A Model-Integrated Program Synthesis Environment for Parallel/Real-Time Image Processing

Michael S. Moore[a], Janos Sztipanovitz[a], Gabor Karsai[a], & Jim Nichols[a]

[a] Vanderbilt University Measurement and Computing Systems Laboratory
400 24th Ave S. Nashville, TN 37235 USA

ABSTRACT

In this paper, it is shown that, through the use of Model-Integrated Program Synthesis (MIPS), parallel real-time implementations of image processing data flows can be synthesized from high level graphical specifications. The complex details inherent to parallel and real-time software development become transparent to the programmer, enabling the cost-effective exploitation of parallel hardware for building more flexible and powerful real-time imaging systems.

The Model Integrated Real-Time Imagine Processing System (MIRTIS) is presented as an example. MIRTIS employs the Multigraph Architecture (MGA), a framework and set of tools for building MIPS systems, to generate parallel real-time image processing software which runs under the control of a parallel run-time kernel on a network of Texas Instruments TMS320C40 DSPs (C40s). The MIRTIS models contain graphical declarations of the image processing computations to be performed, the available hardware resources, and the timing constraints of the application. The MIRTIS *model interpreter* performs the parallelization, scaling, and mapping of the computations to the resources automatically or determines that the timing constraints cannot be met with the available resources.

MIRTIS is a clear example of how parallel real-time image processing systems can be built which are (1) cost-effectively programmable, (2) flexible, (3) scalable, and (4) built from Commercial Off-The-Shelf (COTS) components.

Keywords: image processing, real-time, parallel, parallel programming, parallel systems, model-based systems, model-integrated program synthesis, Multigraph architecture, TMS320C40

1. INTRODUCTION

Digital imaging applications require huge computational performance due to the large data sets involved. Applications such as robotics, military target tracking, autonomous vehicle control, and on–line video processing require sequences of images to be processed in *real–time*. Image sequences of normal resolution (640 x 480 pixels) and standard frame rate (30 $\frac{frames}{sec}$) with 256–level gray scale ($8\frac{bits}{pixel}$) represent a data rate of 8.8 Megabytes per second. A color sequence ($24\frac{bits}{pixel}$) at the same pixel resolution produces $26.4\frac{Megabytes}{sec}$. Typical applications require on the order of hundreds or even thousands of operations per pixel in order to enhance, segment, and extract features from image sequences,[1] which translates into a demand for tens of Giga–operations per second. Even less computational intensive applications, such as video enhancement, require hundreds of Mega–operations per second. Moreover, it has been estimated that future applications, such as dynamic scene interpretation, may require on the order of hundreds of Giga–operations per second.[2] Hardware architectures consisting of a single general–purpose processor are incapable of delivering these levels of computational performance.

Due to these high performance needs, most successful applications of real–time imaging have by necessity been built from custom hardware designed to perform fixed sets of specific image processing algorithms. Although such specialized hardware solutions have met computational requirements of some Real–Time Image Processing (RTIP) applications, there are many drawbacks to this approach. Hardware implementations are expensive, and real–time performance is achieved by sacrificing end–user programmability and flexibility.

Further author information (Send correspondence to M.S.M):
M.S.M.: Email: msm@vuse.vanderbilt.edu; WWW: http://www.vuse.vanderbilt.edu/~msm; Phone: 615-343-8302; Fax: 615-343-6702

The need has been stated for research efforts targeted toward producing real–time image processing support for recent applications such as remote command and control, High Definition Television (HDTV), virtual reality modeling, military target tracking, and rapid image identification.[3] These applications require more flexible and scalable real–time image processing solutions than are currently available. A more generalized and flexible approach toward system development is needed in order to support such applications.

The most obvious direction to take is to parallelize the computations and map them to a scalable parallel hardware architecture. Image processing algorithms have inherent concurrency which is relatively simple to exploit.[4] This idea is certainly not new. Papers describing parallel architectures, algorithms, kernels, programming models, compilers and software generators abound in the literature. However, even though there are many parallel hardware architectures which boast incredible numerical benchmarks, and there have been instances in which parallel implementations have been successfully deployed in real applications, the inherent difficulty of programming parallel machines has limited the number of cases in which programmers without parallel processing expertise have successfully and cost–effectively exploited the technology in a real–world applications.

It has become clear that developing a general parallel image processing system involves much more than building a high performance parallel hardware architecture or a parallel language. Some of the most difficult problems lie at the *systems* level, which includes the hardware, software, and their relationships. Without high–level programming environments and tools designed for building parallel *systems*, exploiting parallelism in real world image processing applications is, in most cases, not the most cost–effective or practical approach. An environment and framework for building parallel RTIP systems is needed, and it is crucial that this framework be system–centric, simultaneously treating parallel hardware, parallel software, and integration issues. The goal is a system with which non parallel programming experts can generate real–time parallel software implementations using the available parallel hardware architecture. The system must insulate the user from the underlying parallel implementation details, and retain generality, flexibility, and ease of use of uni–processor software systems.

In this paper, an approach toward this goal is presented. The claim is made (and supported) that, by using MIPS techniques and by taking advantage of the natural concurrency present in image processing computations, real–time parallel image processing systems can be automatically synthesized from high–level system specifications. Since the parallel part of the software is automatically generated, the complex details inherent to parallel software development are effectively rendered transparent to the programmer. The MIPS approach promises to lead to the cost–effective exploitation of parallel hardware architectures for building more flexible and powerful real–time imaging systems than are currently available.

The Model Integrated Real–Time Image Processing System (MIRTIS) is presented as a demonstration of this concept. MIRTIS was developed for both the real–time processing of video and the high speed processing of very large archived data sets.[5–7] It employs the Multigraph Architecture (MGA), a framework and set of tools for building MIPS systems, to generate image processing applications which run under the control of a parallel run–time kernel on a network of Texas Instruments TMS320C40 DSPs (C40s). The run–time system is configured automatically from graphical models declaring (1) the computations to be performed, (2) the C40 network configuration, and (3) the performance constraints of the target application. The configuration is performed by the MIRTIS *model interpreter*, which first analyzes the models to determine the feasibility of a real–time implementation, then (if feasible) parallelizes, scales, and maps the given computations to the resources such that the real–time constraints will be met. It also configures a graphical user interface with which the user can adjust processing parameters dynamically as the system runs.

The following sections discuss the image processing problem domain, and how MIRTIS uses the Multigraph framework to generate parallel real–time implementations of image processing computations. MIRTIS is evidence that (1) architectures build up of Commercial Off–The–Shelf (COTS) components can support the performance requirement of real–time imaging, (2) image processing algorithms are amenable to being parallelized and run on such networks, (3) given an adequate set of information about the image processing computations, the hardware architecture, and the run–time system, the processes of parallelizing and mapping image processing computations onto the hardware architecture can be automated. The conclusion made is that, although the MIRTIS interpreter was necessarily built around assumptions about the underlying run–time system, the MGA approach of specifying applications in terms of models provides a level of architecture independence which will allow much of the system to be re–used when the target hardware platform evolves with the availability of faster and cheaper hardware.

2. BACKGROUND

This section provides a brief description of image processing computations, the nature of real–time imaging applications, and the drawbacks of traditional hardware implemented real–time imaging systems.

2.1. Image processing computations

Image processing is related to the larger field of computer vision. Computer vision systems process data acquired from *image* sensors, which detect visible, infrared, or even magnetic radiation, and attempt to construct some model of the surroundings which may be used in formulating controls over the environment and/or presented for human interpretation. The early vision steps are made up largely of algorithms which operate on image data and produce images, transformations of images, or simple data structures describing images. These *image processing* algorithms are sometimes referred to as *low level vision*[8] because the role they play is to pre–process images for transformation into symbolic data (edges, connected components, etc).

Because applications from the video processing domain have driven this effort, the emphasis has been placed on image processing systems which are *persistent*. Persistent systems do not execute and terminate, but execute continuously, computing sequences of result data structures from sequences of input images. This motivates the extension of the genre of image processing algorithms to be considered from those which operate on images to those which operate on image streams, or even volumes.

Next, the image processing algorithm model used in this work is defined. The model, which accommodates image sequence computations concentrates not upon the mathematics of algorithms, but upon the way in general that data is accessed in computing the individual elements of the results sequence.

2.1.1. General image processing algorithm model

Consider image processing algorithm $F()$ which computes an output data structure sequence $O(k)$ by accessing pixels from N input image sequences $I_1(r,c,k)...I_N(r,c,k)$ and past values of the output sequence $O(k)$ (see figure 1). Select

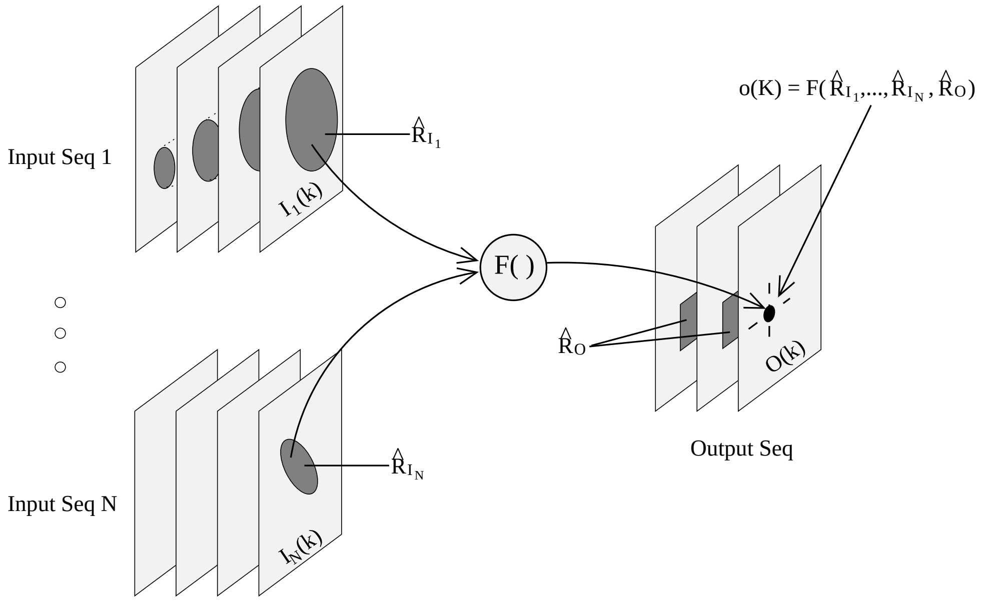


Figure 1. The image processing algorithm model

any particular element of the output data structure at time K, say $o(K) \in O(K)$. Note that if $O(k)$ is an image sequence, then this is a pixel in output frame K, and $o(K) = O(R, C, K)$ for some particular row R and column C. $o(K)$ is computed from some set of data made up of pixels from the inputs sequences, and elements from past values of the output data structure. These input data are represented by the grayed areas in the input and output sequences in figure 1. The corresponding pixels required from the n_{th} input sequence, I_n, in order to calculate the output element, $o(K)$, necessarily lie in a finite set of pixel locations which together form a *region* $\hat{R}_{I_n}$ inside sequence I_n.

$$\hat{R}_{I_n} = \left\{ \vec{P} = (r,c,k) \;\mid\; o(K) \longleftarrow I_n(r,c,k) \right\} \tag{1}$$

where $\vec{P} = (r,c,k)$ is the pixel location vector designating row r, column c, of frame k. The arrow pointing from the input location to $o(K)$ specifies that the particular output element $o(K)$ *depends upon* $I_n(r,c,k)$, in the sense that in order to compute $o(K)$, the algorithm requires direct knowledge of the pixel value at location $\vec{P}$ in I_n.

The union of the required regions from the input sequences with the required region from the past values of the output sequence $O(k)$ forms the *total data dependency set* of output data element $o(K)$, $\hat{D}_{o(K)}$.

$$\hat{D}_{o(K)} = \left\{ \bigcup_{n=1}^{N} \hat{R}_{I_n} \right\} \bigcup \hat{R}_O \tag{2}$$

2.1.2. Image processing algorithm definition

For the purposes of this paper, a function $F()$ operating on N input image sequences $I_1(r,c,k)...I_N(r,c,k)$ to compute output data structure sequence $O(k)$ is an image processing algorithm if and only if

- $F()$ is causal. For each output data element $o(K) \in O(K)$

  $$if\ \vec{P} = (r,c,k) \in \hat{D}_{o(K)} \;\longrightarrow\; k \leq K$$

  i.e. The data dependency set for a current output data element $o(K)$, $\hat{D}_{o(K)}$, contains only input data locations from the current and past input images.

- For each output data element $o(K) \in O(K)$

  $$\hat{D}_{o(K)} \cap O(K) = \emptyset$$

  i.e. The data dependency set for a current output data element $o(K)$, $\hat{D}_{o(K)}$, contains no data locations in the current output structure $O(K)$.

- For each output data element $o(K) \in O(K)$, the total data dependency set $\hat{D}_{o(K)}$ required to compute $o(K)$ is finite, and a bounding set can be determined a priori to the start of computation. The computations need not be fixed, as long as both the amount of computations and the regions required to compute each output have specifiable upper bounds.

This definition is quite non–restrictive, especially in light of the types of computations which are usually performed in low–to–mid level vision. These computations are regular, and usually traverse the image in raster order, performing fairly simple operations to produce each output pixel.[4]*

2.2. Image processing data flows

A popular method of building up image processing applications is to combine several pre–coded algorithms together to form a Large Grain Data Flow (LGDF). A LGDF is a directed graph in which the graph nodes represent processing blocks and the arcs represent communication between the blocks. The LGDF style specification is commonly used in signal and image processing. One reason is that it's visual nature promotes the integration of high–level graphical programming interfaces. For example, Khoros, a popular image processing development environment from the University of New Mexico, provides a visual programming interface in which data flow graphs are drawn to specify image processing computations. Khoros performs the control flow and transfer of data between the computation blocks automatically as it executes the data flow.[9]

*Note that the definition does not require that the computation traverse the image in raster order, or that the operations be simple. The statement was made to drive home that fact that the definition encompasses at least those types of algorithms.

In keeping with this approach, the problems considered are LGDF computations made up of algorithms meeting the definition in section 2.1.2. The following restrictions are placed on the data flow: (1) it must be a Synchronous Data Flow (SDF), which means that for each computation block, the amount of data consumed and produced each time the block runs is fixed,[10] (2) it must have no cycles (loops in the data flow graph that don't containing a delay element), and (3) it must have no *fan ins* (connections cannot be merged together). *Fan outs* are allowed (a connection can have multiple readers). Such computations will be referred to as *synchronous image processing data flows.*

2.3. Real–time image processing

Real–Time systems interaction with their environments, and thus must produce outputs which are not only numerically correct, but which also meet timing constraints necessary for these interactions. Such systems are said to be *embedded* into their environments. The relevant environmental interactions for image processing systems are receiving data from sensors or other systems, and outputting data to displays, plots, devices, or other systems, which may apply controls to the surroundings directly (e.g. a vision system might generate controls for a robotic arm).

2.3.1. Relevant timing constraints

RTIP systems may be required to service the input devices as quickly as the data is produced, produce outputs at a sustained rate, or produce an output from each particular input within a constrained amount of time. The two relevant types of temporal constraints in real–time image processing systems are:

- *throughput*: the rate at which results are produced by the system, usually quantified in $\frac{frames}{sec}$.
- *latency*: the total time between the sensing of a particular image and the results of that image leaving the system, usually quantified in *seconds*, and sometimes in $frames$.

It is important to note here the difference between a *high performance* system and a *real–time* system. Real–time image processing systems must not only support the large I/O data rates and computational power discussed previously, but the performance must be predictable and controllable. It must be known a priori to run–time (1) if a computation can be done within the timing constraints on the available hardware, and (2) if so, how to utilize the hardware to achieve the required performance. Thus, predictive models of both throughput and latency are necessary. Since it is generally difficult to accurately characterize the performance behavior of a computation, the models must be developed using knowledge of the algorithms and the underlying run–time system.

2.4. Specialized real–time hardware solutions

The approach traditionally taken in supporting real–time imaging has been to implement the most commonly used computations in specialized real–time hardware. Hardware implementation has been the most practical and cost–effective solution that could meet the performance requirements of RTIP. There are many commercially available machines which perform various set of standard image processing computations at real–time rates (eg. Matrox, Coreco, DataCube). Some are more general and flexible than others. However, no hardware solution provides all of the following:

- Programmability: They are either not end–user programmable, or offer limited programmability. Adding functionality may require costly VLSI design. It is not practical for the user to invent and experiment with non–standard algorithms. As systems such as Khoros[9] have shown, the ability to rapid–prototype and experiment with algorithms for the application at hand can greatly enhance the ability to generate effective solutions.
- Flexibility: The data paths are either hard–wired or have a fixed number of configurations, so the possible ordering of the computations is often limited.
- Scalability: More computations or higher performance cannot always be achieved by adding more hardware.
- Ease of use: They are difficult and expensive to use. Learning to use specialized hardware systems can require months of training, even for experienced image processing experts. Training time and cost is a major factor in the economics of computer solutions, since labor is traditionally more expensive than hardware.

- Cost–effectiveness of hardware: Since the systems do not use COTS parts, the hardware is expensive.

The use of specialized real–time image processing hardware has proven successful for some applications. However, the inherent limitations have caused the real-time imaging industry to develop the mind-set of trying to fit problems to the fixed capabilities of the available real–time hardware, instead of building integrated solutions to the problems at hand. This has had the unfortunate effect of placing a barrier between the algorithm development community and many real world embedded applications. Since it has not been feasible to create real–time implementations of new, non–standard algorithms to use in embedded imaging systems, much of the theoretical image processing developments have not been used in embedded applications.

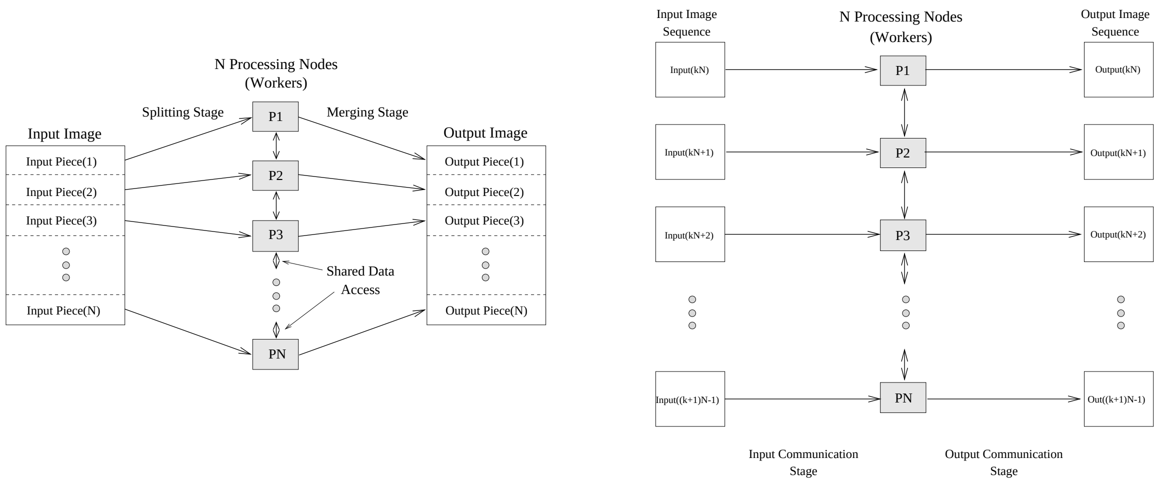
2.5. Parallelizing image processing computations

Instead of using hardware implementations, a more flexible approach is to generate parallel software version of the computations and map them to a distributed memory multi–computer.† Because the operations have characteristics which make them particularly suitable for implementation on parallel computers, image processing has been the most common area for the application of high performance parallel computing.[4,11]

The basic parallel computing constructs can be defined which best support these characteristics are data–parallelism (spatial decomposition or temporal decomposition) and functional parallelism. These parallel processing constructs and how they can be exploited in image processing are discussed next.

2.5.1. Spatial decomposition

Because of their properties, many image processing algorithms are easily data parallelizable by decomposition of the data in the image plane. A simple data parallel programming technique which is applicable to image processing is the *split–and–merge model*.[4] In this technique, each input data structure is *split* into N pieces, which can be blocks



Temporal data parallelism (sequence splitting)

Spatial data parallelism (split-and-merge)

Figure 2. Data parallel decomposition (spatial and temporal)

of rows, blocks of columns, panels, overlapping regions, etc. The pieces are distributed across the memories of the N *worker* processors, each which performs the same algorithm on its sub–section of the data. The partial results are then *merged* to form the output. In figure 2, a split–and–merge computation is shown in which the data is split into blocks of rows, and the final result is formed by concatenating the partial results.

Since each of the workers computes $\frac{1}{N th}$ of the result concurrently, the per image computation time is reduced by a factor of at most N. This has the potential of both reducing latency and increasing throughput when executing

†We assert that distributed memory multi–computer hardware architectures are appropriate for the RTIP application domain.[7]

on image sequences. However, the acceleration actually achieved depends upon the overheads introduced by the decomposition.

Referring to figure 2, sources of overhead in the split–and–merge processing model are (1) splitting inputs: distributing the image pieces to the processors, (2) merging outputs: communicating the partial results to be combined,[‡] and (3) sharing data: communicating *shared* data and partial results between worker processors.

How the data is to be split, shared, and merged depends upon the algorithm's data access patterns. Referring back to the general image processing algorithm model defined in section 2.1.2., note that the *total data dependency set* required by the algorithm in computing output pixel $o(K) = O(R, C, K)$ is sufficient information to determine which pixels of the input data must be available locally to the processor computing that data element.

2.5.2. Temporal decomposition

Algorithms which either cannot be split in this fashion, or for which the resulting gains of spatial decomposition would minimal, can still be successfully data parallelized when operating on image sequences by taking advantage of the sequence structure. Image sequences can be decomposed temporally instead of spatially (split the data along the *time domain*, instead of the *spatial domain*). Instead of distributing pieces of an image to worker processors, as shown in figure 2, the pieces of the image sequence (entire images) can be distributed as shown in figure 2. $Input(i)$ is distributed to processor $(i \ mod \ N) + 1$, where N is the number of processors, and $i \in \{0, 1, \ldots\}$.

Each worker processes an entire image, so there is no decrease in the time it takes for any one image to be processed (latency). However, images are being processed concurrently, so there is a potential increase in throughput.

2.5.3. Functional parallelism

Functional parallelism takes advantage of the natural concurrency of a computation. The computation is broken down into semi–independent sub–computations, which interact by passing data.

$$Sequential\ Computation \longrightarrow \{SC_1, SC_2, \ldots, SC_N\}$$

The sub–computations along with the data passed between them form an implicit data flow computation graph. The application of functional decomposition to parallelized LGDF computation is straight–forward. The N sub–computations can be executed concurrently on N processing elements, with data being transferred between the computations via inter–processor communication. The allocation of processes to processors is given by

$$SC_k \longrightarrow Node_{map(k)} \qquad (k = 1 \ldots N)$$

where $map(k)$ is the mapping function.

The performance gains of functional parallelism are achieved through allowing the sub–computations to execute concurrently, and are controlled by the structure of the data flow. The speedup is bounded above by N, and the efficiency, $\frac{speedup}{N}$, is affected greatly by the relative complexities of the sub–computations as well as the communications overhead. Unless the sub–computations are of similar complexity, the system will not be *load balanced.* Non load balanced systems have performance bottlenecks that result in the inefficient use of the parallel resources and poor performance gains.

2.5.4. Hybrid parallel constructs

It is reasonable to combine these approaches and form a hybrid parallel construct which uses two levels of parallel decomposition, the top data flow level being functionally parallel, and the underlying sub–computations being data parallel. Scaling and load balancing to the target throughput can be achieved by scaling the data parallelism of the sub–computations independently until each achieves the target throughput. This type of hybrid parallel construct can be applied automatically to data flow computations if methods of automatically data parallelizing and scaling the sub–computations are developed.

[‡]In some cases extra computations are required to combine the partial results to form the form the output.

3. APPROACH

The approach taken in this work has been to use MIPS techniques, specifically the Multigraph Architecture, to generate parallel software versions of synchronous image processing data flows made up of existing sequentially coded algorithms, and automatically scale and map the resulting decomposition to a parallel hardware architecture. This section provides an overview of the approach taken toward this automatic decomposition, scaling, and mapping. Then it introduces MIPS and the MGA, and how it can be used in automating the parallelization decisions.

3.1. Data flow decomposition and allocation to hardware

The data flows are decomposed using the two level hybrid parallel construct discussed in the previous section. As is shown in figure 3, the first step is to *partition* the data flow into computation blocks. Each block is then decomposed using either spatial data parallelism, temporal data parallelism, or no parallelism. Scaling is performed by assigning a number of processing nodes to each block. Referring to figure 3, note that *Block2* has been decomposed using spatial data parallelism, and assigned a scaling factor $N = 6$. The data parallelism has been implemented by making 6 replicas of the computation in *Block2*, which have been allocated to 6 processing nodes in the hardware architecture.

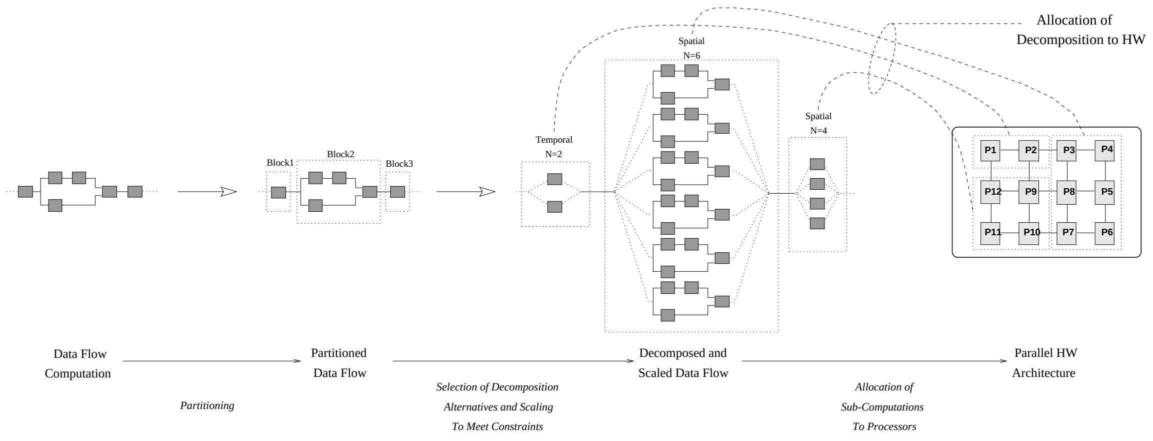


Figure 3. Mapping data flows to parallel hardware architectures

3.2. Difficulties

Nothing has been said yet of how the implementation, including the control flow and communication, is supported. Moreover, the decision making processes for selecting the types of parallelism, the scaling factors, and the allocation of the decomposed data flow to the processing nodes have yet to be discussed. At this point is where the concept of developing a *system* becomes most important.

The difficulty of the problem lies in that fact that the tasks of decomposing the algorithms, supporting the inter–process communication, providing control flow (synchronization and scheduling), figuring out to how many and to which processors the computations should be allocated, and determining how to route the communications through the communications network are inter–related. Some of these problems are difficult to solve even when considered alone. For instance, the general assignment problem is NP–complete.[12]

The following inter–related problems must be solved simultaneously (assuming the partitioning has been done):

- Decomposition of blocks: select a parallelization method for each block of the partition.
- Scaling blocks: select the number of processor for each block to load balanced and meet timing constraints.
- Allocation: assign the decomposed, sub–computations of the scaled data flow to the available processors.
- Communications routing: determine the path along which the communications will flow. For instance, each of the lines representing communication in figure 3 must be routed somewhere through the hardware network.

- Formation of performance models: determine accurately the performance that will result from a particular decomposition and mapping of a computation to the available resources.
- Support of parallel execution: provide the scheduling, communication, and synchronization to support execution of the data flow.

A major obstruction to automating the decomposition, scaling, and mapping processes is that these tasks are inherently inter–dependent. Note that the construction of performance models, which are used in making the parallelization decisions, requires knowledge of both the run–time system and the allocation of computations and hardware nodes. However, as figure 3 shows, the allocation is done after the parallelization decisions have been made. Performance models can not be built without taking into account the allocation, and vice versa.

Because of these inter–dependencies, the general problem of mapping image processing synchronous data flows to arbitrary multi–computer networks while simultaneously guaranteeing that throughput and latency constraints are met has no closed–form solution. Moreover, performing an exhaustive search of all decompositions and allocations until the constraints are met is not a practical alternative because the search space is too large.

The approach toward automatically mapping the computations to the resources must be to reduce the size of the search space by developing simplified decomposition procedures and allocation techniques which exploit the capabilities of the target hardware architecture and favor the properties of the majority of the applications. The is the approach taken in the MIRTIS system, as will be discussed in the next section. First, however, a short introduction to model–integrated program synthesis and the Multigraph architecture is in order.

3.3. Model–integrated program synthesis overview

MIPS is a method of synthesizing software systems from high-level models. MIPS is related to code generation performed by compilers, but the goal of MIPS is not the generation of machine code. MIPS systems generate instead either code to be executed on a virtual machine, or a configuration of existing computations. Common to all the various existing MIPS approaches is a component called the *model interpreter*, which actually performs the program synthesis. The model interpreter transforms high–level system models, specified in terms of a paradigm, or language, into the system program.[13]

3.4. The Multigraph architecture

The MGA is a MIPS architecture developed at Vanderbilt University which provides a frame–work and tools for (1) building graphical domain specific models and (2) transforming the graphical models into executable applications.[14] By using domain specific models and interpreters, MGA allows the domain experts to specify a system in familiar terms without dealing with the underlying software engineering details. The MGA consists of the following components: (1) A *graphical model builder* (GMB). This is a graphical environment in which domain specific models are built and manipulated. The current MGA model builder is called XVPE. (2) A *model database* in which the models are stored. The current implementation uses a public domain *Object–Oriented DataBase* (OODB) called *obst.* (3) Domain specific *model interpreters*, which translate the system models into the various components of the target system. (4) *Integrated applications* make up the target software system. (5) The *run–time kernel*, *application libraries*, and/or *operating system platform* together form the run–time environment.

For more detailed information about the MGA, refer to 14.

3.5. MGA models

The models are built and manipulated via the XVPE graphical model building environment. XVPE is configured with a domain specific *modeling paradigm* which contains the concepts particular to the application. The modeling concepts available in the MGA system include attributes, parts, hierarchy, connection, association, reference, and multiple aspects. This set of modeling concepts has been shown to be rich enough to support the needs of a large and diverse set of problem domains.[15–19]

4. MIRTIS

4.1. Overview

MIRTIS is an MGA–based realization of the ideas which have been developed in the previous sections. It uses a combination of automatic program translation and meta–level driven software synthesis to automatically parallelize image processing data flows made up of sequentially coded algorithms. The parallelization decisions (types of parallelism and scaling factors) and the allocation of the decomposed data flow to the parallel architecture are performed automatically. The decision are driven by the real–time constraints, which are modeled explicitly.

It was determined that the decomposition and allocation algorithms should be simplified in order to decrease the search space of data flow to network mappings, and to make the automatic mapping of processes to processors a practical endeavor. The simplification in the mapping algorithm was made possible by adding complexity to the run–time system. Specifically, a special routing technique was implemented for the C40 which enables all communication to be routed along a hardware pipeline.

The run–time system was kept semi–architecture independent by implementing the communication components as a separate layer which can be re–implemented for new hardware architectures, thus allowing the re–use of a large part of the implementation. Both a prototype C40 run–time system and the resulting mapping algorithm have been designed and implemented for this work. The special communications routing support in the run–time system reduced the complexity of the mapping algorithm significantly, making the interpretation process more feasible.

4.2. The MIRTIS architecture

The MIRTIS architecture, shown in figure 4, follows the MGA framework, and consists of the following: (1) the IPDL model building environment, (2) a model database, (3) the MIRTIS model interpreter, (4) an image processing application library, (5) the PCT–C40 run–time system, (6) the MIRTIS graphical user interface, and (7) a network of C40s. The most important aspects of these elements are discussed next.

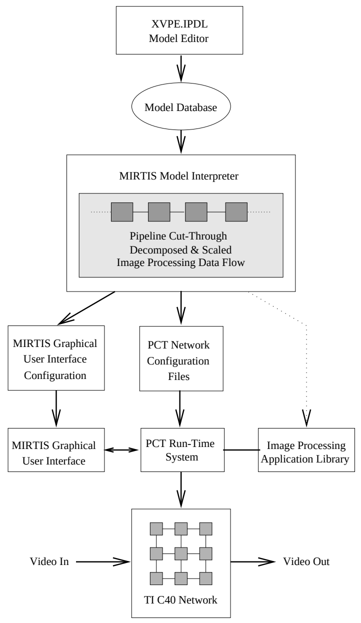

XVPE.IPDL
Model Editor
Model Database
MIRTIS Model Interpreter
Pipeline Cut-Through
Decomposed & Scaled
Image Processing Data Flow
MIRTIS Graphical User Interface Configuration
PCT Network Configuration Files
MIRTIS Graphical User Interface
PCT Run-Time System
Image Processing Application Library
Video In
TI C40 Network
Video Out

Figure 4. The MIRTIS architecture

4.3. The IPDL modeling paradigm

The MIRTIS modeling paradigm, called Image Processing Description Language (IPDL), was designed specifically for real–time image processing. The concepts were developed by extracting the set of information required to support

The IPDL Paradigm		
Paradigms	Models	Model Aspects
SignalFlow	Algorithm	Structure Constraints DataDependency ParameterInterface
	Application	Structure
Hardware	Node	Hardware
	HostNode	Hardware
	Network	Hardware
Constraints	RealTimeConstraints	Goals

Table 1. Concepts in the IPDL modeling paradigm

the automatic decomposition and mapping approach outlined in the previously. This section will briefly describe the paradigm, putting emphasis on the novel concepts.

As is shown in table 1, IPDL contains three types of models, *Signal Flow*, *Hardware*, and *Constraints*, which represent the data flow computation to be performed, the hardware resources available for the solution, and the timing constraints required by the solution, respectively. The combination of a Signal Flow Application model, a Hardware Network model, and a Constraints RealTimeConstraints model together form a the specifications for a real–time image processing *system*.

4.3.1. Signal flow models

Signal flow models specify the image processing computations to be performed. The two types of signal flow models are *applications*, and *algorithms*. Applications are simply data flow graphs made up of algorithm models, each which declares pertinent information about an algorithm in the image processing library. Only the most important parts of the algorithm model are discussed here. For a detailed description, along with examples, see 7.

Algorithm data dependency specification: When implementing temporal or spatial decomposition, the interpreter must be able to determine how the input data is to be split and allocated to the worker processors so it can (1) configure the communication system correctly, and (2) accurately model the communication overheads.

For modeling the data dependency behavior of the algorithms, a mathematical dependency specification format was devised that is unique to this development. The idea is to show an algebraic relationship between an output pixel location and a region in an input image sequence. The data dependency specification is contained in an attribute of each algorithm model, the format of which is given by:

$$Out[rvar, cvar, tvar] \quad "\longleftarrow" \quad In[rowrange, colrange, timerange]$$

$$range \;=\; begin()\;[\;"\ldots"\;\; end()] \;\;|\;\; ":"$$

where *Out* and *In* are names of two of the algorithm's image signals, and *rowrange*, *colrange*, and *timerange*, following the *range* format. In the range specification, *begin*() and *end*() are algebraic formulas specifying the first and last indices of the range, and a ":" specifies the entire gamut of valid indices.

A data dependency parser was implemented which parses and evaluates the data dependency specifications. During interpretation, the specification is parsed to determine the data access patterns for each algorithm in the data flow. For a particular decomposition method, this information is used in characterizing communications overheads in the performance models and in configuring the PCT communications engines.

Algorithm benchmarks: The approach taken in modeling each algorithm's execution time as a function of data size was to rely upon empirically gathered benchmarks. Using measured execution times instead of other approaches is both more accurate and straight–forward.

Each algorithm model contains a set of *benchmark* parts, each having an attribute specifying the dimensions of the algorithm's input images for that measurement. By providing execution time benchmarks for various data sizes (on a single node of the target architecture), an execution time versus data size curve can be constructed. The method of estimating the execution time of an algorithm for a particular data size is to use linear interpolation on the benchmarked data sizes, including an implicit benchmark of zero seconds for zero data size. This unique benchmark–based interpolative approach to approximating execution time has proven to provide very accurate results in testing.

4.3.2. Hardware models

The types of hardware models are *Node models*, *HostNode models*, and *Network models*. Network models are inter–connected hierarchies containing nodes, hostnodes, and other networks. Node models represent the processing nodes which perform the image processing computations on the data stream. In the case of the prototype, these are C40s. Each node model has attributes describing the node type and configuration, the ports, and any special resources, such as frame digitization hardware. HostNode models represent the PCs or workstations, which provide services such as network loading and disk I/O. They contain various attributes and parts, notably host interface card parts, which are boards in the host bus that provide communication links to the C40 network through which loading and run–time communication occur.

4.3.3. Constraints models

Constraints models contain explicit declarations of the target latency and throughput required for an application. Throughput models have a numerical attribute specifying frame rate in $\frac{frames}{sec}$, and latency models have attributes specifying latency in $frames$. Both throughput and latency models have attributes specifying whether it is a *hard* or *soft* constraint. As will be seen, this attribute is used in the interpretation procedure in the case that the constraints cannot be met exactly. It specifies whether that constraint can be relaxed to allow a *best effort* implementation on the available hardware.

4.4. The PCT–C40 run–time system

A real–time image processing kernel called PCT–C40 has been implemented which provides run–time support for spatial and temporal data parallel execution of image processing synchronous data flows on pipeline–connected C40 networks. The kernel runs on each C40 node and performs the scheduling, communication, and synchronization necessary for data parallel computations.

The scheduler on each node runs a Periodic Admissible Sequential Schedule (PASS)[10] which implements the synchronous data flow local to that node. The kernel configures and starts the PCT communication engine, which, in cooperation with the neighboring nodes, distributes the input data appropriately across the processors and combines the local results to form the output data. Since the computation and communication schedules are static, the scheduler introduces minimal run–time overhead. This also has the effect of simplifying the kernel by pushing the work of generating computation and communication schedules into the model interpretation process.

4.4.1. Pipeline cut–through overview

Pipeline Cut–Through (PCT) is a communication technique which allows synchronous data flows parallelized with the spatial or temporal data parallel constructs to be mapped to a group of C40s connected in a pipeline (a PCT group). PCT achieves this by routing all communications, including the distribution (splitting) of input data and collection (merging) the partial results, along the C40 pipeline. PCT also provides coordination between the communication and computation processes. Since PCT implements the parallel facilities automatically, the data parallelism is absolutely transparent to the programmer.

Each node of a PCT group performs the same computations on a different section of the image data. The incoming stream is split and spread across the memory banks of the group nodes, and after the local data flow computation has produced the partial results, they are combined (merged) to form the output data stream. As well as splitting and merging the data stream, the communication engine also supports the sharing of pixels from the input sequences between two or more nodes in a PCT group. For more information about PCT, see Refs. 5–7.

4.5. The image processing library

The actual image processing functionality is provided by a library of image processing algorithms written in C and compiled with the standard Texas Instruments C40 compiler. This library is the simplest component of the system, since the image processing functions can be written as if they were to be used in a normal uni–processor system.[§] It was decided to take this approach instead of generating the image processing specific code directly from the most so that image processing libraries optimized for the target architecture could be re–used, which saves time and results in better resource utilization.

4.6. The MIRTIS interpreter

The model interpreter is the heart of any MGA system, and requires the largest implementation effort. The job of the MIRTIS model interpreter is to translate the IPDL models into a scaled decomposition of the data flow, map the decomposition to the underlying hardware architecture, and construct network communication and computation schedules which configure the real–time image processing kernel and realize the parallel real–time data flow. Referring to figure 4, the products of the interpretation are (1) PCT network configuration files, and (2) a GUI configuration file. These files are used in (1) booting the network, (2) configuring the network communication engines and schedulers, and (3) configuring the dynamic parameter graphical user interface.

4.6.1. Relationship between performance models and allocation

Performance models are needed for determining (1) if a particular computation can meet the specified performance goals using the available hardware, (2) a decomposition method and granularity of parallelism (scale) for each block, and (3) a mapping of the decomposed computations to the hardware that will meet the constraints.

In general, performance models are dependent upon the properties of the particular computations, the parallelization technique, and the allocation to the hardware network. This forces the processes of decomposing the data flow and allocating it to the hardware to somehow occur simultaneously. It is preferable to decouple these processes to make the mapping more practical to automate.

Due to the properties of the PCT communication technique and the support provided by the PCT run–time system, the allocation scheme and hardware topology can be simplified enough that the throughput and latency models can be built in a separate step before allocation. This effectively decouples the decomposition and allocation processes, making the automation of mapping data flows to hardware tractable. The emphasis in developing performance models can thus be placed on the properties of the computations. See 7 for a complete development of the performance models.

4.6.2. The interpretation procedure

Because the decomposition and allocation processes have been effectively decoupled by the assumption that the PCT run–time system will be used, the search for an appropriate mapping between the image processing data flow and the hardware pipeline can be reduced to finding an appropriate partition, and choosing a *decomposition alternative* (a supported type of parallel decomposition) and a scaling factor for each PCT Block.

Judging the success of a particular decomposition involves building throughput and latency models and comparing them to the throughput and latency goals specified in the system's RealTimeConstraints model, then making sure that the hardware architecture can support the decomposition. Enough of the right kind of processors must be available, and they must be connected in an appropriate topology.

[§]The functions must follow a loose format and set of rules. For instance, there is a programming API through which must be used in obtaining the input and output buffers.[5]

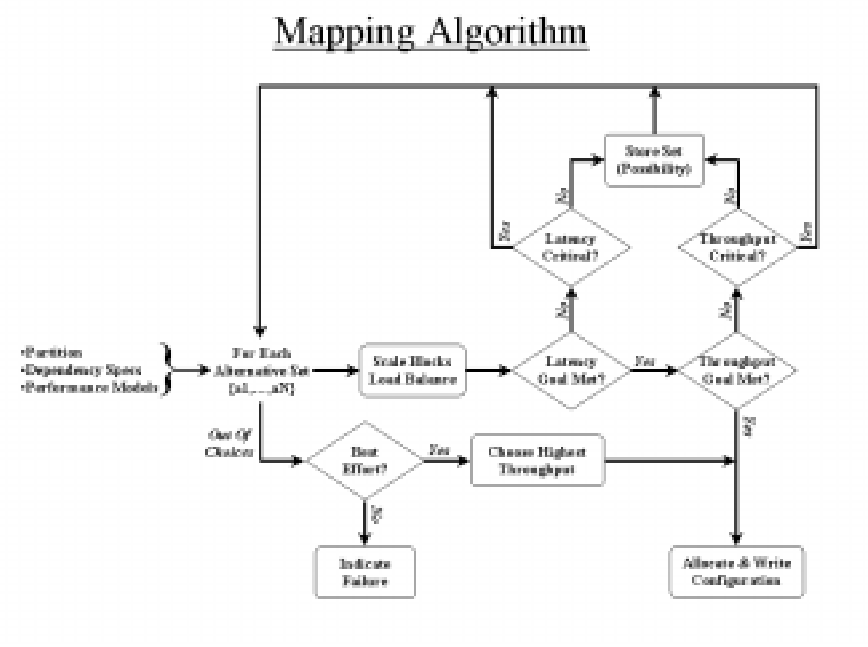


Figure 5. The interpretation procedure

The procedure followed by the interpreter is to first partition the synchronous image processing data flow such that it is compliant with the PCT run–time system (see Refs. 5–7). Then a search is performed for a combination of block decomposition alternatives and scaling factors that will meet the performance constraints and that the hardware architecture can support.

The interpretation algorithm, shown in figure 5, performs an exhaustive search of all decomposition alternative combinations until either the constraints have been met, or the valid alternative sets have been exhausted. Alternative combinations which meet the hard real–time constraint(s), but may not meet the other(s), are stored in the *possibilities set* during the search. The end result of a successful search is a partition, and a set of decomposition alternatives and scale factors for the partition blocks which can be allocated to the hardware pipeline to achieve the target throughput and latency. If no solution was found during the search, an attempt is made to relax the throughput and/or latency constraints. If both constraints are *hard* constraints, then no concessions are made. Otherwise, the alternative sets which were stored in the *possibilities set* are examined, and the one which most nearly matches the constraints is chosen. The decision of which of these most nearly matches the constraints is made by putting priority on throughput by choosing the set which produces the highest frame rate. This decision was made primarily because the system was designed with real–time video in mind, and in video applications throughput is most often the more important constraint.

The allocation occurs only after a scaled decomposition has been chosen for the solution. This decoupling of the decomposition and allocation was made possible by first partitioning the data flow and using the PCT run–time system. Without this simplification, the performance models would be inextricably dependent upon the allocation, and thus a much more complicated procedure would be required.

5. CONCLUSIONS

This work has begun to address the problems which must be solved to support the image processing and vision applications that will become prevalent in the future. Some vision applications may require on the order of hundreds of billions of operations per second. The work has taken a novel approach toward parallel programming by generating parallel real–time implementations of image processing data flows from high–level specifications.

The implemented system includes a graphical environment with which the user builds visual models of the data flow computation, the hardware resources available to solve the problem, and real–time specifications of an application. A model interpreter automatically transforms these models into a configuration of a real–time system which executes the modeled computation. The interpreter performs the data flow decomposition, performance modeling, scaling, load balancing, and scheduling automatically, then allocates the decomposed, scaled computation to a network of DSPs. A parallel image processing run–time kernel provides communication, routing, scheduling, and synchronization for the implementation.

Although the MIRTIS interpreter was necessarily built around assumptions about the underlying PCT–C40 run–time system, the MGA approach of specifying applications in terms of models provides a level of architecture independence which is expected to allow much of the system to be re–used when the target hardware platform evolves with the availability of faster and cheaper hardware.

ACKNOWLEDGEMENTS

This work was partially supported by the AFOSR/AFMC, US Air Force, contract number F49620-94-C-0076.

REFERENCES

1. C. Weems *et al.*, "The darpa image understanding benchmark for parallel computers," *Journal of Parallel and Distributed Computing* **11**, pp. 1–24, 1991.
2. C. Weems *et al.*, "Ui parallel processing benchmark," *IEEE Journal of Parallel and Distributed Computing* , pp. 673–688, 1988.
3. P. A. Laplante, "Issues in real-time image processing," *Proceedings of the 1993 IEEE Systems, Man, and Cybernetics Conference* , pp. 323–326, (Le Touquet, France), October, 1993.
4. J. A. Webb, "Steps toward architecture-independent image processing," *IEEE Computer* , pp. 21–31, February, 1992.
5. M. S. Moore, "A dsp-based real-time image processing system," *Proceedings of the 6th International Conference on Signal Processing Applications and Technology (ICSPAT)* , (Boston, MA), October, 1995.
6. M. S. Moore and J. Nichols, "Model-based synthesis of a real-time image processing system," *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)* , (Ft. Lauderdale, FL), November, 1995.
7. M. S. Moore, *Model-Integrated Program Syntheis for Real-Time Image Processing.* PhD thesis, Vanderbilt University, Nashville, TN, May, 1997.
8. A. Choudhary and S. Ranka, "Parallel processing for computer vision and image understanding," *IEEE Computer* , pp. 7–10, February, 1992.
9. J. Rasure *et al.*, "Visual language and software development environment for image processing," *International Journal of Imaging Systems and Technology* , pp. 183–199, August, 1990.
10. E. A. Lee *et al.*, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers* **36(1)**, pp. 24–35, January, 1987.
11. J. A. Webb, "High performance computing in image processing and computer vision," *Proceedings of the International Conference on Pattern Recognition* , (Jerusalem), October, 1994.
12. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theorey of NP-Completeness*, W.H. Freeman and Company, New York, NY, 1979.
13. B. Abbott *et al.*, "Model-based software synthesis," *IEEE Software* , pp. 42–52, May, 1993.
14. G. Karsai, "A configurable visual programming environment," *IEEE Computer* , pp. 36–44, March, 1995.
15. T. Bapty *et al.*, "Synthesis of large-scale real-time instrumentation systems using model-based techniques," *Proceedings of the Software Engineering Research Forum* , (Boca Raton, FL), 1995.
16. R. Carnes *et al.*, "Integrated modeling for planning, simulation and diagnosis," *Proc. of the IEEE Conference on AI Simulation & Planning in High Autonomy Systems* , (Cocoa Beach, FL), April, 1991.
17. A. Misra *et al.*, "A model-integrated information system for increasing throughput in discrete manufacturing," *International Conference on Engineering of Computer Based Systems (ICBS)* , (Monterey, CA), March 1997.
18. G. Karsai *et al.*, "Model-based intelligent process control for cogenerator plants," *Journal of Parallel and Distributed Computing* **15, no. 6**, pp. 90–102, 1992.
19. S. Padalkar *et al.*, "Real-time fault diagnostics with multiple-aspect models," *Proc. of the IEEE International Conference on Robotics and Automation* , pp. 803–808, (Sacramento, CA), April, 1991.