nodes` We first assume that a data dependent operation has only a single parallel implementation. We isolate a data dependent node from a task graph, TG, in such a way that nodes and edges preceding the data-dependent node become a task graph (TG_1), and those following the data-dependent node become another task graph (TG_2). The data-dependent node becomes the dummy end node in TG_1 and the dummy starting node in TG_2 with its cost zeroed out. The next proposition shows that the whole operation sequence including the data dependent node can be optimized by optimizing the two task graphs individually.

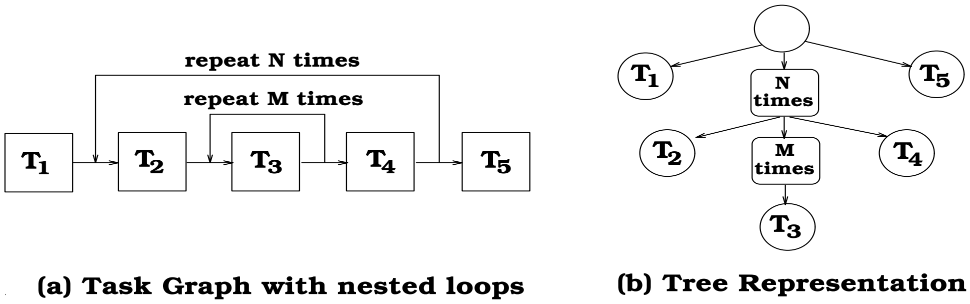


Figure 6: A task graph with simple loops.

Proposition 1 *Let a task graph TG be divided into TG_1 and TG_2 by a data-dependent task as described above. Then the whole sequence is optimized if TG_1 and TG_2 are optimized.*

Proof : Let PT_1 and PT_2 be the parallel times of the optimal paths in SG_1 and SG_2, where SG_1 and SG_2 are the scheduling graphs of TG_1 and TG_2, respectively. Let PT_i be the parallel time of the data-dependent task and $PT_{start,end}$ be the optimal parallel time of the whole operation. Then $PT_{start,end} = PT_1 + PT_i + PT_2$. (Since edges incident on and emitted from T_i are included in TG_1 and TG_2 respectively, data redistribution cost to and from T_i are parts of PT_1 and PT_2.) Assume $PT_{start,end}$ is not optimal. Then there exist certain sub-optimal paths in SG_1 and SG_2 with parallel times PT'_1 and PT'_2, respectively, such that $PT'_{start,end} = PT'_1 + PT_i + PT'_2$ and $PT'_{start,end} < PT_{start,end}$. Then

$$PT_{start,end} - PT'_{start,end} = (PT_1 - PT'_1) + (PT_2 - PT'_2) > 0 \ .$$

However, this is impossible, because $PT_1 \leq PT'_1$ and $PT_2 \leq PT'_2$. $\square$

`Loops` A simple loop, which is a loop without any conditional branching, divides a task graph into three subsequences: the sequence in front of the loop, that following the loop, and that enclosed in the loop. We therefore use the simple loop to partition a task graph into three stages. For nested loops, the subsequence enclosed by the outmost loop is recursively decomposed into three subsequences (Figure 6). The following proposition shows that each stage can be optimized independently.

Proposition 2 *Let a task graph TG be divided into three subgraphs, TG_1, TG_2, and TG_3. Assume TG_2 is enclosed in a simple loop, i.e., TG_2 repeats N times. Then we can optimize the whole sequence by optimizing TG_1, TG_2, and TG_3 individually.*

Proof : Let PT_1, PT_2, and PT_3 be the parallel times corresponding to the optimal paths of the three scheduling subgraphs. Let $PT_{start,end} = PT_1 + c_1 + N \times PT_2 + c_2 + PT_3$, where c_1 and c_2 are the data redistribution costs. Assume that $PT_{start,end}$ is not optimal. Then there exists another schedule, with a parallel time $PT'_{start,end}$ which is less than $PT_{start,end}$. Let $PT'_{start,end} = PT'_1 + c'_1 + N \times PT'_2 + c'_2 + PT'_3$, where PT'_1, PT'_2 and PT'_3 are the parallel times of other (suboptimal) paths in subgraphs TG_1, TG_2, and TG_3 respectively. Thus,

$$\begin{aligned} PT_{start,end} - PT'_{start,end} &= N\left(\tfrac{PT_1 + c_1 + c_2 + PT_3 - PT'_1 - c'_1 - c'_2 - PT'_3}{N} + (PT_2 - PT'_2)\right) \\ &> 0 \end{aligned}$$

To satisfy $(PT_{start,end} - PT'_{start,end}) > 0$ as $N \rightarrow \infty$, PT_2 must be greater than PT'_2. However, this is impossible as PT_2 is the parallel time of the optimal path for T_2. □
We note here that results of Proposition 2 is applicable to any level of nested loops.

The Reduced Task Graph The previous step breaks a task graph into several stages, each stage is now a maximal loop-free/data-independent subsequence. A task graph resulted this way in general may still comprise many operations. To reduce the complexity in finding the optimal schedule for a task graph, we want to reduce the length of such a graph by coalescing adjacent nodes into a single node. Note that by merging adjacent tasks into one node these operations will now assume the same data partitioning scheme. One useful heuristic is to coalesce a node representing a pixel operation with the operations immediately preceding or following it. Because the execution time of a pixel operation is independent of the data partitioning scheme used, a sequence of pixel operations can be merged into the adjacent masking operations or global operations without increasing the total parallel time (Proposition 3). This produces a reduced task graph, RTG, from a TG.

Proposition 3 *Given a schedule of a task graph, the schedule can be transformed into a new schedule by merging a node representing a pixel operation in the task graph with its adjacent node. The new schedule has a cost which is less than or equal to that of the original schedule.*

Proof : Let T_i denote a pixel operation, $T_{cp}(i)$ the computation time of T_i, and $T_{cm}(i)$ the communication time of T_i, which is zero. Let $PT_{k,l}$ be the parallel time from the kth task to the lth task in the task graph, then we have

$$PT_{start,end} = PT_{start,i-1} + c_1 + T_{cp}(i) + c_2 + PT_{i+1,end} ,$$

where c_1 and c_2 are the data redistribution costs in the original schedule. Assume T_i is merged into the previous node T_{i-1}. Then T_{i-1} and T_i use the same data partitioning scheme, which makes c_1 zero. Thus,

$$PT'_{start,end} = PT_{start,i-1} + T_{cp}(i) + c_2 + PT_{i+1,end}.$$

Since $PT'_{start,end} \leq PT_{start,end}$, merging a pixel operation cannot increase the parallel time of the whole chain of tasks. □

The Scheduling Graph For each node in an RTG, our algorithm first expands it into several nodes, each representing a data partitioning scheme which can potentially be employed at that particular stage. For example, if a particular node in the task graph represents a masking operation as established by the taxonomy, then it can assume six different data partitions: row, column, block, row-cyclic, column-cyclic, and block-cyclic. Furthermore, we assume that the blocking factor b in the cyclic patterns is such that the computation time is greater than the communication time for an efficient parallel implementation [8]. A node representing a global operation can be similarly expanded; however, legal data partition schemes may be limited to those supported by the library of existing parallel codes. For example, if 2D FFT is implemented as a two-pass operation based on 1D FFT, which assumes either a row or a column partition, then only two nodes will be generated for the 2D FFT, representing the row and column partitions. We call graphs constructed this way the *scheduling graphs* or SG.

Since nodes representing pixel operations have been absorbed into adjacent nodes representing masking or global operations, the types of nodes in the RTG are either masking or global. For masking operations, their costs can be estimated as described in Section 3. Computation and communication costs of a global operation using various data partitioning schemes are expected to have been registed in the parallel library.

A node in an RTG is connected to nodes at the following stage by an edge which represents the needed data redistribution in between them. Edge costs are determined by the particular data partitioning schemes used in the two nodes and these costs have been tabulated in Section 4. A single dummy starting node is prepended and a single dummy end node is appended to serve as the source and the sink of the graph, respectively.

Shortest Path Searching The computation and communication cost of each node and the data redistribution cost of each edge in an SG are resolved using the given set of parameter values. Then a shortest path algorithm [4] is employed to find the shortest path from the dummy starting node to the dummy end node. Because a scheduling graph is a dag, the shortest path can be found efficiently in $O(|V| + |E|)$ time. Nodes along the path give the data allocation schemes for tasks in the SG. Once we determine the best schedules for all subsequences, the results can be easily concatenated to give the optimal data partitions for the original task graph. This step thus determines the optimal schedule for a particular setting of the task parameters.

Minimum Performance Variation Scheduling So far, the optimal scheduling results are obtained for a fixed set of parameters (e.g., a fixed image size). If parameters are allowed to change, the scheduler may generate different optimal paths. For example, the shortest path for input image size of 64×64 could be different from that for image size of 1024×1024. Furthermore, a schedule may change if different numbers of processors are used. However, it is not advisable to produce different parallel codes for different parameters as there many be many parameters and many different settings of them. Generating a different parallel code for each possible setting of a parameter may lead to an exponential explosion of the code size. Hence, a more practical approach is to generate one parallel code which provides moderate performance variation over a predefined range of parameter values. We call this schedule the minimum performance variation schedule (MPVS).

Let OPT_k be the parallel time of the optimal path with the kth parameter setting, where $k = 1 \ldots m$, and m is the number of valid parameter values. Then the MPVS is a schedule which minimizes the metric mean performance ratios (MPR):

$$MPR = \frac{1}{m} \sum_{k=1}^{m} \left| \frac{GOPT_k}{OPT_k} - 1 \right| , \tag{4}$$

where $GOPT_k$ is the parallel time of the MPVS path with the kth parameter setting.

For example, if images may assume three different sizes, say, $256 \times 256, 512 \times 512$, and 1024×1024, then $m = 3$ and OPT_1 corresponds to the parallel time of the optimal path for 256×256 input images, OPT_2 the parallel time of the optimal path for 512×512 input images, etc. $GOPT_1$ is then the parallel time of the MPVS path (which minimizes the MPR) when running the algorithm on 256×256 images. $GOPT_2$ and $GOPT_3$ are defined in a similar manner.

A brute force method to compute the MPVS is to try all possible paths in the m scheduling graph variances. However, the cost can be as high as $O(6^l * m)$, where l is the number of nodes in the scheduling graph, and m the number of valid parameter values. Instead, we construct a new scheduling graph such that the topology of the graph is the same as the graph generated in step 3 of the scheduling scheme, but the weight of node i is now $\sum_{k=1}^{m} \frac{v_k^i}{OPT_k}$, and that of edge j is $\sum_{k=1}^{m} \frac{e_k^j}{OPT_k}$. The following proposition shows how to compute the MPVS using this new, cost-aggregated scheduling graph.

Proposition 4 *Given a task chain, the schedule derived below minimizes the MPR with respect to a range of different parameter settings.*

Proof: Since $OPT_k \leq GOPT_k$ for all k, we need to minimize $\sum_{k=1}^{m} \frac{GOPT_k}{OPT_k}$ in order to minimize the MPR. Assume $\{v^1, e^1, v^2, \ldots, e^{l-1}, v^l\}$ is the ordered list of nodes (v^i) and edges (e^j) on the path that minimizes the MPR. Let a node value, v_k^i, be the costs of the ith node along the path with the kth parameter setting. Edge weights are defined similarly. Then, we have

$$\sum_{k=1}^{m} \frac{GOPT_k}{OPT_k} \quad = \quad \sum_{k=1}^{m} \frac{v_k^1}{OPT_k} + \sum_{k=1}^{m} \frac{e_k^1}{OPT_k} + \ldots + \sum_{i=k}^{m} \frac{v_k^l}{OPT_k}. \tag{5}$$

Thus, if we compute the new edge and node costs as outlined earlier by aggregating the corresponding edge and node costs in the m scheduling graph variances, then the shortest path of the new scheduling graph minimizes MPR. Furthermore, the complexity is only $O(lm)$, which represents the cost aggregation from the m SG variances and the shortest path searching. □

6 Experimental Results

We tested the scheduling scheme on an image processing operation for noise reduction and image enhancement. The input image was assumed to be corrupted by a multiplicative noise process and an additive noise process. The image processing operation used histogram equalization to stretch the image intensities back to the original range, and followed by median filtering to remove "salt-and-pepper" additive noise. An ln operator was then applied to separate the multiplicative noise from the image itself. The multiplicative noise was removed by a filtering operation in the frequency domain (through Fourier and inverse Fourier transforms). Finally, the image was scaled back using an exponent operator.

An implementation of the above algorithm using `Khoros` is shown in Figure 7. The implementation also included input/output and data type conversion operations, which are used for inputing image files, displaying image files, and conversion among different data formats (e.g., from bytes to floats).

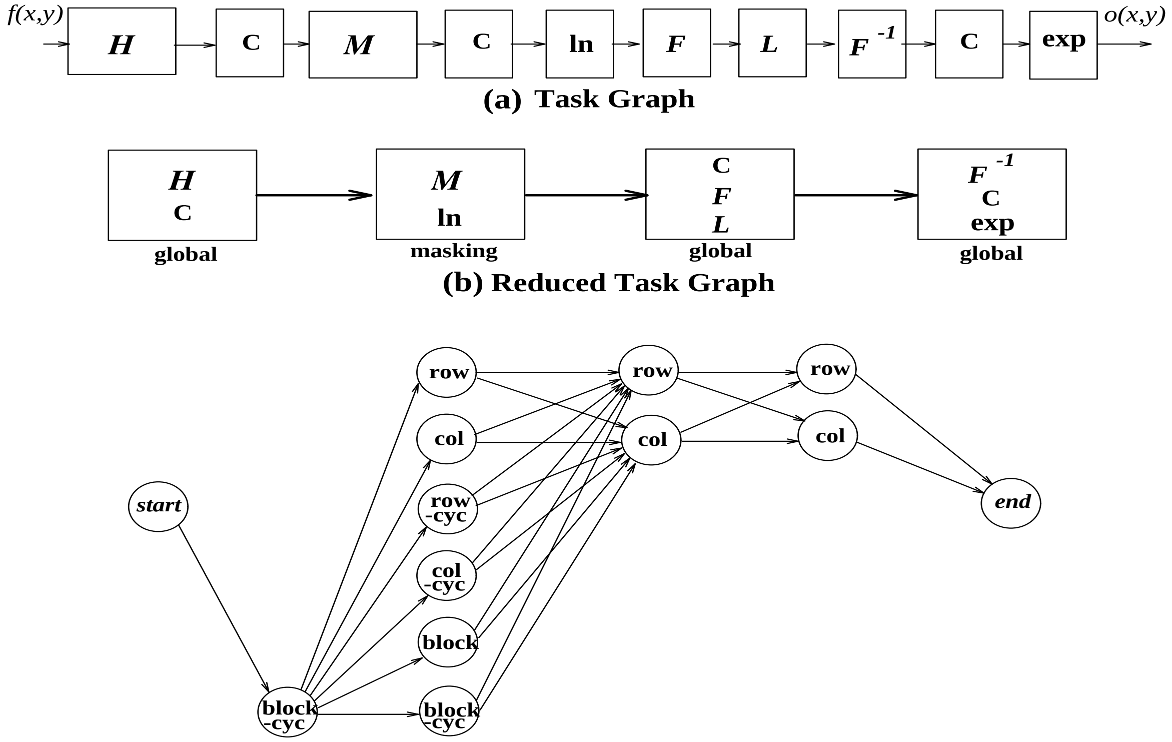
A task graph of this operation is shown at Figure 8(**a**), where the input/output routines used in `Khoros` were omitted. Figure 8(**b**) is the reduced task graph of the operation. Since

Figure 7: An image processing operation constructed using `Khoros`. The image shown at bottom left is the input image, and The image shown at bottom right is the output image from the operation.

the type conversion operators and the ln operator were both pixel operators, they were merged into the adjacent histogram equalization $\mathcal{H}$ and the median filtering masking operation $\mathcal{M}$. The frequency filter L and the exponent operator `exp`–both pixel operations–were merged into the Fourier transform $\mathcal{F}$ and the inverse Fourier transform $\mathcal{F}^{-1}$, respectively.

Figure 8(**c**) shows the scheduling graph created from the reduced task graph in Figure 8(**b**). Edges incident on the starting and the end nodes were of zero weight. Costs of the other edges were tabulated in Table 2. The first node in this RTG represented a histogram equalization operation, which had only a single parallel implementation based on the block-cyclic partition in the library. The next node was a masking operation; hence, the node was expanded into six nodes in the scheduling graph using six different data partitioning schemes. The Fourier transform operation assumed two different implementations. Both of them used the transpose algorithm for the parallel 1D FFT operation. For 2D FFT [7], one library routine used row partitioning for 1D FFT, followed by a transpose operation, and then by another row-wise 1D FFT. The other routine was similar but assumed a column partition for 1D FFT instead. The inverse Fourier transform operation used the same data partitioning schemes as the forward transform.

In our experiments, we assumed $r = s = 3, n = 1024$, and $p = 64$. We used an INTEL iPSC/860 architecture with $\alpha = 136\mu sec$, $\beta = 1.6\mu sec/word$, $\omega = 0.2\mu sec/flop$; an nCube architecture with $\alpha = 200\mu sec$, $\beta = 2.4\mu sec/word$, $\omega = 1.6\mu sec/flop$; and an INTEL Paragon architecture with $\alpha = 250\mu sec$, $\beta = 0.1\mu sec/word$, $\omega = 0.2\mu sec/flop$. b, the blocking factor, was assumed to be 8 to assure an efficient parallel implementation where the local communication cost did not dominate the local computation cost [8]. Using the



(c) Scheduling Graph

Figure 8: (a) A task graph for a noisy-reduction filter, (b) the reduced task graph, and (c) the scheduling graph

INTEL iPSC/860 parameters, the shortest path in Figure 8 was found to pass through the block-cyclic partition node for the histogram equalization operation, the row partition node for the masking operation, the row partition node for the Fourier transform, and the row partition node for the inverse Fourier transform. The estimated optimal parallel time using the following equation was $1.57 \times 10^6 \mu sec$

$$\begin{aligned} PT \;=\; & (\tfrac{2n^2\omega}{p} + 2\alpha(\sqrt{p}-1) + \beta(p-1) + \tfrac{n^2\omega}{p}) + (\tfrac{n^2 rs\omega}{p} + \tfrac{(r-1)p}{n}(\alpha + \beta n^2/p) + \\ & \tfrac{n^2\omega}{p}) + (\tfrac{22n^2\omega}{p}\log n + (2\alpha + \beta\tfrac{n^2}{p})(\sqrt{p}-1) + 2\tfrac{n^2\omega}{p}) + (\tfrac{22n^2\omega}{p}\log n + \\ & (2\alpha + \beta\tfrac{n^2}{p})(\sqrt{p}-1) + 2\tfrac{n^2\omega}{p}) \,. \end{aligned} \tag{6}$$

Denote the speedup factor as $SP = ST/PT$, where ST is the sequential execution time and PT the parallel execution time. SP is a measure commonly used to show how much performance gain is achieved by parallelizing a given application over its sequential implementation. ST of this operation was estimated to be

$$\begin{aligned} ST &= 8n^2\omega + n^2 rs\omega + 44n^2\omega\log n \\ &= n^2\omega(rs + 44\log n + 8)\,, \end{aligned} \tag{7}$$

where the histogram equalization operation needed $2n^2\omega$ operations; the pixel operations, ln, L and exp, had a complexity of $n^2\omega$ operations each; the median filter using an $r \times s$ mask consumed $n^2 rs\omega$ operations; and the Fourier and the inverse Fourier transforms used $22n^2\log n\omega$ operations each (the large constant was due to operations with complex

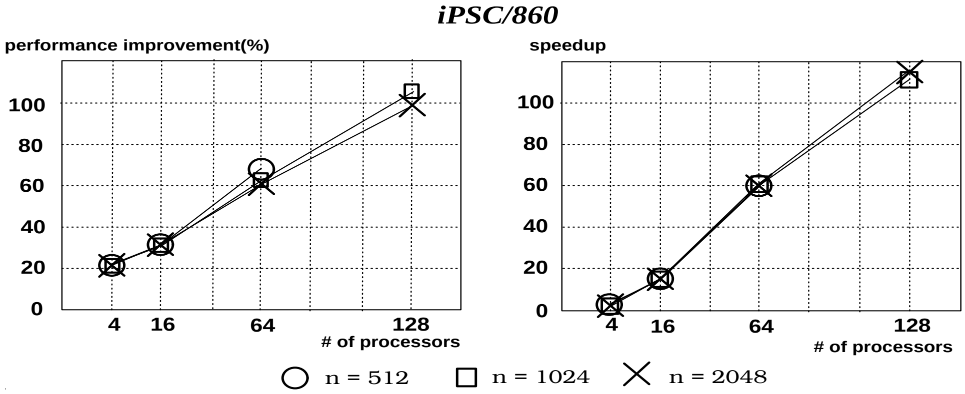


Figure 9: Simulation on Intel iPSC/860. The performance improvement was measured using the worst case schedule against the optimal.

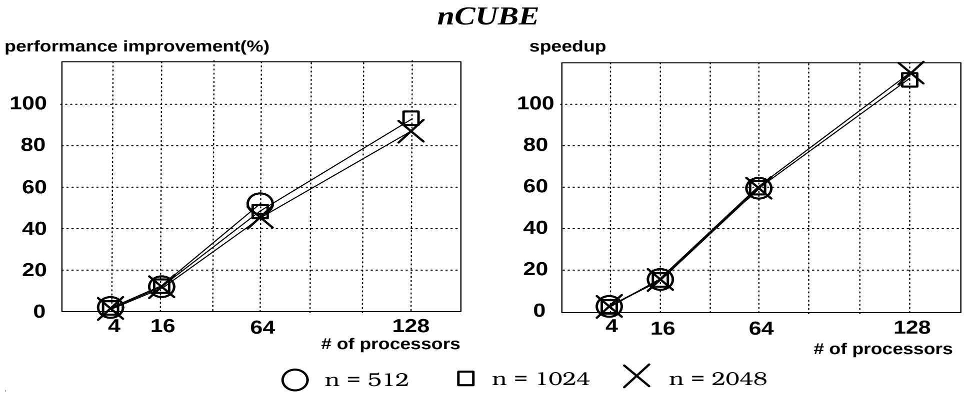


Figure 10: Simulation on nCUBE. The performance improvement was measured using the worst case schedule against the optimal.

numbers). Using the above parameters, we estimated the sequential time to be $9.58 \times 10^7 \mu sec$, which translated into a speedup factor of $SP = 61.07$ for using 64 processors.

A scheduler is essential in achieving a reasonable speed up, as suboptimal schedules can drastically reduce the efficiency of a parallel execution. For example, the longest path in Figure 8, which was the result of the worst case scheduling, went through the block-cyclic partitioning node for the histogram equalization operation, the block partitioning node for the masking operation, the column partitioning node for the Fourier transform, and the row partitioning node for the inverse Fourier transform. And the estimated parallel time for the worst case scenario was $1.64 \times 10^6 \mu sec$. If we define the performance improvement as

$$\text{performance improvement} = (\text{suboptimal parallel time})/(\text{optimal parallel time}) - 1,$$

then the performance improvement in this particular case was $1.64 \times 10^6 / 1.57 \times 10^6 - 1 \approx 43\%$. Or the proposed scheduling algorithm generated a parallel code 43% faster than the worst-case parallel code.

Figures 9-11 show the performance improvement and speedup factors using various settings of the task and resource parameters. The performance improvement was measured using the worst case schedule against the optimal. The number of processors used could go

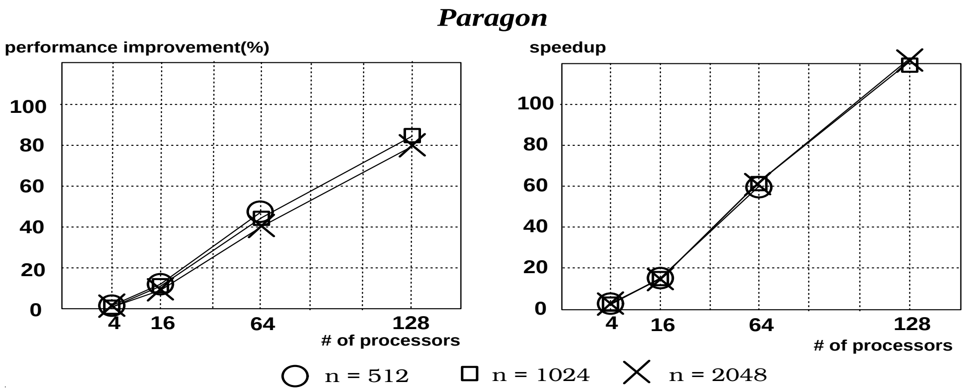


Figure 11: Simulation on Intel Paragon. The performance improvement was measured using the worst case schedule against the optimal.

from 4 to 128 and the image size varied from 512×512 to 2048×2048. These figures show that the worst case performance degraded as the number of processors increased. This was probably caused by a large communication overhead resulted from poor scheduling of many processors. Our analysis for $n = 1024$ and $p = 128$ showed the performance improvement could be as high as 85%, i.e., the optimal schedule generated a parallel code 85% faster than the worst case (Figure 9). These figures showed that appropriate schedules could reduce parallel execution time significantly.

In these figures, we can also see that SP increased with the number of processors used. As shown in Equation 7, ST is independent of p. PT is inverserly proportional to p, thus SP increases with p. In our simulations, the SP's were almost the same as the number of processors used, which demonstrated a very efficient utilization of processors. These figures showed that, with a proper schedule, a task of CVIP operations can be implemented on various parallel architectures efficiently.

To compute the MPVS, the parallel time of the optimal schedule assuming a given set of task and resource parameters was computed first. These optimal parallel times assuming a particular set of parameters were then used to generate an augmented scheduling graph for computing the MPVS for the above example. We chose the nCUBE architecture with $b = 8$, $r = s = 3$, $n = 512$ and $p = 4, 16, 64$, and the optimal parallel times for $p = 4, 16, 64$ were $4.47 \times 10^7 \mu sec, 1.14 \times 10^7 \mu sec$, and $2.98 \times 10^6 \mu sec$, respectively. Their corresponding optimal paths are shown in Figures 12(a)-(c). Since the computation times of the histogram equalization and the type conversion operations using the block-cyclic data partition were $3.15 \times 10^5 \mu sec, 7.99 \times 10^4 \mu sec$, and $2.26 \times 10^4 \mu sec$ for $p = 4, 16$, and 64, the aggregated node cost was thus

$$\begin{aligned} v^1_{GOPT} &= \tfrac{3.15\times10^5 \mu sec}{4.47\times10^7 \mu sec} + \tfrac{7.99\times10^4 \mu sec}{1.14\times10^7 \mu sec} + \tfrac{2.26\times10^4 \mu sec}{2.98\times10^6 \mu sec} \\ &= 0.022. \end{aligned} \tag{8}$$

Similarly, the aggregated edge costs from the block-cyclic histogram equalization node to the six masking operation nodes were

$$\begin{aligned} e^1_{GOPT} &= \tfrac{1.58\times10^5 \mu sec}{4.47\times10^7 \mu sec} + \tfrac{1.19\times10^5 \mu sec}{1.14\times10^7 \mu sec} + \tfrac{7.16\times10^4 \mu sec}{2.98\times10^6 \mu sec} \\ &= 0.55, \end{aligned} \tag{9}$$

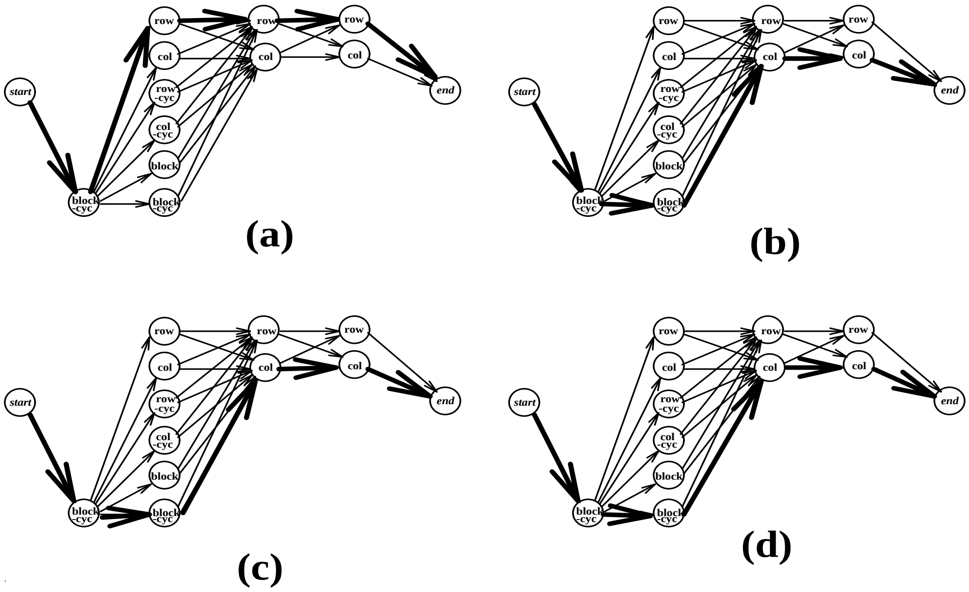


Figure 12: The optimal paths on nCUBE with $r = s = 3, b = 8$, and $n = 512$. (a) $p = 4$, (b) $p = 16$, (c)$p = 64$, (d) the minimal performance variation path for $p = 4, 16$, and 64.

because the data redistribution cost from a block-cyclic partition to any other partition was DT_2 in Table 2, and DT_2 with $p = 4, 16, 64$ were $1.58 \times 10^5 \mu sec, 1.19 \times 10^5 \mu sec$, and $7.16 \times 10^4 \mu sec$.

Once the new scheduling graph was generated, the optimal path of this graph was computed by the shortest path algorithm (Figure 12(d)). The shortest path of the aggregated scheduling graph passed through the block-cyclic node for the histogram equalization operation, the block-cyclic partition node for the masking operation, and the column partition nodes for the Fourier transformation, and the inverse Fourier transformation. The shortest path defined the MPVS which minimized performance variation over $p = 4, 16, 64$. In this case the MPR was

$$\begin{aligned} MPR &= \tfrac{1}{3}\left(\left|\tfrac{4.47\times10^7 \mu sec}{4.47\times10^7 \mu sec} - 1\right| + \left|\tfrac{1.14\times10^7 \mu sec}{1.14\times10^7 \mu sec} - 1\right| + \left|\tfrac{2.99\times10^6 \mu sec}{2.98\times10^6 \mu sec} - 1\right|\right) \\ &= 0.0011. \end{aligned} \tag{10}$$

The given example show that a schedule was found which had a 0.11% performance variation over the three different numbers of processors used. The MPR thus provides a good measure to determine how many different parallel codes are needed. For a pre-defined set of parameters, one parallel code may be sufficient if the MPR is small. If the MPR is too large, then the parameter set should be divided into small subsets such that the MPR of schedules in each parameter subset is less than a preset threshold.

7 The Concluding Remarks

In summary, we presented a preliminary design of a static global scheduler for CVIP operations. The scheduler is for automatically generating processor assignment and data partitioning schemes to achieve a reasonable parallel performance without user intervention. This facilitates a transparent migration from a single processor environment to a multiprocessor one with a minimum amount of effort.

The scheduler's design takes into consideration both the *intra-operation* communication and computation complexity and the *inter-operation* data exchange overhead in formulating optimal data partitioning and processor assignment schemes. It also tolerates variation in the task and resource parameters. Using an image processing taxonomy and the available data partitioning types, it is possible to formulate and tabulate in advance most of the communication, computation, and data exchange costs. The scheduler can then retrieve these cost parameters through an efficient table lookup process and minimizes the total operation cost using a shortest path algorithm.

Our future research plan is to construct a prototype of such a scheduler. This will include finding the cost expressions for other connection topologies–such as the fat tree topology used by the MEIKO parallel machine, constructing a library of parallel CVIP modules, and implementing the parallel code generator.

References

[1] M. Annaratone *et al.*, *The Warp Computer: Architecture, Implementation and Performance,* IEEE Trans. Computers, Vol. C-36, No. 12, Dec. 1987, pp. 1523-1538.

[2] R. F. Browne and R. M. Hodgson, *Mapping Image Processing Operations onto Transputer Networks,* Microprocessors and Microsystems, Vol. 13 No. 3, April 1989, pp.203-211

[3] P. K. Biswas, J. Mukherjee, and B. N. Chatterji, *Component Labeling in Pyramid Architecture,* Pattern Recognition, Vol. 26, No. 7, 1993, pp. 1099-1115.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[5] *IRIX Explorer User's Guide,* Silicon Graphics, Inc., Mountain View, CA. 1992.

[6] M. Gupta and P. Banerjee, *Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers,* IEEE Trans. Parallel and Distributed Systems, Vol. 3, No. 2, Mar. 1992, pp. 179-193.

[7] R C. Gonzalez and R. E. Woods, *Digital Image Processing,* Addison-Wesley Pub. Company, 1992.

[8] A. Gerasoulis and T. Yang, *On the Granularity and Clustering of Directed Acylic Task Graphs,* IEEE Trans. Parallel and Distributed Systems, Vol. 4, No. 6, June 1993, pp. 686-701.

[9] L. H. Jamieson, *et al.*, *Parallel Scalable Libraries and Algorithms for Computer Vision,* Proceedings of the 12th IAPR Int. Conf. on Pattern Recognition, Jerusalem, Israel, Oct. 1994, pp. 223-228.

[10] L. H. Jamieson, E. J. Delp, J. N. Patel, C.-C. Wang, and A. A. Khokhar, *A Library-Based Program Development Environment for Parallel Image Processing,* Proceedings of Scalable Parallel Libraries Conference, Mississippi State Univ., Mississippi, Oct. 1993, pp. 187-194.

[11] J. F. Jeng and S. Sahni, *Reconfigurable Mesh Algorithms for the Hough Transform,* Journal of Parallel and Distributed Computing, Vol. 20, 1994, pp. 69-77.

[12] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel, "The High Performance Fortran Handbook", MIT Press, Cambridge, Mass., 1994.

[13] V. K. P. Kumar (Ed.), *Parallel Architectures and Algorithms for Image Understanding,* Academic Press Inc., Boston, Mass., 1991.

[14] V. K. P. Kumar, *et al.*, *Introduction to Parallel Computing: Design and Analysis of Algorithms,* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA., 1994.

[15] S.-Y. Lee and J. K. Aggarwal, *A System Design/Scheduling Strategy for Parallel Image Processing,* IEEE Trans. PAMI, Vol. 12, No. 2, Feb. 1990, pp. 194-204.

[16] C. Lee, Y. F. Wang, D. Uecker, and Y. Wang, *Image Analysis for Automated Tracking in Robot-Assisted Endoscopic Surgery,* Proceedings of 12th Int. Conf. on Pattern Recognition, Jerusalem, Israel, Sep. 1994, pp. 88-92.

[17] J. Rasure and S. Kubica, *The Khoros Application Development Environment,* Khoral Research Inc., Albuquerque, New Mexico, 1992.

[18] A. P. Reeves, *Parallel Programming for Computer Vision,* IEEE Software, Vol. 8, No. 6, Nov. 1991, pp. 51-59.

[19] H. J. Siegel, J. B. Armstrong, and D. W. Watson, *Mapping Computer-Vision-Related Tasks onto Reconfigurable Parallel-Processing Systems,* IEEE Computer, Vol. 25, No. 2, Feb. 1992, pp.54-63.

[20] J. A. Webb, *Steps Toward Architecture-independent Image Processing,* IEEE Computer, Vol. 25, No. 2, Feb. 1992, pp. 21-31.

[21] J. A. Webb, *High Performance Computing in Image Processing and Computer Vision,* Proceedings of 12th IAPR Int. Conf. on Pattern Recognition, Jerusalem, Israel, Oct. 1994, pp. 218-222.

[22] C. C. Weems, *et al., Parallel Processing in the DARPA Strategic Computing Vision Program,* IEEE Expert, Vol. 6, No. 5, Oct. 1991, pp. 23-38.

[23] F. Weil, L. H. Jamieson, and E. J. Delp, *Dynamic Intelligent Scheduling and Control of Reconfigurable Parallel Architectures for Computer Vision/Image Processing,* Journal of Parallel and Distributed Computing, Vol. 13, 1991, pp. 273-285.

[24] S. Wharton and J. Newcomer, *Land Image Data Processing Requirements for the EOS Era,* IEEE Trans. Geoscience and Remote Sensing, Vol. 27, No. 2, Mar. 1989, pp. 236-242.

[25] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers,* MIT Press, Cambridge, Mass., 1989.

[26] R. Wolski and J. Feo, *Program Partitioning for NUMA Multiprocessor Computer Systems,* Journal of Parallel and Distributed Computing, Vol. 19, No. 3, Nov. 1993, pp. 203-218.

[27] T. Yang and A. Gerasoulis, *PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors,* Proceedings of 6th ACM Int. Conf. on Supercomputing, Washington D.C., Jul. 1992, pp. 428-437.