Global Optimization for Mapping Parallel Image Processing Tasks on Distributed Memory Machines

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

Many parallel algorithms and library routines are available for performing computer vision and image processing (CVIP) tasks on distributed-memory multiprocessors. The typical image distribution may use column, row, and block based mapping. Integrating a set of library routines for a CVIP application requires a global optimization for determining the data mapping of individual tasks by considering inter-task communication. The main difficulty in deriving the optimal image data distribution for each task is that CVIP task computation may involve loops, and the number of processors available and the size of the input image may vary at the run time. In this paper, a CVIP application is modeled using a task chain with nested loops, specified by conventional visual languages such as `Khoros` and `Explorer`. A mapping algorithm is proposed that optimizes the average run-time performance for CVIP applications with nested loops by considering the data redistribution overheads and possible run-time parameter variations. A taxonomy of CVIP operations is provided and used for further reducing the complexity of the algorithm. Experimental results on both low-level image processing and high-level computer vision applications are presented to validate this approach.

1 Introduction

Computer Vision and Image Processing (CVIP) algorithms possess characteristics which are ideally suited for implementation on a variety of parallel architectures [29, 30]. Many efficient parallel CVIP algorithms have been proposed on distributed memory machines [7, 25]. However, a CVIP application usually employs many individual algorithms, and optimizing individual tasks in a processing pipeline comprising many tasks does not guarantee the optimal performance of the whole sequence of operations. This is because, 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., the input image size and the number of processors available). This research thus aims at developing a global optimization scheduler–which takes into consideration the data shuffling overheads in between processing stages and possible run-time variation of the task and resource parameters–to generate data partition and processor assignment schemes for optimizing an entire sequence of operations, instead of individual tasks [18, 19].

The need for such software systems for effectively integrating and utilizing existing libraries of parallel algorithms has been recognized before. For example, This research direction has been demonstrated in the ScaLAPACK [8] for general scientific computing and in the DISC and other image library projects [13, 31].

In this paper, we have developed a mapping scheme as the first step to realize a parallel visual programming system based on `Khoros` [17, 18, 19, 23]. Khoros has been successfully utilized for many CVIP and scientific applications. In `Khoros`, a user employs a visual programming language to specify CVIP operations as task graphs and the system employs existing library routines to process the required computation. A natural extension is to develop a parallel Khoros system which takes the user specification as an input, and generates parallel codes that call parallel library routines based on the user specification. Such a system is beneficial in that it facilitate the migration from a single-processor paradigm to a multiprocessor one in a *transparent* manner. With the proposed system, vision researchers can still use the visual programming environments, such as `Khoros` and `Explorer` which they are accustomed to, for designing their algorithms. The system will then analyze the corresponding task graphs automatically and determines suitable processor allocation and data partitioning schemes. Such an idea can be found in the graphical parallel programming tool research [4, 3] which uses coarse-grain computation graphs to model user applications.

Here, we address a fundamental mapping problem arising in realizing such a parallel programming environment. A library routine typically provides a few alternative image data distribution patterns, and the computation mapping can be determined by the data mapping (e.g., the owner computes rule). A typical mapping scheme for 2D image data is row, column, or block based, and we will assume the basic distribution strategies used in HPF [14]. Each task may employ a different mapping scheme to exploit data parallelism, as one mapping may be better than another in terms of individual task performance. However, a CVIP application is usually composed of a sequence of processing operations, the system hence must make a decision on the choice of mapping for each task. The mapping inconsistency between two tasks causes data redistribution, which incurs a significant amount of communication overhead on distributed memory machines. Thus it is necessary to study the selection of data mapping for each task and its impact on the global performance. Furthermore, there are other complications in determining the mapping. As mentioned before, the run-time variation of the task and resource parameters, such as the image sizes and the number of processors used, can also affect the choice of mapping and should be considered.

We assume that a task computation specification produced by Khoros is a chain of data-parallel tasks with nested loops. The goal of optimization is to achieve the minimum overall processing time by assigning multiple processors to each task. We present an algorithm that determines task data distribution schemes to optimize the average global run-time performance.

In certain CVIP applications, component tasks may involve branching in the code and thus their complexity is dependent on the content of the input images. These tasks are called data-dependent. Our work addresses data-dependent computation to a limited degree, and mainly deals with applications which don't have an excessive number of branching and data-dependent tasks. We will discuss a taxonomy of CVIP operations that helps our algorithm in identifying the data-dependent regions in a task chain and in merging adjacent tasks in a chain to reduce the mapping complexity.

The remainder of this paper is organized as follows: Section 2 summarizes related research work. Section 3 develops a CVIP taxonomy and tabulates data redistribution costs for later use in developing the main scheduling algorithm. In Section 4, an optimization algorithm for mapping is described. Section 5 presents some experimental results and, finally, Section 6 contains concluding remarks.

2 Related Research Work

There have been a number of compiler projects aimed at developing software environments 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, Lee and Aggarwal [16] discussed static and dynamic design/scheduling strategies for image processing operations comprising a linear sequence of tasks. Although this research provided a useful design framework for parallel image processing, it is not applicable to message-passing architectures and does not consider the communication overheads for redistributing image data.

DISC [31] uses dynamic scheduling in handling data-dependent tasks and conditional branches on a dynamic reconfigurable and repartitionable machine (PASM). Run-time dynamic scheduling is advantageous in dealing with data dependent and conditional operations. But dynamic scheduling also incurs a high control overhead on distributed memory machines. Compared with DISC, our research focus is different. We mainly address issues in mapping parallel tasks with data-independent computations, and discuss how we can handle loops and avoid the use of dynamic scheduling in dealing with run-time parameter variation. We do not want to imply that static scheduling is a feasible choice for data-dependent operations on distribution machines and it is indeed an open problem how to devise effective dynamic scheduling for efficient data-dependent computation on distributed memory machines.

Library-based parallel system development tools were discussed in [13, 24]. Reeves [24] constructed a parallel library of CVIP routines, but he did not provide scheduling support for parallel execution. Jamieson, *et al.* [13] illustrated a user-friendly interface to a parallel library, which stores 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. These systems can benefit from the result of this paper.

Our model uses task graphs where each task is parallelizable. This model can be also found in the recent work for exploiting data and task parallelism, e.g. [22, 32]. Their work deals with DAGs of fixed data distribution and processor parameters, but not with graphs with loops. In [12], techniques for optimizing the data distribution are presented for nested loops, which can be viewed as the optimization for one macro task. In contrast, our current scheme uses domain knowledge and deals with the data distribution for task graphs with loops and the goal is to minimize the overall parallel time.

3 Preliminary

Our work is targeted at distributed memory machines. Each processor has its own local memory, and processors communicate by sending messages. The cost of sending a message is therefore $\alpha + \beta \times M$, where M is the size of the message, α is the latency and β is the transmission speed. We denote the processor speed as ω, which is the time for a floating point multiplication. We assume that each library task has an associated cost function using the above parameter and the data redistribution cost is also available. For ease of illustration, we provide the redistribution cost functions for a 2D square wrap-around mesh or a fully connected machine using worm-hole routing. In this section, we will define the optimization problem, and present a taxonomy of image operations and tabulate the data redistribution costs.

3.1 The Optimization Problem

The input CVIP task graph is assumed to be a linear chain of operations with any number of nested loops. A recursive definition of a task graph is: $G = (G_1, G_2, \cdots, G_h)$, which represents a sequence of computation subgraphs $G_1, \cdots, G_h$. Each subgraph G_i could be a single task T_i, or a loop structure $M_i(G)$ where M_i is the iteration number or conditions for executing subgraph G. We use a tree to depict the nested loop control structure in an input task graph. For example, the tree representation of a task graph is shown in Figure 1. The tree leaf nodes represent the basic tasks and the internal nodes represent the number of iterations.

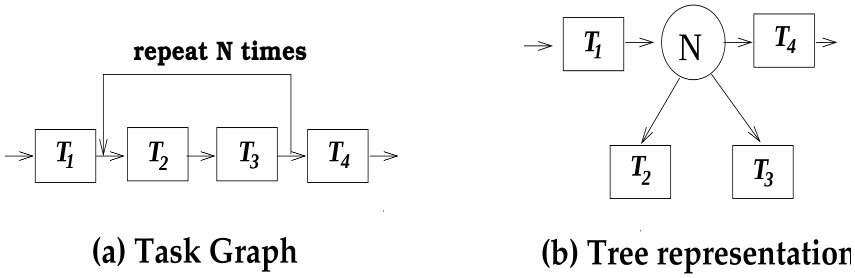


Figure 1: (a) A task chain with a loop, and (b) its tree representation.

Let T be a task in a task graph G and let p be the number of processors available. We assume that task T will use all p processors and T employs one of the following six data distribution schemes based on the High-Performance Fortran standard [14]: row, column, row-cyclic, column-cyclic, block, and block-cyclic partitions. The assignment of computation in each task to the p processors is based on the data distribution pattern assumed for that particular task. Note that an algorithm may have different performance using different distribution patterns. For example, the 2D "row-column" parallel FFT algorithm using the row or column partition outperforms the 2D decimation FFT algorithm adopting a block partition for small problem-size/machine-size ratios in coarse-grained machines [13]. Since we will employ existing parallel algorithm libraries to execute individual tasks in a graph, the possibility that some tasks may be implemented with a few fixed choices of data distribution must be considered. 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 data distribution schemes are possible for the 2D FFT. For the block-cyclic partitioning, we assume that the blocking factor b in the cyclic pattern is fixed such that the computation time is greater than the communication time for an efficient parallel implementation [10].

The following definitions will be used in this paper:

- We model the total parallel time of task T_i as $PT(T_i) = f_i(D_i, Z)$, where Z is a set of run-time parameters such as the number of processors p used and the image size. D_i is a feasible data distribution strategy for T_i provided in a library implementation. We may also use the simplified notation $DT(T_i)$ in this paper.
- If the data distribution patterns assumed in two adjacent tasks are denoted as D_s and D_t, we can model the redistribution cost as $DR(D_s, D_t, Z)$.
- $DE(G)$ is the entry distribution of a subchain G, which is the distribution of the left and bottom most task in the tree representation of the subchain G.
- $DX(G)$ is the exit distribution of a subchain G, which is the distribution of the right and bottom most task in the tree representation of the subchain G.
- Given a graph G, the parallel time is recursively defined as follows:
  1. If G is a leaf node T_i, $PT(G) = f_i(D_i, Z)$.
  2. If G is a loop node $N_i(H)$, $PT(G) = N_i \times PT(H)$.
  3. If $G = (G_1, G_2, \cdots, G_h)$,

$$PT(G) = PT(G_1) + DR(DX(G_1), DE(G_2), Z) + PT(G_2) + \dots .$$

	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 2: A taxonomy of computer vision and image processing operations.

The goal of optimization is therefore to find the data distribution scheme for each task in a task graph such that the overall parallel time is minimized.

Some operations in CVIP applications have a time complexity dependent on the value of the image data, e.g., in the eigenvalue analysis. Another factor that complicates the optimization process is that the number of processors available and the size of the image may vary for each run. Our goal is to produce a schedule that has a good average performance, allowing the task parameters to vary at the run time. The advantage of finding a good average schedule is that there is no need to produce multiple mappings tuned for different parameter settings, which may lead to an exponential growth of the code size.

3.2 Classification of Image Operations

In this section, we present a taxonomy of several image processing operations in `Khoros` based on the data requirement of different image processing algorithms. This taxonomy is useful in assisting us to design the optimization algorithm and take advantages of some domain properties. We selected a set of frequently-used image processing operators in `Khoros`, and classified them in Figure 2. We believe that these operations are representative of those in typical image processing algorithms and they form a test suite for our research.

In this taxonomy we classify an operation as *local* or *global.* 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* [6]. Otherwise it is local. A local operation can be either a pixel operation or a masking operation. A pixel operator functions on a pixel-by-pixel basis while a masking operator draws its inputs from a small area in an image which is usually a small $r \times s$ rectangular tile.

If a task employs a pixel operator, it can be trivially shown that $f_i(D_i, Z) = n^2 l\omega/p$, assuming that image data have been partitioned in such a way that each processor operates on the same number of image pixels. l is the number of operations performed on a pixel and n^2 is the image size. There is no need for a processor performing pixel operations to communicate with other processors, as all information resides in the processor itself.

For a masking operator using an $r \times s$ mask, the computation time is proportional to the size of the input image, the size of the mask, the number of operations, the processor speed, and the number of processors used. Thus $f_i(D_i, Z) = n^2 rsl\omega/p + T_{cm}$. The communication cost T_{cm} may not be zero though,

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	$4\alpha+2\beta(\frac{n}{bp})^2(\frac{b(s+r-2)}{2}+\frac{(s-1)(r-1)}{4})$

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

as pixels needed in a masking operation may not be stored in the same processor. Communication costs of a masking operation using different data partitioning schemes are summarized in Table 1 (Details can be found in [17]).

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 should be employed to supply the computation and communication time estimates used in the analysis.

CVIP operations can also be classified as either *data-independent* or *data-dependent*. In a data-independent operation, complexity of a computation depends only on the size of the input image and the 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 [31]. And some worst-case, best-case, or average-case analysis should 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 4.

3.3 Data Redistribution

A task may employ one of the following six data partitioning schemes, which are also provided by the High-Performance Fortran [14]: row, column, row-cyclic, column-cyclic, block, and block-cyclic partitions. These data partitioning schemes are depicted in Figure 3 assuming that the size of the image is $n \times n$ and the number of processors is p.

Since we allow six data partitioning schemes, there are a total of thirty-six 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 thirty nontrivial data redistributions assume non-zero costs.

In [17] we studied and tabulated the cost functions of data redistribution from one partition to another. The cost table is reproduced in Table 2. Note that the communication algorithms used to generate Table 2 may not be optimal and other algorithms for all-to-all communication could be used, e.g., [15, 27]. Our

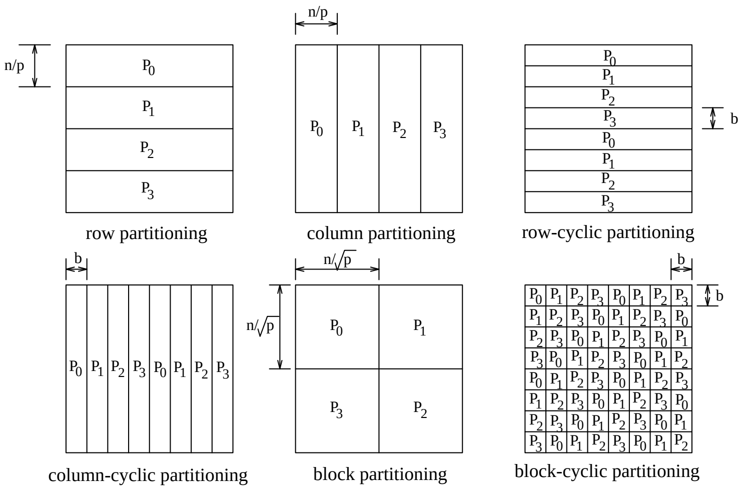


Figure 3: Six types of data partitions of an $n \times n$ image, where b is a blocking factor.

from-to	row	column	row-cyclic	column-cyclic	block	block-cyclic
row	-	DR_2	DR_2	DR_2	DR_1	DR_2
column	DR_2	-	DR_2	DR_2	DR_3	DR_2
row-cyclic	DR_2	DR_2	-	DR_2	DR_2	DR_2
column-cyclic	DR_2	DR_2	DR_2	-	DR_2	DR_2
block	DR_1	DR_3	DR_2	DR_2	-	DR_2
block-cyclic	DR_2	DR_2	DR_2	DR_2	DR_2	-

2D square wrap-around mesh :
$DR_1 = (\alpha + \beta \frac{n^2}{p}/2)(\sqrt{p}-1), DR_2 = (2\alpha + \beta \frac{n^2}{p})(\sqrt{p}-1),$ and $DR_3 = (2\alpha + \beta n^2 \frac{2+\sqrt{p}}{2p})(\sqrt{p}-1)$.
fully connected machine :
$DR_1 = (\alpha + \beta (\frac{n}{p})^2)(p-1), DR_2 = (\alpha + \beta \frac{n^2}{p\sqrt{p}})(\sqrt{p}-1),$ and $DR_3 = (\alpha + \beta \frac{n^2}{p\sqrt{p}})(\sqrt{p}-1)$.

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

purpose here is to demonstrate how redistribution cost functions can be tabulated in advance and later used in the analysis of our scheduling scheme to achieve better efficiency. Another table can be substituted as desired. But for any reasonable redistribution scheme, we observe the following property which will be used in designing our optimization algorithm.

Property: Redistribution triangular inequality. Given three distributions D_{k-1}, D_k and D_{k+1}, we have

$$DR(D_{k-1}, D_k, Z) + DR(D_k, D_{k+1}, Z) \geq DR(D_{k-1}, D_{k+1}, Z).$$

That is because if the above inequality is not true, we can design a new routine for redistribution from D_{k-1} to D_{k+1} as follows. Call a routine for redistributing data from D_{k-1} to D_k followed by a routine for redistributing data from D_k to D_{k+1}. Then the new routine could have a lower cost compared to the one directly redistributing from D_{k-1} and D_{k+1} and the new routine should have been adopted.

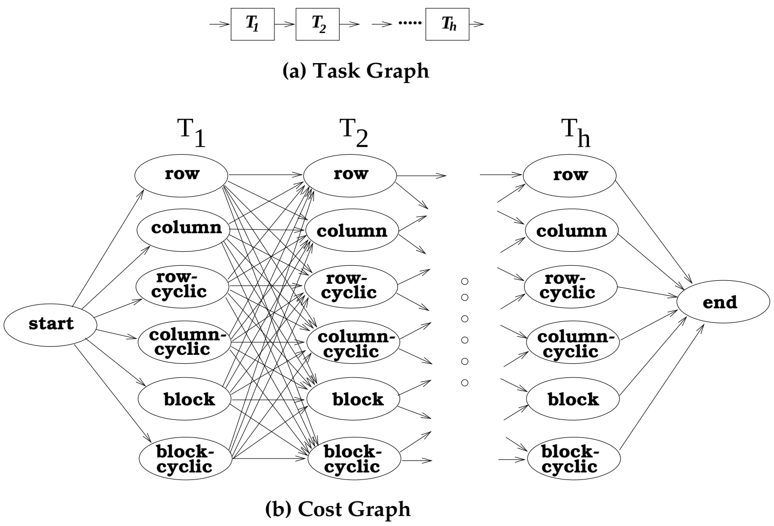


Figure 4: (a) A task graph, and (b) its corresponding cost graph.

4 The Optimization Algorithm

We propose our solution in an incremental fashion. We first discuss the solution for a simple task chain, then we will discuss issues arising from task merging, task chains with loops, and run-time parameter variations. At each step we will analyze the effectiveness and correctness of our strategies. Finally we put all the optimization strategies together in Section 4.6 as a complete algorithm.

4.1 A simple task chain

We first investigate the optimization of a simple task chain $G = (T_1, T_2, \cdots, T_h)$ without loops or data dependent operations. Furthermore, the run-time parameters such as p and n are assumed to be fixed. This is the simplest form of a scheduling problem.

With fixed run-time parameters, each data-independent task has a deterministic cost function based on a given data distribution. To determine the data distribution schemes which minimize the overall parallel time, we construct a graph $CG(G) = (V, E)$, called the *cost graph*, as follow:

- For each possible value of D_i for task T_i, we add a node in V in the CG with a weight $f_i(D_i, Z)$ to represent this particular choice of the data distribution scheme for T_i.
- We then add an edge between every pair of data distribution nodes of two consecutive tasks in the cost graph and assign the edge weight as the data redistribution cost between these two data partitions.

An example is given in Figure 4 where each node in the original task graph is expanded into six nodes (maybe fewer, depending on the implementation in the library) in the corresponding cost graph. The length of the shortest path in the cost graph $CG(G)$ (Figure 4(b)) represents the optimal parallel time. Thus the data distribution scheme of each task can be derived using the scheme represented by the node along the optimal path at that particular stage. Furthermore, because a cost graph is a DAG, the shortest path can be found efficiently in $O(|V|)$ since the number of edges is at most $6|V|$.

4.2 Nodal reduction

Before we augment the above algorithm in finding the best schedule in the presence of loops, we address how the complexity of the above algorithm can be reduced by coalescing adjacent tasks to reduce the size of a cost graph. Note that by merging adjacent tasks into one node, these tasks will now use the same data distribution scheme, which could affect the optimality of the final solution. Hence, we need to examine how node reduction should be done without affecting the optimality.

We identify that if the computation time of a task T_i, is independent of the particular data partition scheme used, i.e.,

$$f_i(D_j, Z) = f_i(D_k, Z),$$

where D_j and D_k stand for any possible data distribution schemes that can be adopted by task T_i, then we can merge this task with its adjacent task without compromising the optimality. for example in Figure 2, a pixel operation satisfies such a condition. This observation is stated more formally in the following lemma.

Lemma 1 *If the computation time of a task is independent of its data distribution, merging the task with its adjacent task in a tree representation of a task chain does not affect the optimality of mapping.*

Proof: Given a tree representation, we can only merge two adjacent leaf task nodes. Without loss of generality, we assume that two leaf nodes are in a subtree G which is $G = (G_1, \cdots, G_{k-2}, G_{k-1}, G_k, G_{k+1}, \cdots, G_h)$. G_{k-1} is T_{k-1} and G_k is T_k. T_k has a computation time independent of the data distribution scheme used. If T_k is merged with its predecessor T_{k-1}, the data partitioning scheme of T_k will be the same as that of T_{k-1}. Thus $G' = (G_1, \cdots, G_{k-2}, T'_{k-1}, G_{k+1}, \cdots, G_h)$. The parallel time $PT(G')$ of the new, reduced graph G' is

$$\begin{aligned} PT(G') \;=\; & \textstyle\sum_{i=1}^{k-2}(PT(G_i) + DR(DX(G_i), DE(G_{i+1}), Z)) \\ & + f'_{k-1}(D_{k-1}, Z) + DR(D_{k-1}, DE(G_{k+1}), Z) \\ & + \textstyle\sum_{i=k+1}^{h-1}(PT(G_i) + DR(DX(G_i), DE(G_{i+1}), Z)) \\ & + PT(G_h, Z), \end{aligned}$$

where

$$f'_{k-1}(D_{k-1}, Z) = f_{k-1}(D_{k-1}, Z) + f_k(D_{k-1}, Z).$$

Note that $f_k(D_{k-1}, Z) = f_k(D_k, Z)$, since the computation time of task T_k is assumed to be independent of the data distribution scheme used.

Let $DE(G_{k+1}) = D_{k+1}$. The difference of the parallel execution time of G and G' is thus

$$\begin{aligned} PT(G) - PT(G') \;=\; & DR(D_{k-1}, D_k, Z) + DR(D_k, D_{k+1}, Z) \\ & - DR(D_{k-1}, D_{k+1}, Z). \end{aligned}$$

Since the data redistribution costs tabulated in Section 3.3 satisfy the triangular inequality,

$$DR(D_{k-1}, D_k, Z) + DR(D_k, D_{k+1}, Z) \geq DR(D_{k-1}, D_{k+1}, Z),$$

we have $PT(G) - PT(G') \geq 0$.

Hence any optimal solution for the original graph can be transformed into one for the reduced graph with a better or equal performance. Thus the optimal solution derived based on a reduced graph is optimal for the original graph. □

4.3 Loop linearization

The strategy discussed in Section 4.1 only works for a task chain without loops. For a task chain with loop structures, the corresponding cost graph contains cycles, and the shortest path algorithm will not work. If loop bounds are known, our approach is to linearize the graph and we propose a *constrained* shortest path algorithm for finding the optimal solution in the linearized graph. We first discuss this method using a single nested loop and then generalize for arbitrary task chains with loops.

4.3.1 Single nested loop

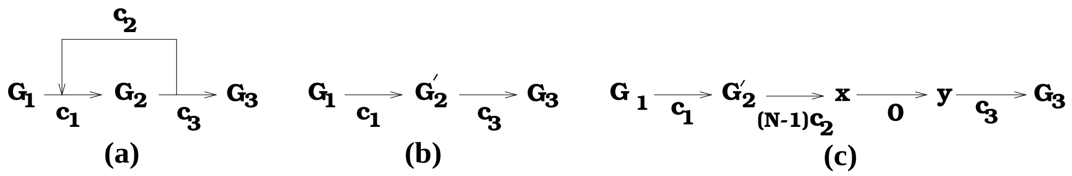


Figure 5: Loop linearization. (a) The original loop, and (b) the resulting linearized graph structure with one node inside the loop, and (c) the resulting linearized graph with multiple nodes inside the loop.

Let a task chain be $G = (G_1, N(G_2), G_3)$ as depicted in Figure 5(a). Let c_1, c_2, and c_3 be the data redistribution costs from G_1 to G_2, from the last operation of G_2 to the first operation of G_2, and from G_2 to G_3, respectively. Then

$$PT(G) \quad = \quad PT(G_1) + c_1 + (N-1)(PT(G_2) + c_2) + PT(G_2) + c_3 + PT_3. \tag{1}$$

There are two cases to be addressed:

- **Case 1.** G_2 is a leaf node (not a compound node).

  Then $c_2 = 0$, Equation 1 is simplified to

  $$PT(G) = PT(G_1) + c_1 + N \times PT(G_2) + c_3 + PT(G_3). \tag{2}$$

  We construct a new graph $G' = (G_1, G_2', G_3)$ as shown in Figure 5(b) by linearizing this loop such that the paths of the linearized graph can model the cost of Equation 1. G_2' is identical to G_2 except that all weights in G_2 are enlarged by a factor of N. It is easy to see that G' has the cost structure as Equation 2. The optimization strategies employed in Section 4.1 can again be used here on the graph shown in Figure 5(b).

- **Case 2.** G_2 contains several tasks or subtask chains.

  Here, the first node and the last node of the loop may assume different data partitioning schemes. Hence c_2 is in general not zero. We can rewrite Equation 1 as

  $$PT(G) = PT(G_1) + c_1 + N \times PT(G_2) + (N-1) \times c_2 + c_3 + PT(G_3). \tag{3}$$

  Thus, we construct a new graph $G' = (G_1, G_2', x, y, G_3)$ as depicted in Figure 5(c). G_2' is identical to G_2 except that all weights in G_2 are enlarged by a factor of N. Two dummy nodes x and y are added after G_2. The redistribution cost from G_2' to x is adjusted to $(N-1)c_2$, that from x to y to zero, and that from y to G_3 to c_3. The node cost of x and y is set to zero.

  In order to make the linearized loop shown in Figure 5(c) having the same cost as Equation 3, two constraints are imposed:

- **CT1:** The data distribution of the first node in G_2' is the same as that for the dummy node x. This is because that c_2 is the data redistribution cost from the last node in G_2 to the first node in G_2, and the data distribution of node x which follows c_2 should be the same as that of the first node in G_2'.
- **CT2:** The data distribution of the second dummy node y is the same as that of the last node in G_2'. This is because c_3 is the data redistribution cost from the last node in G_2 to the first node in G_3.

In this way, the cost of linearized loop satisfies the above equation. For this new graph, we construct its corresponding cost graph as before and search for the shortest path in this cost graph.

Example: We apply our linearization procedure for a task graph with a loop shown in Figure 6. The linearization step produces a cost graph as shown in Figure 6(a). Thus in choosing the shortest path from stage T_1 to T_4, that path mush have the same distribution between T_2 and x according to constraint $CT1$, and between T_3 and y according to $CT2$. □

To find the shortest path in this cost graph with added constraints, we propose a constrained shortest path searching algorithm. The algorithm proceeds as follows:

- We first find the dedicated shortest paths for the nested loop part which starts from the first stage in G_2 and ends at stage y in Figure 5(c). Each dedicated shortest path starts from one distribution $DE(G_2)$ and ends at one distribution of y with constraints $CT1$ and $CT2$. For example in Figure 6(a), the nested loop part is marked with the dash box. There are 6 shortest paths, starting from node 3 or 4 to node 10, 11, or 12. We need two steps to find the paths satisfying the constraints $CT1$ and $CT2$.

    1. First for a given pair $(DE(G_2), DX(G_2))$, we find the the shortest path from $DE(G_2)$ to $DX(G_2)$.
    2. Extend the above path to a distribution of x and y such that $DT(x) = DE(G_2)$ and $DT(y) = DX(G_2)$.
    3. Repeat step 1 for all possible pairs of $(DE(G_2), DX(G_2))$.

- Then we replace the above nested loop part with two stages u and w. The distribution nodes of u include all possible values of $DE(G_2)$ and the distribution nodes of w include all possible $DX(G_2)$. We call this step the *loop kernel compression.* The edges between instances of z and w represent the constrained short paths derived at the above step. The instances of z and w have zero weights.

- Then the shortest path of the new graph satisfies constraints $CT1$ and $CT2$.

Example: For Figure 6(a), the shortest paths from node 3 to 10, 11 and 12 and from node 4 to 10, 11 and 12 are computed. Then the graph transformation procedure will produce a simplified graph as shown in Figure 6(b) where edges from 3' and 4' to 10', 11' and 12' representing the constrained shortest paths derived from T_2 to y. □

4.3.2 A sequence of multiple nested loops

We now generalize our result for any number of nested loops and multiple loops in a task chain. The linearization and searching algorithm proceeds as follows.

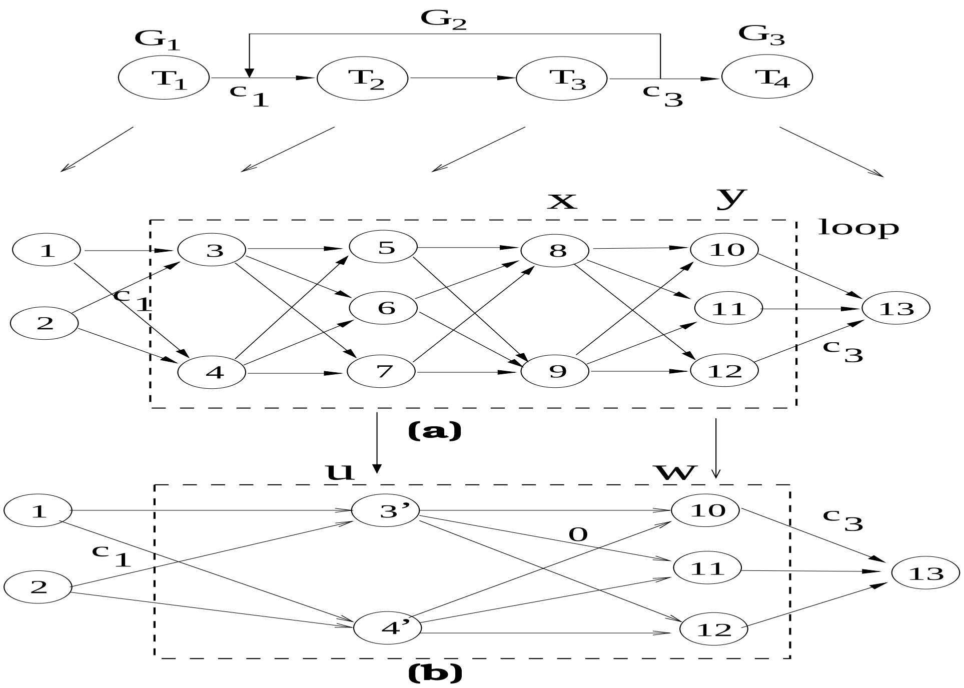


Figure 6: (a) A cost graph with a loop. Nodes 8 and 9 are instances representing x, and nodes 10,11, and 12 are instances of y. (b) The cost graph after merging a loop into two stages for further shortest path search.

- For each multiply-nested loop part, we apply the above linearization and searching procedure in a bottom-up recursive manner.
  - If the inner most loop contains a single task, then we use the linearization procedure depicted Figure 5(b).
  - Else we use the linearization procedure depicted in Figure 5(c). Then we use the kernel compression to further simplify the loop body.

  As a result, for a simple task with a single multiply-nested loop, we will either get one task as shown in Figure 5(b), or obtain a 2-task chain without loop as shown in Figure 6(c) to represent each loop part.

- After each multiply-nested loop part is linearized, we use the classical shortest path algorithm to find the optimal assignment for multiple loop nests.

Complexity analysis. Assume that we have a task chain of V stages with at most r loops. Each loop has at most d nests. Notice that each task has at most 6 instances. Thus without loops, the number of edges in the cost graph is at most $6V$. The shortest path algorithm costs $O(V)$. The complication in dealing with loops comes from the linearization procedure. We can show that the overall complexity is $O(V * d)$.

We examine for one q-nested loop with a total of H tasks, let H_k be the number of tasks which are inside exactly k loop nests. Thus $\sum_{k=1}^{q} H_k = H$. When we apply the bottom up linearization procedure for level q, the kernel transformation costs at most $O(H_k + 2)$. Then the total number of loop levels is reduced to $q - 1$ but the total number of tasks at loop level $q - 1$ increases by no more than 2. After q linearization steps, the cost becomes $\sum_{k=1}^{q} O(H_k + 2) = O(q + H)$.

For a sequence of multiply-nested loops, assume there are r loop parts, each loop part has V_i tasks with $d_i (d_i \leq d)$ level of nests. Then the linearization reduces each loop part into a 2-task chain with a cost of $O(d_i + V_i)$. And the final global shortest path searching costs $O(r)$. Thus the total cost is: $O(r) + \sum_{i=1}^{r} O(d_i + V_i) = O(r + r * d + V)$. Since $r \leq V$, we have $O(d * V)$ as the worst cost.

Correctness analysis. Although the constrained shortest path searching algorithm finds the optimal path within a loop, we need to show that the algorithm also finds the optimal path in nested loops. This is because locally optimal results generally cannot guarantee global optimality. The following lemma establishes the optimality of the constrained shortest path search algorithm for a sequence of multiple loops.

Lemma 2 *Given a task chain with linearized loops, the constrained shortest path searching in the cost graph finds the optimum for the original task chain.*

Proof: We divide our proof into two parts.

Part 1: We first consider a multiply-nested loop G of any level d. Assume $DE(G)$ is chosen as w, and $DX(G)$ is chosen as z in the optimal assignment. We prove by induction on the degree d that there exists the shortest path in the linearized graph between w and z representing the optimal solution for this loop with w and z as the given starting and end distributions.

- $d = 0$. The case is trivial since the task graph does not contain any loop structure.
- $d = 1$: This case was illustrated in the Section 4.3 and Figure 5. It is easily seen that by imposing the additional constraints $CT1$ and $CT2$ on the data distribution patterns of x and y, the cost structure of the linearized loop is the same as that of the original loop.
- Assume the algorithm retains the optimal path when $d = k$ and the innermost k nested loops have been reduced to nodes in a linear sequence. Then we can apply the same loop linearization process as demonstrated in Figure 5(b) to the $k + 1$th nested loops, so that the $k + 1$ nested loops become flattened. This single step will preserve the cost structure as demonstrated in $d = 1$ case.

Part 2: We will show that the the constrained shortest path searching algorithm always finds the optimal assignment for a sequence of nested loops. Assume that the optimal path B in the original graph G in one loop part is shown in Figure `7 (b)`. We show that there exists a path A in the the linearized graph G' with an equal or shorter length.

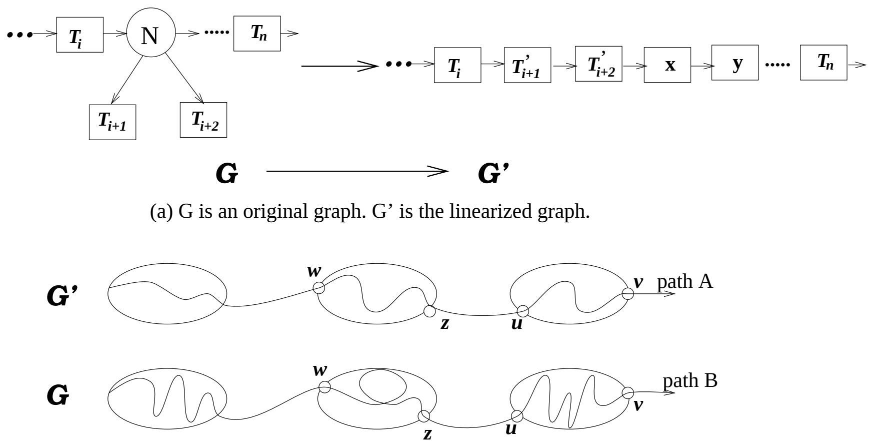
Let PT_{opt} be the parallel execution time of the path B in G. Assume that path B passes through one loop part G_i with $DE(G_i) = w$ and $DX(G_i) = z$.

$$PT_{opt} = \ldots + c + \texttt{path length from w to z} + c' + \ldots.$$

Then according to the Part 1 of the proof, our linearization and searching algorithm retains the optimal path between w and z. Then we can construct a part of path A in G' using the shortest one between w and z in G'. Namely,

$$\texttt{path length from w to z in B} \geq \texttt{path length from w to z in A}.$$

We repeat this construction for all loop parts. A path A is then found in G' such that the length of A is no longer than PT_{opt}. Since our constrained shortest path searching algorithm finds the assignment which is equal or better than the one represented by path A, our algorithm finds the optimal solution for graph G. □



(a) G is an original graph. G' is the linearized graph.

(b) Path B is the optimal path of the original graph, where path A is the result of the algorithm.

Figure 7: (a) G' is the linearized version of graph G, (b) Path B is the optimal path of the original graph G, and path A is the optimal path of G' from the algorithm. Note that each ellipse is subgraph.

4.4 Varying task parameters

So far, the optimal scheduling results are obtained for fixed task parameters (e.g., p and n). If the task parameters are allowed to change, the scheduler may find different optimal paths. For example, the constrained shortest path for input image size of 64×64 could be different from that for image size of 1024×1024 as the computation and communication times are dependent on n. Furthermore, the optimal mapping may change if different numbers of processors are used. However, it is not advisable to produce different parallel codes for different parameters as there may 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 sample the possible parameter values and generate one parallel code which provides moderate performance degradation over the sampled range of the parameter values. We call such a mapping a *minimal performance degradation mapping* (MPDM).

Let R be all possible values of Z and let $m = |R|$. Let $OPT(G, Z)$ be the parallel time of the optimal schedule with respect to one particular parameter value Z. Let $GOPT(G, Z)$ be the parallel time of the MPDM derived with respect to Z. Then a MPDM minimizes the metric mean performance degradation (MPD):

$$MPD = \frac{1}{m} \sum_{Z \in R} |\frac{PT(G, Z)}{OPT(G, Z)} - 1|. \tag{4}$$

For example, if images may assume three different sizes, say 256×256, 512×512, and 1024×1024, then $m = 3$. The optimal mapping may be different from one case to another. Our algorithm will derive one mapping for all three cases and the summation of the performance degradation from the optimum for each case should be minimum.

Given a simple task chain without loops, a brute force method to compute such a mapping is to try all possible paths in the m cost graph variances. However, the cost can be as high as $O(6^V * m)$, where V is the number of tasks in the cost graph. In stead, we propose the following algorithm to compute a mapping with the minimum MPD, which is much more efficient.

- For each parameter setting Z, we find the optimal solution $OPT(G,Z)$.
- We construct a new cost graph. Each instance of a task T_i with distribution D_i uses the weight of

$$\sum_{Z\in R}\frac{f_i(D_i,Z)}{OPT(G,Z)}.$$

The weight of the edge from D_i to D_{i+1} is

$$\sum_{Z\in R}\frac{DR(D_i,D_{i+1},Z)}{OPT(G,Z)}.$$

This graph is called the *augmented cost graph.* Then the shortest path in this graph finds the optimal mapping with the minimum MPD.

The above algorithm has a complexity $O(Vm)$. We demonstrate the optimality of the algorithm in the following lemma.

Lemma 3 *Given a task chain without loops, the shortest path in the augmented cost graph minimizes the MPD with respect to a range of pre-sampled parameter settings.*

Proof: To minimize MPD, we have to minimize $\sum_{Z\in R}\frac{PT(G,Z)}{OPT(G,Z)}$. Let $G=(T_1,T_2,\cdots T_h)$.

$$\begin{aligned}\sum_{Z\in R}\frac{PT(G,Z)}{OPT(G,Z)} &= \sum_{Z\in R}\frac{f_1(D_1,Z)+DR(D_1,D_2,Z)+\ldots+DR(D_{h-1},D_h,Z)+f_h(D_h,Z)}{OPT(G,Z)}\\ &= \sum_{Z\in R}\frac{f_1(D_1,Z)}{OPT(G,Z)}+\sum_{Z\in R}\frac{DR(D_1,D_2,Z)}{OPT(G,Z)}+\ldots+\\ &\quad+\sum_{Z\in R}\frac{DR(D_{h-1},D_h,Z)}{OPT(G,Z)}+\sum_{Z\in R}\frac{f_h(D_h,Z)}{OPT(G,Z)}.\end{aligned}$$

Since each term in the above equation corresponds to a node or edge weight in the augmented cost graph, we can minimize $\sum_{Z\in R}\frac{PT(G,Z)}{OPT(G,Z)}$ by finding the shortest path in the augmented cost graph. □

For task chains with loops, we can generalize our result if the loop iteration numbers remain constant. That is done in Section 4.6.

4.5 Decomposing into data-independent subgraphs

The above analysis has assumed that the cost of a task is data independent. If the costs are data dependent, or the loop bounds are unknown, it is difficult to predict the performance and obtain a good schedule at the compile time.

In general, our strategy is to apply our algorithm to optimize subgraphs in which task weights and loop bounds are deterministic. Hence, the run-time data dependent parts are isolated and the maximum data-independent subgraphs are identified to which our optimization algorithm applies.

Given a graph $G=(G_1,\cdots,G_h)$, we search the corresponding tree structure in a bottom-up manner to identify the data-dependent regions. If G_i contains a data-dependent task or a loop with an unknown loop count, we mark G_i accordingly. We apply the optimization strategy to each maximum subgraph sequence $G_i,\ldots,G_j$ such that all these subgraphs contain only data-independent operations and loops with known loop bounds. Note that such an optimization strategy will in general fail to locate a globally optimal solution — as such a solution cannot be found without knowledge of the actual execution pattern.

4.6 The mapping algorithm

Now we are ready to put all the above optimization strategies together and present our scheduling algorithm as follows.

- **Step 1:** Traverse the tree structure of the task chain to identify a set of maximal subgraph chains which contain only data-independent tasks and known loop bounds. For each of these subchains, we apply the optimization technique described from Step 2 to Step 5.
- **Step 2:** Coalesce pixel operations into adjacent tasks to reduce the number of tasks.
- **Step 3:** For each parameter setting, traverse the tree structure of each chain in a bottom-up manner to linearize loops, which results in a flattened task chain without loops. Construct a cost graph and derive the optimal data assignment using the constrained path search.
- **Step 4:** For each task in the reduced graph, reset its task weight with respect to distribution D_i as

  $$\sum_{Z \in R} \frac{f_i(D_i, Z)}{OPT(G, Z)}.$$

  The weight of the edge from D_i to D_{i+1} becomes

  $$\sum_{Z \in R} \frac{DR(D_i, D_{i+1}, Z)}{OPT(G, Z)}.$$

  Then use the linearization procedure for this chain with new weights and derive the optimal assignment. This assignment has the minimum MPD.

Notice that if loop bounds vary from one parameter setting to another, the above algorithm cannot find the optimum. Otherwise the following theorem asserts that the above algorithm will find the best assignment.

Theorem 1 *For each maximum data-independent subgraph chain with loops, the above algorithm finds an optimal schedule that minimizes the MPD with respect to a range of pre-sampled parameter settings.*

Proof : The proof is essentially a combination of the results of Lemmas 1, 2, and 3.

We need first to show that for each parameter setting Z, we find the optimal solution $OPT(G, Z)$. By Lemma 1, Step 2 does not diminish the optimality of the scheduling result. Furthermore, Lemma 2 shows that the constrained shortest path searching in a cost graph with linearized loops finds the optimum. Thus Steps 2 and 3 find the optimum for a given Z.

For Step 4, the linearization procedure is applied to the augmented task chains. For a simple task chain without loop, Lemma 3 shows the algorithm is optimal. For a task chain with loops, we just need to show that the optimal solution for the chain with updated weights minimizes the MPD function. The way is similar the one used for proving Lemma 1. Since we assume that the loop bound is a constant, we can expand the term $PT(G, Z)$ in $\sum_{Z \in R} \frac{PT(G,Z)}{OPT(G,Z)}$ using the parallel time definition given in Section 3. □

5 Experimental Results

We have implemented the partitioning and scheduling scheme and parallel versions of several CVIP algorithms on a MEIKO CS-2 parallel processor at UCSB. MEIKO CS-2 is a distributed memory parallel

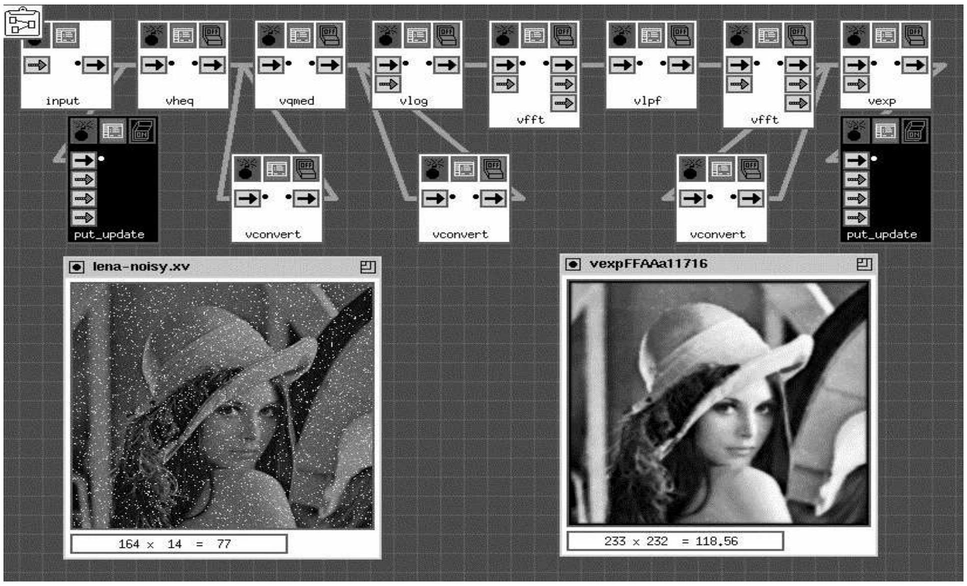


Figure 8: An image processing operation constructed using `Khoros`. The image shown at bottom left is the input image, and The image shown at bottom right is the output image from the operation.

machine. Each node has a SUN SPARC Viking processor for computation and a Sparc-like co-processor for communication. The structure of the CS-2 data communication network is a multi-stage packet-switching fat-tree using a wormhole routing scheme. There are several message passing libraries available and the one we used is NX/2 (the Intel communication interface). And the NX/2 version ScaLAPACK [8] was employed as a parallel library.

For system parameters, we estimated $\alpha = 80\mu sec$, $\beta = 0.026\mu sec/byte$, and $\omega = 0.33\mu sec/flop$ using the NX/2 communication package on the MEIKO CS-2. The blocking factor, b, was selected to be eight so that the ratio of computation speed and communication speed is about one [10] and parallelism can be exploited without excessive communication overhead.

5.1 Image Enhancement and Noise Reduction

Problem description We tested the scheduling scheme on a popular image processing operation, homomorphic filtering [11], 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 operation used histogram equalization to stretch the image intensities back to the original range, followed by median filtering to remove the "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 low-pass filtering operation in the frequency domain (through Fourier and inverse Fourier transforms). Finally, the image was scaled back using an *exponent* operator.

A visual implementation of the above algorithm using `Khoros` is shown in Figure 8. A task graph of this operation is shown at Figure 9(a), where the input/output routines used in `Khoros` were omitted but various type conversion operators (denoted by C) were retained. Figure 9(b) shows the reduced task graph (after node merging) of the operation. Since 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 filter 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.

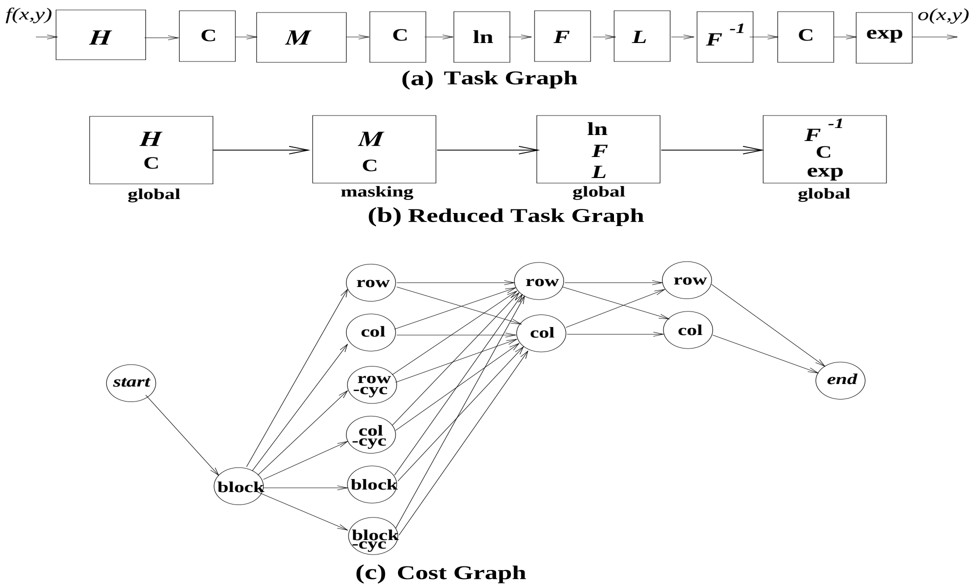


Figure 9: (a) The task graph for the noisy-reduction filter, (b) the reduced task graph, and (c) the cost graph.

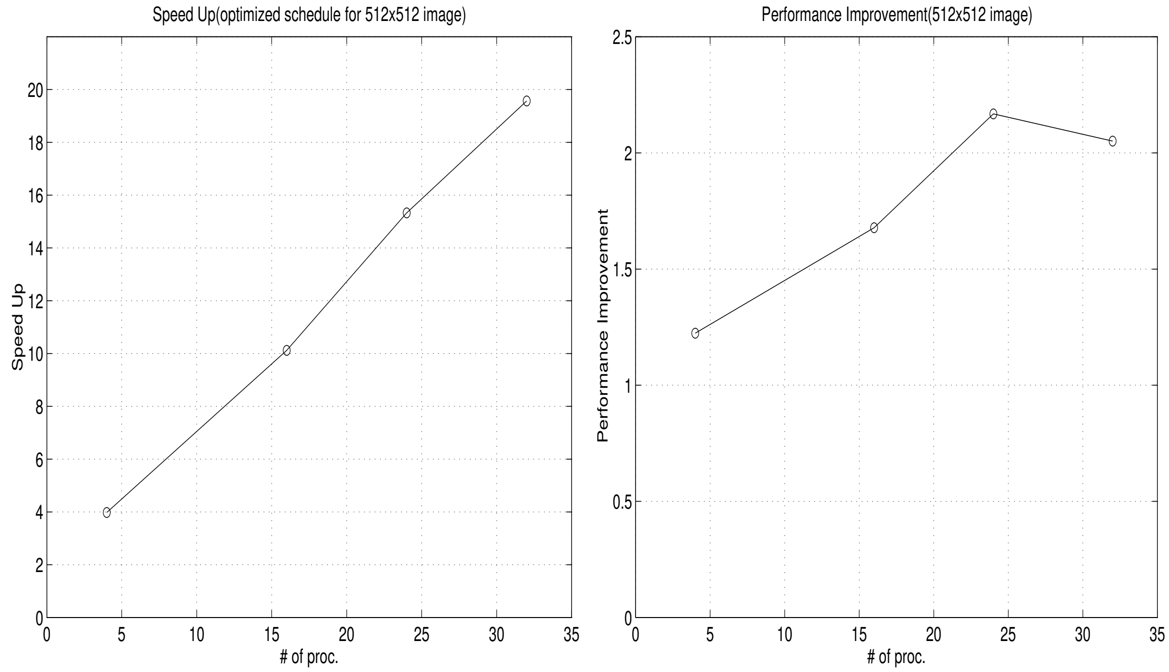
Table 3: The parallel execution time (in sec.) of the optimized code for images of size 512x512.

	HIST	DR	MED	DR	FFT	DR	IFFT	TOTAL
1 proc	0.12		1.91		5.78		6.63	14.44
4 proc	0.05	0.35	0.50		1.32		1.41	3.62
16 proc	0.03		0.14	0.55	0.34		0.37	1.43
24 proc	0.01		0.06	0.43	0.21		0.23	0.94
32 proc	0.01	0.39	0.04		0.15		0.16	0.74

HIST:histogram equalization, DR:data redistribution, MED:median filter, FFT:fast Fourier transformation, IFFT:inverse FFT

Figure 9(c) shows the cost graph created from the reduced task graph in Figure 9(b). Edges incident on the starting and the end nodes were of zero weight. Costs of all other edges were tabulated in Table 2. The first node in this reduced graph represented a histogram equalization operation, which had only a single parallel implementation based on the block partition in our library. The next node was a masking operation; hence, the node was expanded into six nodes in the cost graph using six different data partitioning schemes. The Fourier transform assumed two different implementations, both of which used the transpose algorithm for the parallel 1D FFT operation. For 2D FFT [11], one library routine used the row partition for 1D FFT, followed by a transpose operation, and then by another row-wise 1D FFT. The other routine was similar but assumed the 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 used $r = s = 3$. The image size was 512×512, and the number of processors used were 4, 16, 24, and 32. For example, using the above parameters and $p = 4$, the shortest path in Figure 9 was found to pass through the block 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 parallel time was found to be $3.62sec$. Table 3 shows the breakdown of the parallel execution time of the optimized codes using 4, 16, 24, and 32 processors. Note that some DR columns, which show the data redistribution time in between two adjacent stages, are



(a) (b)

Figure 10: (a) The speedup of the optimized parallel program over the sequential program on MEIKO CS-2. (b) The performance improvement of the optimized codes over unoptimized codes on MEIKO CS-2.

empty because there was no data redistribution in between the stages.

Figure 10(a) shows the speedup achieved by parallel implementations. As the sequential execution time was $14.44sec$, we estimated the speedup factor SP to be 3.99, 10.12, 15.32, and 19.57 for image size of 512×512 using 4, 16, 24, and 32 processors, respectively. Figure 10 shows that the parallel performance is scalable.

Effectiveness of optimization A scheduler is essential in achieving a reasonable speed up, as unoptimized codes can drastically reduce the efficiency of a parallel execution. We define the performance improvement as the average of

$$\frac{\texttt{parallel time of an unoptimized code}}{\texttt{parallel time of the optimized code}}.$$

Figure 10(b) shows the performance improvement using different numbers of processors for image size of 512×512. Our analysis for $n = 512$ and $p = 24$ shows the performance improvement could be as high as 2.05, i.e., the optimal schedule generated a parallel code whose parallel execution time was more than twice faster than that of the unoptimized one. The savings arise mostly from reducing unnecessary data redistribution in between stages.

MPDM Next, we show how our algorithm handles variation of parameters using the MPDM algorithm. To compute the MPDM, 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 cost graph. We used the MEIKO CS-2 architecture with $b = 8$, $r = s = 3$, $n = 512$, and used both four and sixteen processors. The optimal parallel times for $p = 4$, 16

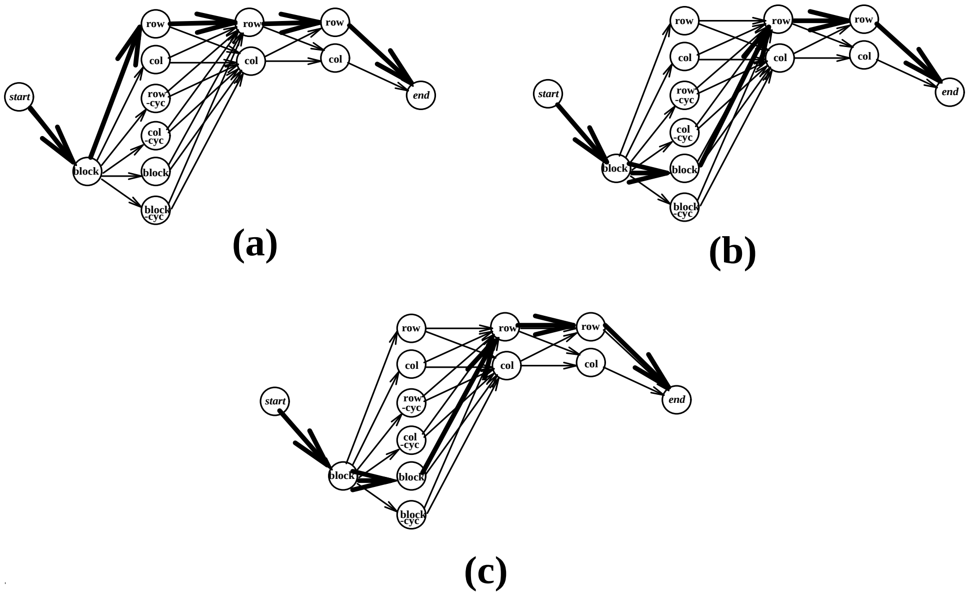


Figure 11: The optimal paths shown as the heavier lines on MEIKO CS-2 with $r = s = 3, b = 8$, and $n = 512$. (a) $p = 4$, (b) $p = 16$, (c) the minimal performance degradation path for $p = 4$ and 16.

were $3.62sec$ and $1.42sec$, respectively. Their corresponding optimal paths are shown in Figures 11(a) and (b). Based on them, we generated a new cost graph (Figure 11(c)) by aggregating the edge and node costs. If p can be either 4 or 16, our algorithm obtained the MPDM path shown in Figure 11(c). In this case the MPD was

$$\begin{aligned} MPD &= \tfrac{1}{2}(|\tfrac{3.98sec}{3.62sec} - 1| + |\tfrac{14.22sec}{14.22sec} - 1|) \\ &= 0.049. \end{aligned} \tag{5}$$

The given example shows that a schedule was found which had a 4.9% performance degradation over two different numbers of processors used. In other words, a single parallel code generated by the algorithm can be executed with two different parameter sets with a slight performance degradation. The MPD thus provides a good measure to determine the performance degradation of a parallel code over different parameters. For a pre-defined set of parameters, one parallel code may be sufficient if the MPD is small. However, if the MPD is too large, then the parameter set should be divided into smaller subsets such that the MPD of the schedule in each parameter subset is less than a preset threshold.

5.2 Face Recognition using Eigen Images

Problem description A second example was on parallelizing a high-level vision recognition algorithm based on the principal component analysis, which was first proposed in [21] and later made popular by [28]. The basic idea is to treat an image as a point in a very high dimensional space. Then for a collection of images, the principal components of the image ensemble can be found by computing the eigenvectors of the covariance matrix of the set of images. These eigenvectors (or eigen images) then serve as a compact basis set to represent the image ensemble. In [28], this eigen image paradigm was applied for human face recognition, and is fast gaining popularity.

More specifically, an image $I(x, y)$ is usually treated as a two-dimensional matrix of dimension, say, $n \times n$. But the same image may also be considered as a one-dimensional vector of dimension n^2 (result of a row-by-row or a column-by-column scan). So a typical image of size 256×256 becomes a vector of dimension $65,536$, or equivalently, a point in a $65,536$-dimensional space.

For a collection of linearly scanned images, $\Gamma_1, \Gamma_2, \ldots, \Gamma_M$, where M is the total number of images, the average image of the set is defined as $\Psi = \frac{1}{M}\sum_{i=1}^{M} \Gamma_i$, and each image differs from the average by the vector $\Phi_i = \Gamma_i - \Psi$. If we assemble the set of Φ_i images into a matrix $A = \{\mathbf{\Phi}_1, \mathbf{\Phi}_2, \ldots, \mathbf{\Phi}_M\}$ of size $n^2 \times M$, and subject the resulting matrix to the principal component analysis, which seeks a set of M orthonormal vectors, $\mathbf{u}_M$, that best describe the distribution of the data in the image ensemble, then the kth eigenvector (eigen image), $\mathbf{u}_k$, is chosen such that

$$\lambda_k = \frac{1}{M}\sum_{i=1}^{M}(\mathbf{u}_k^T \Phi_i)^2$$

is a maximum subject to

$$\mathbf{u}_l^T \mathbf{u}_k = \delta_{lk} = \begin{cases} 1, & \texttt{if } l = k \\ 0, & \texttt{otherwise}. \end{cases}$$

The vectors $\mathbf{u_k}$ and scalars λ_k are the eigenvectors and eigenvalues, respectively, of the covariance matrix

$$C = AA^T .$$

The matrix C is of a dimension n^2 by n^2, and determining the n^2 eigenvectors and eigenvalues of C is an intractable task for typical image sizes. We need a computationally feasible method to find these eigenvectors. [28] suggested computing the eigenvectors and eigenvalues of $A^T A$ instead, which is of a much smaller dimension of $M \times M$. And it can be trivially shown that the eigenvectors of AA^T and those of $A^T A$ are related by a multiplication by A.

Following this analysis, we construct the $M \times M$ matrix $L = A^T A$, where $L_{ij} = \mathbf{\Phi}_i^T \mathbf{\Phi}_j$ and find the M eigenvectors, $\mathbf{v}_l$, $i = 0, \cdots, M-1$ of L. The linear combinations of these vectors and the M input images form the eigen images $\mathbf{u}_l$,

$$\mathbf{u}_l = \sum_{k=1}^{M} \mathbf{v}_{l_k} \mathbf{\Phi}_k, \quad l = 1, \ldots, M.$$

For face recognition applications, a sample image set is first used to compute the eigen basis images (or eigenfaces). Then to classify a face image using eigenfaces, a new face image (Γ) is decomposed into its eigenface components as

$$\omega_k = \mathbf{u}_k^T(\Gamma - \Psi)$$

for $k = 1, \ldots, M'$. These coefficients form a vector $\Omega^T = [\omega_1, \ldots, \omega_{M'}]$ that describes the contribution of each eigenface in representing the input face image. Note that M', $M' \leq M$, is the number of dominant eigenfaces selected among the M eigenfaces. The vector may then be used as a feature vector in a standard pattern recognition formulation to determine its membership in the predefined face classes.

The scheduling algorithm was used to generate a parallel version of this face recognition algorithm. We used the same image data set as that in [28]. The visual representation of the algorithm using `Khoros 2.0.2` is shown in Figure 12. The corresponding task graph of this operation is shown in Figure 13(a). Again, input/output routines were omitted in the task graph.

The first loop in Figure 13(a) is for computing the average image (Ψ) of the input image ensemble. The second loop is for rasterizing individual images to form an image ensemble matrix $\mathbf{A}$ of size $n^2 \times M$. **TRANS** routine generates the transpose of $\mathbf{A}$. $\mathbf{A^T A}$ is then computed to generate a matrix of size $M \times M$ and the eigen analysis is performed on the resulting matrix. Finally, the third loop is for generating the M eigen images ($\mathbf{u}$).

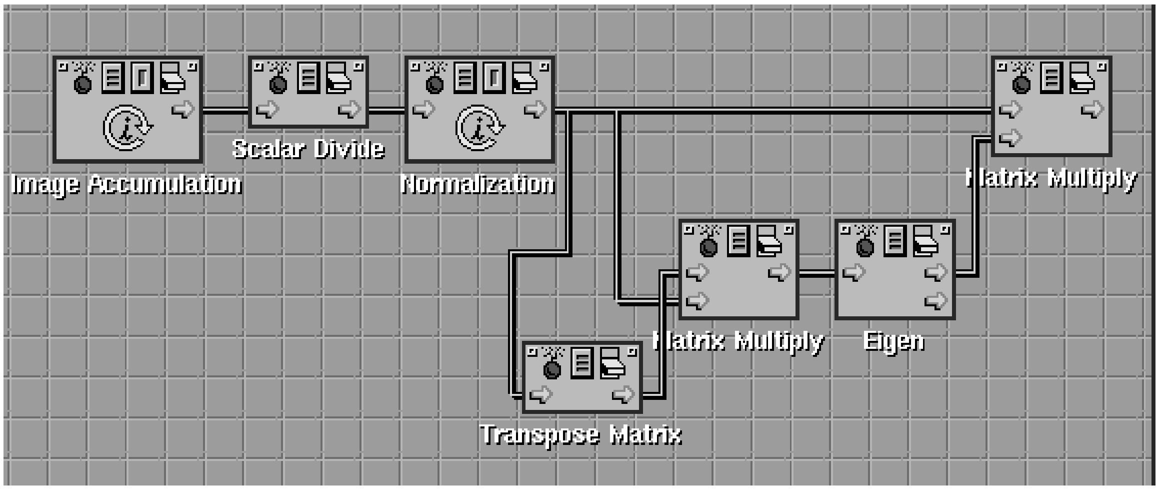


Figure 12: Implementation of the eigenface recognition algorithm using `Khoros`.

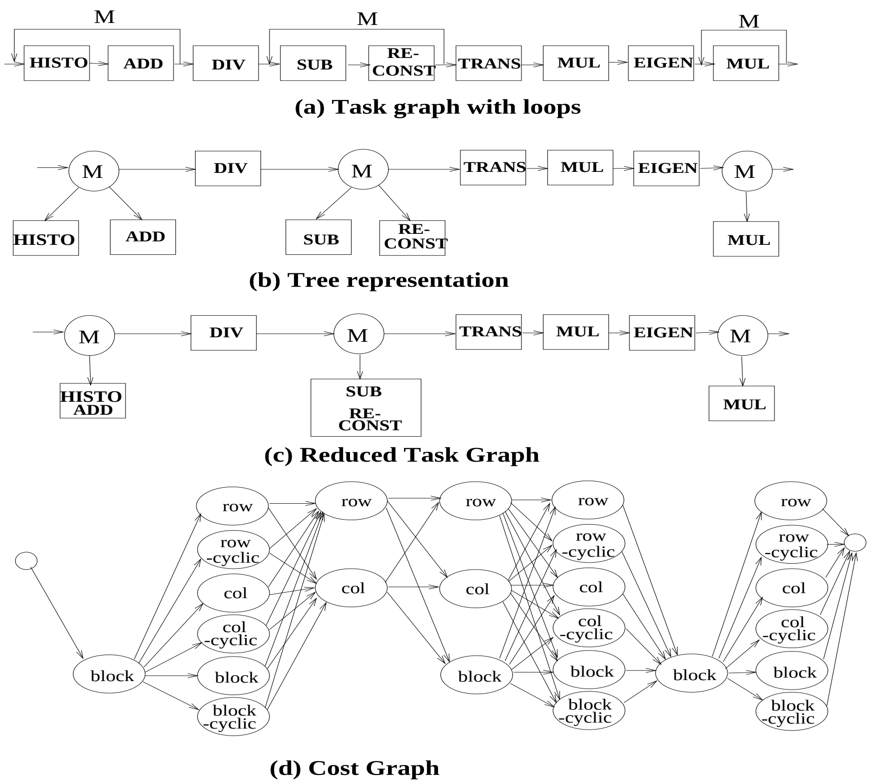
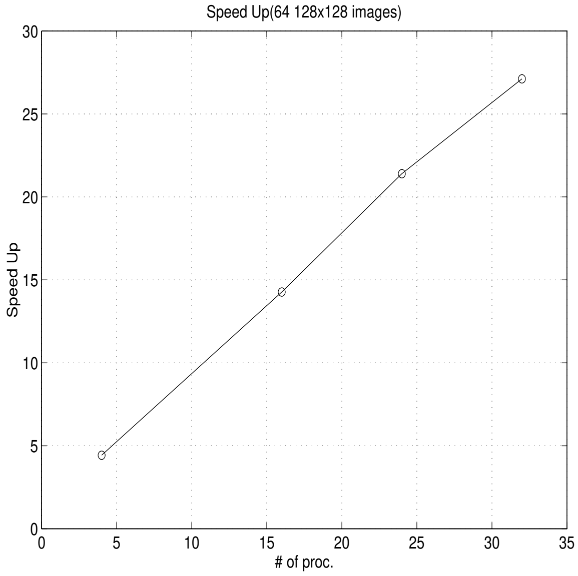
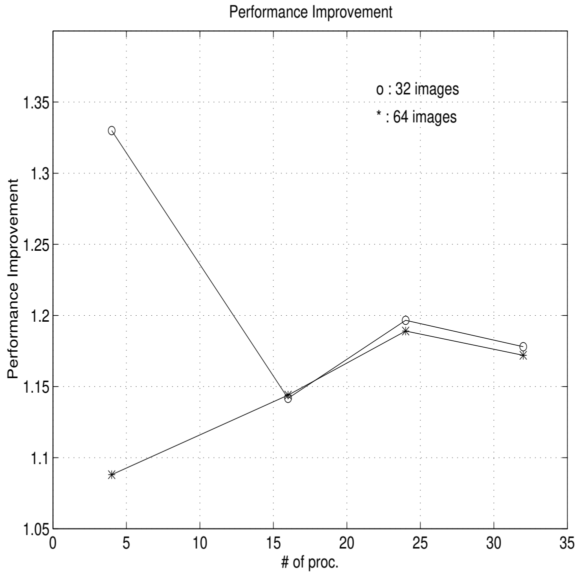


Figure 13: Graph representations of the eigen image computation.



(a)



(b)

Figure 14: (a) The speedup of the parallel eigenface computation on MEIKO CS-2 using 64 128×128 face images. (b) The performance improvement of the parallel eigenface computation using 32 64×64 face images and 64 128×128 face images on MEIKO CS-2. $p = 4, 16, 24$ and 32.

Figure 13(b) shows the tree representation of the task graph where each loop becomes a tree rooted at that particular processing node. Pixel operations were then merged into the adjacent operations, and the reduced task graph is shown in Figure 13(c). The histogram operation (HISTO) had only a single implementation based on the block partition. The reconstruction routine (RECONST) had both row-wise and column-wise implementations. The matrix transpose routine (TRANS) had three implementations based on row, column, and block partitions in the parallel algorithm library. Hence, the scheduling graph based on the available library implementations is shown in Figure 13.

In our experiment, the eigenvalue and eigenvector computation was done sequentially because we were not able to make the parallel routines in ScaLAPACK to work properly. When M is small, sequential eigen computation of an $M \times M$ matrix is fast enough to achieve an acceptable performance. Thus the eigen computation comprised three stages. The first and the last stages were for gathering and scattering data from and to the previous and the next stages. The middle stage was for computing eigenvalues and eigenvectors.

Overall performance The image ensemble comprised a set of 32 face images of size 64×64 and another set of 64 face images of size 128×128. The number of processors used were 4, 16, 24, and 32. The shortest path for $p = 4$, $M = 64$, and $n = 128$ (Figure 15(a)) was found to pass through the block partition node for the first loop, the column partition node for the scalar division, the column partition node for the second loop, the column partition node for the matrix transpose, the column partition node for the matrix multiplication, the block partition node for the eigen analysis, and the block partition node for the third loop.

Figure 14(a) shows that the speedup of the large data set is more significant than that of the small data set. The speedup was as high as 27.11 when $p = 32, M = 64$, and $n = 128$. This figure shows that the parallel eigenface computation is scalable as the number of processors increases. From this figure, we can conjecture that near-linear speedup can be achieved as the input data size increases. However, the large

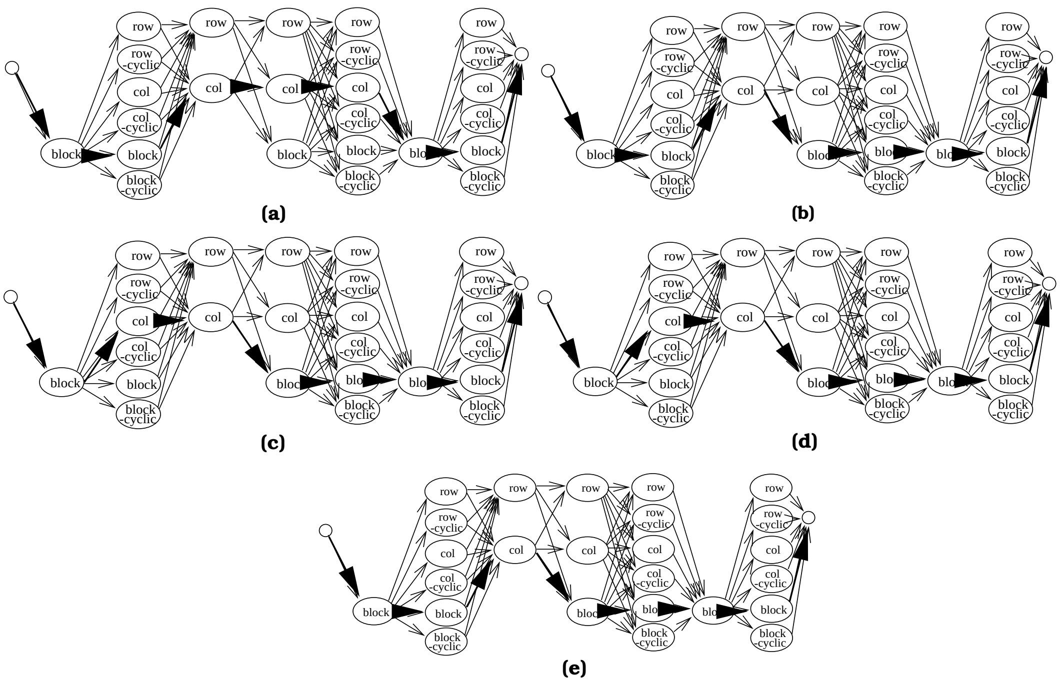


Figure 15: The optimal paths on MEIKO CS-2 with 64 images. (a) $p = 4$, (b) $p = 16$, (c) $p = 24$, (d)$p = 32$, (e) the minimal performance degradation path for $p = 4, 16, 24$ and 32.

amount of communication needed in the data redistribution routines and the sequential eigen computation might keep the parallel program from achieving even better performance.

Effectiveness of optimization Figure 14(b) depicts the average performance improvement of the optimized code over an unoptimized one. The best performance improvement in this figure is 1.33, where $p = 4, n = 64$, and $M = 32$. In this specific application, if the input data size is large, the parallel execution becomes computation bound. If the input data size is small, then the communication time dominates the parallel execution. Since our algorithm minimizes the data exchange costs, it achieves a better performance improvement for communication bound tasks as illustrated in Figure 14(b). To summarize, our algorithm generated parallel codes with a significant performance improvement for the small data set and achieved a near-linear speedup for the large data set.

MPDS The MPDS for $M = 64, n = 128$ and $p = 4, 16, 24$, and 32 is shown in Figure 15. The optimal parallel times for $p = 4, 16, 24$, and 32 were $27.22sec, 8.44sec, 5.63sec$ and $4.45sec$, respectively. Their corresponding optimal paths are shown in Figure 15(a)-(d). The augmented scheduling graph (Figure 15(e)) was generated by aggregating the edge and node costs. The constrained shortest path algorithm found the MPDS over $p = 4, 16, 24$ and 32 with an MPD

$$\begin{aligned} MPD &= \tfrac{1}{4}(|\tfrac{28.13sec}{27.22sec} - 1| + |\tfrac{8.44sec}{8.44sec} - 1| + |\tfrac{5.84sec}{5.63sec} - 1| + |\tfrac{4.72sec}{4.45sec} - 1|) \\ &= 0.032. \end{aligned} \quad (6)$$

The MPDS had a 3.2% performance degradation over the four different parallel runs using 4, 16, 24, and 32 processors. In this case the MPD is small enough to be tolerable, and one parallel code can be used over the pre-defined set of parameters.

6 The Concluding Remarks

The main contribution of this work is the development of a mapping scheme for automating the selection of data distribution in integrating a sequence of parallel library routines modeled by a task chain. The proposed algorithm optimizes performance in dealing with loop structures and run-time parameter variation. The experiment results with image processing operations indicate our scheme provides a good automation in mapping those applications. There are still a number of issues to be addressed in future. For example, handle more complicated loop conditions and application graph structures with a large set of data-dependent computation. Another future work is to incorporate our scheme in a visual programming environment such as Khoros with standard parallel libraries MPI and ScaLAPACK.

References

[1] E. Anderson, *et al., LAPACK Users' Guide, Second Edition,* SIAM, Philadelphia, PA, 1995.

[2] H. M. Alnuweiri and V. Prasanna, *Parallel Architectures and Algorithms for Image Component Labeling,* IEEE Trans. Pattern Analysis and Machine Intelligence, Vol. 14, No. 10, Oct. 1992, pp.1014-1992.

[3] D. Banerjee and J. C. Browne Complete Parallelization of Computations: Integration of data partitioning and functional parallelism for dynamic data structures, Proc. of IPPS' 96, IEEE, Hawaii, pp. 354-360.

[4] A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek, V.S. Sunderam, Graphical development tools for network-based concurrent supercomputing, *Proc. of Supercomputing '91*, IEEE, 1991, pp.435-444.

[5] A. L. Beguelin, *et al., Solving Computational Grand Challenges using a Network of Heterogeneous Supercomputers,* Proceedings of Fifth SIAM Conference on Parallel Processing, Philadelphia, 1992, pp. 596-601.

[6] 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

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

[8] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, *ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers,* Proceedings of the 4th Symp. on Massively Parallel Computing, 1992, pp.120-127.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] C. Lee, Y. F. Wang, and T. Yang, *Static Global Scheduling for Optimal Computer Vision and Image Processing Operations on Distributed-Memory Multiprocessors*, Technical Report TRCS94-23, University of California, Santa Barbara, CA, 1994.

[18] C. Lee, Y. F. Wang, and T. Yang, *Static Global Scheduling for Optimal Computer Vision and Image Processing Operations on Distributed-Memory Multiprocessors,* 6th International Conference on Computer Analysis of Images and Patterns, Prague, Czeck, Sept. 6-8, 1995, pp. 920-925.

[19] C. Lee, Y. F. Wang, and T. Yang, *Partitioning and Scheduling for Parallel Image Processing Operations,* Seventh IEEE Symposium on Parllel and Distributed Processing, San Antonio, Texas, Oct. 25-28, 1995, pp. 86-90.

[20] The MPI Forum, *MPI:A Message Passing Interface,* Proceedings of Supercomputing '93, Portland, Or., pp.878-883.

[21] H. Murakami and B. V. K. Kumar, *Efficient Calculation of Primary Images from a Set of Images,* IEEE T-PAMI, Vol. 4, No. 5, Sep. 1982, pp. 511-515.

[22] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, *A convex programming approach for exploiting data and functional parallelism.* In Proc. of 1994 Inter. Conf. on Parallel Processing, Vol. 2, pp. 116-125.

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

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

[25] 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.

[26] G. Srellner, S. Lamberts, and T. Ludwig, *NXLib Users' Guide,* Institut fur Informatik, Technische Universitat Munchen, Munchen, German, Mar. 1994.

[27] R. Thakur and A. Choudhary, *All-to-all communication on meshes with wormhole routing,* Proceeding of 1994 Inter. Parallel Processing Symposium, pp. 561-565.

[28] M. Turk and A. Pentland, *Eigenfaces for Recognition,* Journal of Cognitive Neuroscience, Vol. 3, No. 1, 1991, pp.71-86.

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

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

[31] 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.

[32] 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.