Practical Considerations for Executing Vision Algorithms On Parallel Arrays

Garrison W. Greenwood
University of Washington

Abstract

Many vision algorithms have been proposed as being suitable for mapping onto parallel arrays. All of these proposals have claims of expected performance. But the actual performance can be less than that claimed if certain factors are not taken into consideration. This paper examines some mappings of vision algorithms onto parallel arrays and considers various issues that affect the algorithm's performance.

1: Introduction

An algorithm may be viewed as a problem graph where vertices represent tasks or computations and edges represent message flows. When vision algorithms are represented in this form, it becomes apparent that some of the calculations can be performed concurrently. By assigning these concurrent tasks to different processing elements (PES) in some parallel processing machine, it is possible to execute the algorithm in less time than that possible on a single processor.

Deciding whether or not to run such an algorithm on a parallel processing system can be determined by answering three questions:

1) What does it cost to use the parallel array?
2) How fast will the algorithm run on the parallel array?
3) What is involved in porting the algorithm to the parallel array?

The first question deals with financial costs (e.g., rental) and will not be discussed. This paper considers issues that influence the answers to the last two questions. The paper is organized as follows. In this section we review parallel array topologies suitable for vision algorithms. Section 2 is the main section where algorithm porting and speed issues are discussed. Section 3 provides a summary.

Figure 1 SIMD and MIMD Systems

1.1: A Taxonomy of Parallel Arrays

Parallel processing arrays are composed of many processing elements (PES) and an interconnection network to provide communication paths between PES. Each PE consists of a processing unit capable of performing logical and arithmetic functions, a communications unit to allow communication with other PES, and some local memory.

Most parallel processing arrays fit into one of
two classifications [Flyn66]. The single instruction stream-multiple data stream (SIMD) class has instructions broadcast to each PE from a central control unit; every active PE executes the same instruction in a lockstep manner on its own data set. In the multiple instruction stream-single data stream (MIMD) class each PE has its own dedicated instruction stream and data stream.

Deciding which classification to use is not easy as each has certain advantages. SIMD arrays have a single program that is resident in the host while the MIMD has multiple interacting programs that reside in each PE. Because of the lower memory requirements, SIMD systems can have physically smaller PEs which makes them attractive for massively parallel systems. Additionally, any overhead associated with data synchronization is less with SIMD arrays as each PE operates in a lockstep manner. On the other hand, MIMD arrays are more flexible since they possess no constraints on operations that can be performed concurrently. They are also considerably more efficient in executing data dependent instruction sequences. Siegel, et. al, has recently published any excellent article contrasting SIMD and MIMD arrays used in vision applications [Sieg92].

Figure 2 shows several topologies ideally suited for executing vision algorithms. Several of these arrays are available on the commercial market [Alma89][Ston90].

2: Discussion

Suppose that we are given a vision algorithm represented as an acyclic digraph G which consisting of a node set $V_G$ and an edge set $E_G$. We refer to this digraph as a dependency graph (DG). $V_G$ is the set of tasks involving one or more computations and $E_G$ is the set of communications links used to pass results between tasks. The directions of the edges depicts a natural precedence among tasks. Similarly, let $H$ be a non-shared memory parallel array where $V_H$ is the set of PEs of the array and $E_H$ is the set of dedicated links or buses between PEs. We assume that if $|V_H| \geq |V_G|$ then there exists a mapping $\pi: V_G \rightarrow V_H$ such that the tasks in $G$ have been assigned to PEs in $H$ and the edges of $G$ to paths in $H$. A time schedule for task execution can be developed once the assignments have been made. The elapsed time to run a particular algorithm on a given machine is referred to as its schedule length.

Mapping thus consists of two distinct operations. The first operation assigns tasks to PEs and the second operation schedules their time of execution. Assigning the tasks of a DG to PEs of a parallel array is referred to as an embedding. This embedding is isomorphic if there is a one-to-one correspondence between the nodes of $G$ and the nodes of $H$ and adjacent nodes in $G$ are adjacent in $H$. Isomorphic embeddings for vision problems are often achievable. For example, image smoothing and 2D discrete fourier transforms are easily embedded on mesh or hypercube arrays while recursive doubling and global histogramming algorithms run efficiently on binary tree arrays. Isomorphic embeddings lead to optimal schedule lengths.

In some cases it will not be possible to achieve an isomorphic embedding. For example, the DG for global histogramming is a binary tree. Binary trees can be isomorphically embedded into hypercubes but not normally into mesh arrays. However, an embedding may be possible by inserting delay nodes into certain edges of the DG. Such a process yields a homeomorphic embedding.

Speedup is a measure of how much faster an algorithm runs on a parallel array as compared to a single processor. The best case occurs when speedup increases linearly with the number of processors. However, in an $N$ processor system, the speedup
cannot be greater than \( N \). To see this consider an algorithm that runs in \( T \) steps on a parallel array with \( N \) PEs. On a single PE this algorithm will take \( A \leq T/N \) steps. Since the speedup \( S = A/T \), clearly \( S \) cannot exceed \( N \).

The three factors that effect an algorithm's schedule length are 1) total computation time, 2) time to pass intermediate results between PEs (i.e., communication time), and 3) extent to which resources are shared. Studies have shown that if a certain ratio of computation time to communication time cannot be achieved in a parallel array, it will be better to run the algorithm on a single processor [Pric89]. In the following sections we discuss each of these factors.

2.1: Computation Time

Computation time is often influenced by the way data is represented. For example, suppose that an algorithm requires floating point rather than integer calculations. Unless each PE incorporates hardware that supports floating point operations, these operations must be done in software. The additional overhead associated with subroutine calls increases the computation time for each operation. For instance, the software must equalize exponents prior to performing an floating point addition. A proper choice of hardware resources can considerably decrease computation time and make the system level software easier to develop.

2.2: Communication Time

Communication time is influenced by a variety of factors. In non-shared memory parallel arrays message passing is the normal means of transferring data between PEs. Each message contains the data to transfer and the address of the destination PE. The parallel array also contains some form of routing circuitry that extracts the address information from the message and sets up the appropriate communications channel. This circuitry must close the communications channel after the message has been sent. This can take a relatively long period of time. For instance, in the NCUBE/2 hypercube, the start-up and close time for a connection is approximately 140 \( \mu s \) [Mess91]. There also a propagation delay time due to the channel bandwidth. These parameters determine the minimum communications time.

2.3: Shared resources

The authors of vision algorithms often claim that their algorithm will process \( N \) data elements on a parallel array with some non-exponential complexity (e.g., \( O(\log N) \) running time). These claims of performance assume that the parallel array is large enough to contain the entire DG. Unfortunately, in many instances the DG's size exceeds the size of the

Figure 3 Binary Tree Embedding in Mesh Array

The most obvious factor affecting communication time is the physical distance between PEs that must exchange data. As previously stated, the DG representing a global histogramming algorithm is a complete binary tree. Figure 3 shows one possible embedding of a complete binary tree of height \( h=4 \) into a mesh array. Notice that several nodes (e.g., the root node in the center of the array) are not directly connected to their two children nodes. It is not hard to see that the communication time in these cases will be greater since data must be routed through intervening PEs. This is an example of a homeomorphic embedding. Communication time is minimized by assigning adjacent nodes in the DG to adjacent PEs in the parallel array.
parallel array (i.e., $|V_G| > |V_A|$). In this case resources must be shared. This implies that the claims of performance can be overly optimistic.

Because of array size limitations it is often necessary to have a single PE perform several tasks in the DG. This leads to increased schedule lengths because even in parallel arrays a PE can only perform one operation at a time. For example, the $N=2^2$ point FFT has a DG of the familiar butterfly topology. This topology is easily embedded into an n-dimensional hypercube [Leig92]. Yet this embedding assigns several computations in the butterfly to the same PE in the hypercube.

Care must be taken in exactly how resources are shared as it can influence the schedule length. To illustrate, we use an image smoothing problem. An $M \times M$ pixel image is "smoothed" by averaging the gray code value of a pixel with that of its eight nearest neighboring pixels. Pixels on the borders of the image are not averaged; their value is kept. The following example was given by Siegel, et al. [Sieg92].

Suppose the time required to smooth four $M \times M$ pixel images on a $N \times N$ grid array is to be minimized. We could partition this problem two ways: 1) sequentially smooth each image (each PE stores a $M/N \times M/N$ pixel subimage), or 2) smooth the four images concurrently using $N/4$ PEs per image (each PE stores a $M/(N/4) \times M/(N/4)$ pixel subimage).

The total computation time remains the same regardless of which method is chosen. However, there is a difference in the number of inter-PE data transfers between the two methods. The first method requires $4 \cdot (4M/N + 4)$ data transfers while the second method only requires $(4M/N(N/4) + 4)$ data transfers. For $M = 512$ and $N = 1024$ this means 272 vs 132 data transfers for the first and second methods, respectively. The second method has a smaller schedule length (and greater speedup) because of less communication time.

McFarland et al., recently discussed several scheduling techniques [McFa90]. The first type of scheduling is called as soon as possible (ASAP). In this form of scheduling operations are topologically sorted (i.e., lexicographically ordered with respect to time.) Operations are taken one at a time and each is executed as soon as possible.

<table>
<thead>
<tr>
<th>ASAP SCHEDULE</th>
<th>LIST SCHEDULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEP</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Suppose a parallel array has two functional units that perform the operations '*' and '+'.

Figure 4 shows a DG and which computations are performed in these functional units using an ASAP schedule. The schedule length is 4 time steps.

When resources are shared the ASAP schedule may not be optimal. List scheduling is a means of overcoming this problem. In this type of scheduling the operations are assigned a priority; operations with the highest priority are allocated first to the available resources. The authors have indicated that list scheduling can produce results nearly as good as those obtainable by branch-and-bound techniques. The numbers in parenthesis in Figure 4 represents the length of a path to the end node. Notice that if the longer paths represent higher priorities, a schedule length of 3 time steps can be obtained with this method. Therefore, care must be taken in selecting a scheduling technique if speedup is to be maximized.

3: Summary

The authors of vision algorithms usually claim some level of performance. These claims can be overly optimistic if the computation time and
communication time are not quantified. We have shown that both of these parameters affect the schedule length of algorithms running on parallel arrays.

Examples were given to show how the embedding technique and problem partitioning can affect communication time. When the number of computations exceeds the number of PEs, it is necessary to share resources. But care must be taken in how this is done to minimize the schedule length. We have also shown that the scheduling technique chosen directly impacts speedup.

4: References


