
ELSEVIER

Parallel Computing 23 (1997) 1793–1809

PARALLEL COMPUTING

A system for parallel media processing

John A. Watlington, V. Michael Bove Jr. *

MIT Media Laboratory, 20 Ames Street, Room E15-324, Cambridge MA 02139, USA

Received 5 November 1996; revised 12 May 1997

Abstract

We describe a parallel computer system for processing media: audio,video, and graphics, among others. The system supports medium to coarse grain parallelism, using a dataflow model of execution, on a range of machine architectures scaling from a single von Neumann or general purpose processor (GPP) up to networks of several hundred heterogeneous processors. A distributed resource manager, extending or subsuming the functionality of a traditional operating system, is an integral and necessary part of the system. While we are building a system for processing a variety of media, in this paper we concentrate on video because it provides an extreme case in terms of both data rates and available parallelism. © 1997 Elsevier Science B.V.

Keywords: Multimedia; Dataflow; Streams; Digital signal processing

1. Introduction

In order to provide higher compression, greater flexibility, and more semantic description of scene content, video is increasingly moving toward representations in which the data are segmented not into arbitrary fixed and regular patterns, but rather into objects or regions determined by scene-understanding algorithms [12,15]. These representations are effectively sets of objects and 'scripts' describing how to render output images from the objects. In other papers, we have described the advantages of this approach, and have shown a general structure for decoding flexible, object-based video representations [5,4]. While it is possible to view this trend as reducing the regularity of the processing, and thus reducing the efficiency available from computational optimizations, we note that the individual objects are amenable to the same sorts of techniques

* Corresponding author. E-mail: vmb@media.mit.edu.

PII S0167-8191(97)00088-4

used in 'traditional' video coders, and also that the encoding algorithms require vastly more processing than current DSP-based approaches; further, most such advanced representations contain sufficiently many degrees of freedom that hardwired algorithmic pipelines are out of the question, and it becomes even more important to consider strategies for efficient programmable systems. Irrespective of the coding method, computational needs for video are likely to increase greatly in coming years. Digital video – unlike digital audio – is far from operating at human perceptual limits. As display technologies and communications bandwidth permit, higher definition systems will add to the computational demands. Alternative output technologies, for example the holographic video displays developed at the MIT Media Laboratory [18,22], push these demands still further.

Custom processors operating in parallel and using hardwired communications networks are capable of meeting the computational demands of media processing for a given application, yet the flexibility to support different algorithms is difficult to provide with these architectures. Single 'general-purpose' processors, now often with specialized instructions/datapaths for manipulating small data elements in parallel (e.g., using a 32-bit ALU to process 8-bit R,G,B pixel values simultaneously [20,13]), provide adequate flexibility and are showing promise of meeting the needs of the current generation of media applications. Yet, for the reasons presented above, algorithms and applications which require tens to thousands of times more computation and memory bandwidth than current applications are being developed. Since these requirements of media processing greatly exceed the capabilities of single processors, programmable parallel architectures will remain attractive for all but the cheapest or most limited media applications.

The system model described in this paper is an attempt to support computationally demanding media tasks in an environment in which the programmer can take advantage of parallelism and specify real-time performance without needing to know details of the hardware architecture(s) used to execute the tasks. The parallelism can extend to processing resources available 'outside the box'. Thus differently-scaled or -architected systems can execute the same software, and can avail themselves of other local, underutilized processors; consider, for instance, a VRML viewer on a personal computer borrowing cycles from the rendering engine of a video game in the next room, or several PCs working together to achieve real-time media encoding.

2. Characteristics of media processing

To understand how we might address the problem, we should examine the characteristics of both digital video data and typical processing algorithms. Relevant points about video data are:

- The information is typically stored in compactly filled multidimensional arrays.
- The information extent is typically unbounded along at least one dimension (time).

- If supporting object-based video, the information extent along one or more dimensions may vary (within bounds) along a separate dimension (for example, x and y size varying along the time dimension).
- There is a huge amount of information involved.

The last item provides most of the difficulty involved with video processing. Fortunately, most algorithms of interest do not need concurrent access to a large number of the elements in a data array. Processing commonly operates independently in a locality of limited dimensionality (one color component in one region of a frame, one frame, or one small region of a group of frames), providing a large amount of potential data parallelism. Within the smaller locality accessed by a given invocation of an algorithm step, the data access pattern is typically fixed. We will describe a mechanism, streams, which exposes and utilizes these algorithmic regularities.

3. Hybrid dataflow

An imperative algorithm specification (i.e., a sequence of instructions to be executed sequentially), while an ideal means of controlling a single von Neumann or Harvard architecture processor, provides little opportunity for parallel execution. Instruction level parallelism of small numbers of instructions may be detected and used, but larger amounts of parallelism are not available unless explicitly supported by the application algorithm. This requires the programmer to determine the parallelism, in many cases fixing the granularity at compile time for a particular machine. The fundamental issues encountered in building a parallel processing system [1] are particularly difficult to overcome using imperative algorithm specifications.

A fine-grained dataflow specification provides the maximum available execution parallelism for a given algorithm, and directly addresses the fundamental issues mentioned above. Unfortunately, a fine-grained dataflow implementation encounters the high overhead of synchronizing (token matching, or scheduling) every instruction, producing results which are not competitive with imperative implementations of equivalent cost.

Several of the refinements to fine-grained, static, dataflow are attempts to utilize data and instruction locality [7]. In particular, *hybrid dataflow* schemes [9,16,17] propose a scheduling quantum larger than a single instruction in an effort to minimize the amount of synchronization while still providing an acceptable level of parallelism. The number of basic instructions in the sequence varies from one (fine-grained dataflow) through the tens and hundreds (medium-grain) up to entire applications (coarse-grain). Note that the machines used to execute the instruction sequences may themselves use multiple processing units to efficiently utilize the available instruction level parallelism within the sequence.

[1] According to Arvind [1], the two fundamental issues encountered in building a parallel processor computer system are: (1) the non-deterministic latency associated with accessing shared memory in a multiprocessor, and (2) obtaining efficient synchronization of a process across multiple processors.

We also propose using medium-grain hybrid dataflow in part because the instruction sequences represent algorithm sections (or tasks) of a size suitable for accelerating with a typical specialized processor.

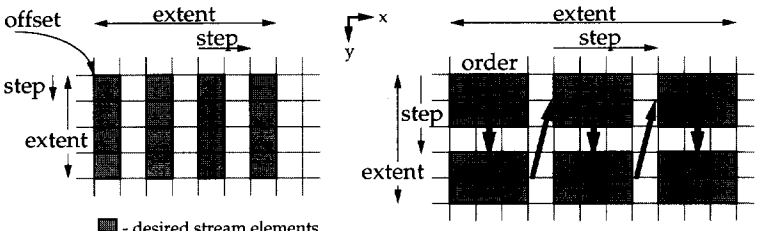
4. Specialized processors

Building a system with the processing power required for analyzing or synthesizing video using only a network of homogenous GPPs is a costly and unwieldy solution. We propose that some of the processing elements be *specialized processors* – capable of executing a restricted set of algorithms much more efficiently than a general purpose processor. Examples of specialized processors are SIMD and MIMD arrays of processing units, well suited to such tasks as matrix multiplication, convolution, and vector distance calculations (e.g., vector quantization or motion estimation). Another example is a configurable processor, which may be rapidly configured to provide application specific functionality [23,8].

Algorithm tasks are dynamically mapped onto the most appropriate (i.e., fastest for it given the current architectures, loading, and data locality) processing elements available in a system by a *resource manager*. The processing elements are grouped into *processing nodes*, which contain a general purpose processing element executing part of the resource manager, and possibly other specialized processing elements which it may control or configure. Even if we limit our machines to general purpose processors, a variation incapabilities among them (e.g., faster processor, larger data cache, or different communication latency) allows us to preferentially schedule non-parallelizable sections of algorithms to more capable processors. Specifying a task in a heterogeneous processing environment must be done in a manner independent of the machine architecture chosen for execution. The simplest method, run-time interpretation of an intermediate, machine independent language, imposes an unacceptable performance penalty. A more attractive solution is to compile, at application start-up, the intermediate form of each task specification into instruction sequences for the particular machine architectures present in the system. While this is quite feasible for general purpose processors, the compilation of configuration information for completely or partially reconfigurable (specialized) architectures is not as well developed. In these cases the 'instruction sequence' may more closely resemble datapath configuration information than any sequence of instructions. In order to support specialized processors easily at the present time, the task representation proposed is a set of pre-compiled instruction sequences – each intended for execution on a particular machine architecture. The use of shared libraries allows common tasks to be easily extended to support system-specific specialized processors. Otherwise, the application would have to be pre-compiled for all target architectures: a highly undesirable situation.

5. Streams

We extend hybrid dataflow with the introduction of *streams* [21], a natural means of describing data which changes along any dimension. In its simplest manifestation, a stream may represent a variable which is modified over time. Each entry along a



a.) D{x, y} E{4, 4} S{2, 1}

b.) D{x, y} E{3, 2} S{4, 3} O{2,1}
of D{x, y} E{3, 2} S{1, 1} O{1,2}

Fig. 1. Stream parameters.

dimension represents a change in the value of the variable. A multidimensional stream is essentially a multidimensional array of relatively small (8–1024 bits) data elements, possibly sparsely populated or unbounded in size along one or more dimensions.

Streams allow the intelligent prefetching of data in order to overcome the uncertain latency of accessing shared data and also provide a convenient framework for performing synchronization. They are a mechanism for specifying a data interconnection between functional tasks, representing (possibly elastic) delay elements for temporary storage of intermediate results. Streams explicitly identify the data parallelisms and access regularities available in the application of a task to a number of data elements. The granularity with which a stream is processed is determined at runtime, providing (within limits) an appropriate level of parallelism for a particular system.

Collapsing a multidimensional array down to a linear sequence of values conceptually involves a set of nested, bounded, incrementable array index counters. We standardize the description of these virtual counters using two or more integer parameters, which may be repeated in a hierarchical manner to describe either a stream segment accessed by a single invocation of a task, or the mapping between a multidimensional data array and its linear form. Two example stream access patterns and their parameters are shown in Fig. 1. Part (b) of the figure shows an example of a two level access pattern, with the step and extent of only the top level indicated. These parameters (usually abbreviated to their first letter) are:

Dimensions Each stream is defined for some number (possibly zero) of dimensions. There is no change in the stream along undefined dimensions.

Offset Establishes the absolute location in the stream of the lesser bound of this stream fragment along each defined dimension. (Abbreviated as L, not O.)

Extent This is the maximum range of the stream fragment along each defined dimension. This may be infinite.

Step The *step* parameters give the counter increment in each dimension. There are several 'special' values that a step parameter may have for a particular dimension. One is zero, which indicates that the same values are to be replicated along that

dimension. The value of the extent along any dimension of the stream (denoted by E_{dim}) may also be specified as a step parameter, allowing the multidimensional array to be stored in a 'packed' format.

Order This is the order of counter nesting (e.g. do we scan horizontally and then vertically, or vice versa).

As there is no limit on the number of dimensions which may be defined in a program, it is safe to say that every stream is embedded in a higher dimensional space. When a task has one or more streams as arguments, it is applied to a particular *fragment*, or subsection, of each stream. If the task only operates over a lower dimensional or smaller space than the fragment, it will be applied multiple times over subspaces of the fragments. This allows synchronization and scheduling costs to be amortized over multiple task invocations. There are a number of common operations (such as convolution, or estimation of image motion) that operate in a area of data which is changed only incrementally between task applications. Using the above access pattern definition, this neighborhood is indicated independently for each dimension, and is defined as the sample extent times the absolute value of the sample step – if the step is non-zero – and one otherwise. The size of the 'sample overlap' along each dimension is equal to the sample neighborhood minus the absolute magnitude of the block step.

If the sample overlap is non-zero, the resource manager will attempt to pipeline the stream operation, thereby amortizing the cost of transferring the overlapped data over as many task invocations as feasible. Pipelining is possible when the function can be executed on a processor that has sufficient local storage to store stream values that it will need again. In the one-dimensional case, there must be enough memory to store the sample extent. In the N-dimensional case, there must be enough memory to store the multiple of the number of task invocations being overlapped times the sample extent for each of the $N-1$ lower dimensions. When it is desirable to pipeline the processing of data extents larger than the local storage provided by the processing units in a system, the stream may be automatically partitioned into multiple smaller streams in order to avoid overflowing the processor local memory, in a manner identical to that used for exploiting data parallelism.

Elements in a stream may only be assigned once. Once written, an element may not be modified. A task which accesses stream elements which have not yet been assigned will not be scheduled for execution until the stream elements are available. While this synchronization of data is effectively performed on each individual value of each stream, for efficiency's sake, it is almost always performed on larger blocks of data. The size of a synchronization data block is determined by the resource manager at runtime, taking into account the stream access pattern, amount of pipelining, and complexity of the input task and output task(s) of a stream.

A user application is responsible for declaring a stream – in the initial program environment or using a system call – before declaring any tasks using it. What is really created is a *stream directory structure*. This structure, which is incompletely replicated across all the processors accessing a stream, contains an entry for each stream fragment in local memory, and information on which processor to query for more data in a

particular dimensional extreme (i.e. for stream fragments with an X dimension position lower than C, ask node D). The resource manager, not the application, is responsible for allocating the actual memory for a stream. Upon stream creation, an attempt is made to allocate adequate stream buffering, but this allocation may be increased during execution if necessary.

6. Naming and protection

We use a capability mechanism [14,19] to provide naming and protection. Capability based addressing is similar to segmented addressing, or addressing through an object descriptor. Like those mechanisms, it provides a level of indirection that aids memory relocation (e.g. objects may be moved into slower storage over time). It also provides for context independent naming (objects may easily be interchanged between processes), and persistent objects – those which outlive the process that created it. All data objects in the system are accessed using capabilities, called tags, including task instructions, hardware devices, streams, process and system environments, and the task tokens used to schedule instruction sequences for execution. Separate bits in a tag allow a data object to be independently readable, writeable, and executable, controlling an applications access to data using a particular tag.

Tags may be freely copied, passed as parameters, and passed from application to application, but may not be modified or forged by an application.This is enforced by isolating the tags from the application instruction sequences. This also allows the scheduling agent to easily locate the tags used by a task (which represent all non-temporary data accessed by a task) while performing processor selection and data prefetching. The data objects represented by the capabilities are fetched into local memory if necessary and the capability resolved into an address in the local address space before executing a task.

6.1. Scoping

Each data object has a single descriptor structure (which may be replicated per processor node), which is stored in an *environment*. Each process has a separate hierarchy of environments, used to store the data and instruction object descriptors used solely within that process. A system environment is provided for storing persistent objects.

In a dataflow system, the data objects accessed by a task must be clearly identified in order to ensure the availability of the data local to a processor when the task is executed. Unlike a program executing in a single address space, there are no ubiquitous globals. All of the data objects referred to in a task's parameter lists must be assigned and located in memory local to the processor, with appropriate permissions, before a task will be scheduled for execution. There are three lists of data elements accessible by the instructions of a given task: Input/Output, Internal, and Private, each with a different role. Each task is provided with a list of Input/Output parameters, which may be either constants or tags and are used to pass arguments to a task and provide destinations for

the results. They are equivalent to function arguments and results in a conventional system model and are the major component of a task token. Internal parameters are provided to allow a task instruction to access private data objects, not available to the calling task (i.e. supporting protected procedures). They are equivalent to global variables, shared by all instances of a code module in a conventional system model. Since the Internal parameters are bound to a task in general, and not to any particular instance, they must be shared by all callers of a task. The Private parameters allow a shared library task to reference data objects unique to a particular process, yet not accessible to user tasks of that process.

6.2. Persistent storage management

Data objects are loosely divided into two categories: those which are described in a Process or Task environment, and those which are described in the System environment. Objects which are part of a Task or Process environment will be automatically destroyed when their application terminates. In contrast, objects described in the System environment will not be deallocated, and may remain defined indefinitely.

It is intended that these long-term objects – applications, multimedia recordings, or infrastructure – be accessed in several different manners:

- The system environment, and process environments are built and stored in long-term (non-volatile) storage, and loaded into memory as needed. Only a minimal system environment need be provided locally in long-term storage on a processor node, as system resources and all application resources may be fetched from other processors in the system.
- Data objects placed in the system environment for sharing by different processes may migrate onto long-term storage if unused and memory is needed.
- Deliberate disk I/O may be performed by specifying a 'file I/O' task as the source or destination (respectively) of a stream. This method is also used for other physical device I/O. The 'device drivers' consist of publicly executable tasks located in the system environment for driving displays, peripherals, and audio outputs, as well as acquiring video, audio, and sensor or other data.

Deallocation of persistent objects is problematic. Unlike the temporary data objects defined local to a process, it is difficult to determine when a persistent object is no longer needed. We anticipate that – similarly to existing disk filesystems – occasional manual intervention will be necessary to remove unneeded data.

7. Scheduling

While hybrid dataflow does not specify any particular method of implementation, the 'unbounded' nature of media streams being processed precludes the use of static dataflow. We are proposing the use of an extension to tagged-token dynamic dataflow. Instead of the conventional model – where tokens refer to data using a 'tag' which

contains both function and iteration specifiers – a data reference in this system consists of a tag identifying a stream object, along with the dimensions, offset and extent of the data being stored or demanded. Additional explicit dependencies are addressed using the tag of the target task token.

When a program is started, the resource manager receives a dependency graph of tasks and streams. The dependency graph is not explicitly provided, instead it is embodied in the dependencies specified in each stream or task object. These initial tasks, when executed, may in turn create (or reference existing) graphs and present them to the resource manager for execution. The resource manager is responsible for evaluating the graphs presented to it to produce output data. Although the nature of the data being processed introduces real-time constraints, we reject as unnecessarily limited systems which guarantee performance through static scheduling [11]. The method of evaluation selected, eduction, heavily influences the inherent fault-tolerance of the system.

7.1. Eduction

There are two basic methods of evaluating the dependency graph which describes a program: demand driven (or call-by-need) and data driven (call-by-value). A data-driven approach performs a computation as soon as the required input values are available. While this ensures that a computation is performed as soon as possible, it generally results in unneeded computations being performed. Visualize, for example, a program that only wants to output a small region of interest of a large input dataset.

While a demand-driven approach prevents unnecessary computations from being performed, a demand for application output typically sees the complete computational latency of the application before the requested data is available. The demand driven evaluation, or eduction [3], of a dependency graph is described by two rules:

1. The need for an data value at the output of a process causes it to be demanded.
2. If a data object (or a particular sub-context of one) is demanded, then and only then are values demanded that are known directly to determine the data object.

Thus, eduction is simply tagged, demand-driven dynamic dataflow, where the tag includes the multidimensional context of the data.

Exactly determining the appropriate input stream fragment for producing a given output fragment is only possible given a constant rate (deterministic) process. If constant rate, the information used by the resource manager for stream splitting and pipelining (the extent, dimension, and step of stream inputs and outputs to a task), in conjunction with a task history establishing an absolute relationship between the different stream coordinate systems at some point in space–time, is sufficient for calculating which input stream fragments should be queried for a given fragment of output stream. If not a constant rate process, upper and lower bounds may be used to characterize the process rate [6], but a following stream merge will require serial stream reassembly. An example of the latter situation from digital media is variable-length coding, for which a maximum and minimum compression factor are known, but the instantaneous rate will vary with the statistics of the data.

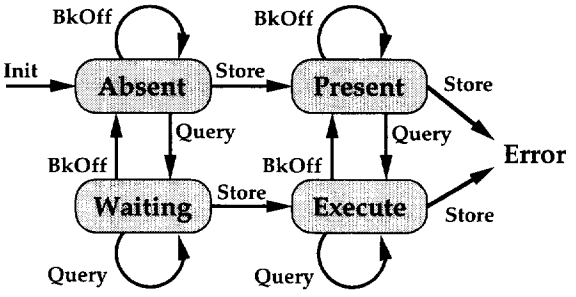


Fig. 2. Stream synchronization state diagram.

If a particular fragment of a stream has been queried (demanded), the process producing that fragment should be executed as soon as possible. At the same time, since a fragment demanded may encompass many of the fragments actually used for scheduling and execution, we provide a mechanism – the `BkOff` message – for signaling that a particular streams buffers are filling up and data is no longer urgently needed.

The synchronization state of a stream fragment follows the state diagram shown in Fig. 2. This is similar to that proposed for an I-structure [2], with an extension to support two priority levels. This is done to avoid the condition of processors sitting idle, by beginning data-driven processing in the absence of data demands. The `Present` state represents a datum that is presently located in memory, but which has not been demanded (via a Query). The `Execute` state represents data for which at least one query has been received. If any tasks have their dependencies cleared, and an output fragment which is `Waiting` or an input fragment which is `Execute`, they will be executed. If no such tasks are ready, tasks with all their Input fragments `Present` are executed. If a `BkOff` message is received for a fragment, the priority of the data is reduced.

7.2. Fault-tolerance

Fault-tolerance is frequently cited as an advantage of distributed systems, but this is perhaps making a virtue out of a feature necessitated by the large number of components in such a system. Data and instructions may be lost or garbled in transmission between processors, and processor nodes may function incorrectly for various reasons. The fault-tolerant nature of functional languages, and eduction in particular, have been noted previously [3,10]. For reasons of brevity we will simply here note that our system is able to detect errors, and (by use of the stream mechanism) recalculate correct data to recover from errors as necessary – a feature perhaps of much greater importance in professional, or media-generation applications than in consumer systems.

8. Requirements for the resource manager

The resource manager is a task distributed across all the processing nodes in a system, both for scalability and fault-tolerance. It is charged with providing the

functions of a traditional operating system, as well as meeting the objectives alluded to previously:

1. It allows an algorithm to be executed on a variety of actual systems by dynamically mapping the algorithm onto the available number and type of processing units.
2. It allows applications executing concurrently to share a limited pool of processor, memory, and communication resources. It should provide graceful degradation of a system (fair access to resources) when overloaded.
3. It performs the run-time partitioning of streams to exploit data parallelism.

The resource manager is responsible for scheduling – deciding at runtime which processing element should be used to execute a given task. This decision is based on how efficiently a particular processing element will perform the task (taking into account any code or data already local to a given processor), the amount of local storage required for efficient pipelining, and the complement and current load of processing elements in the system. In addition to balancing the processing load of an application across a system, this provides a tolerance of faults in individual processing elements.

Long and medium-term (longer than the execution of a single task) memory resources associated with a process are allocated using the resource manager. Streams are a special case of these which may grow dynamically and be stored in a distributed manner. This support allows an algorithm to operate independently of a particular memory architecture – the resource manager will allocate storage in a location it deems optimal for that algorithm and architecture. In addition, automatic deallocation ('garbage collection') of memory objects greatly simplifies the application programmer's task.

The manner in which communication resources are managed varies with machine architecture. Some architectures use communication resources such as shared buses, or packet-switched networks, which rely mainly on fast hardware arbitration. Others use semi-static routing – such as a circuit-switched crossbar – or DMA channels, which must be scheduled. The resource manager both takes the availability of the communication resources into consideration when scheduling algorithm segments, and performs any initialization of communications channels (e.g., configuring the crossbar, or DMA controller) required.

8.1. Distributed resource management

Although a resource manager is capable of 'stand-alone' operation, it is designed to operate in conjunction with resource managers on other processors in a distributed system. The extent of this distributed system is determined when the resource manager initializes itself, and is modified over time to reflect the addition and removal of resources. A node has a list of 'primary nodes' stored locally, and uses it to register itself and obtain information about the local system of which it is a component. This list is not meant to be complete, rather it contains a subset of nodes likely to be operational.

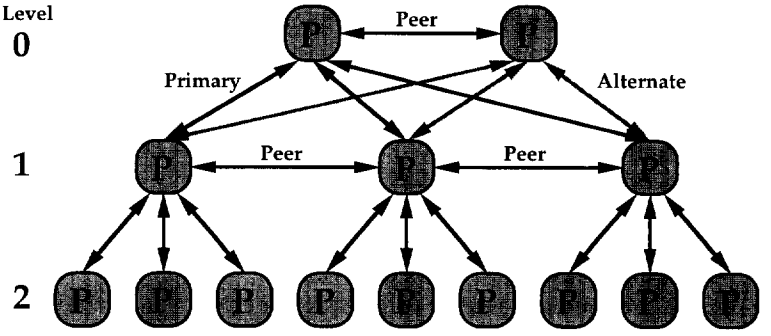


Fig. 3. Logical hierarchy of processors in a system.

The distributed system is organized logically [2] as a tree with no single node at the top (level 0). At each level in the tree below the top, a processor recognizes a *primary* processor in the level above it, an *alternate* in the level above it to be used in case of primary failure, peer processors at the same level, and nodes below it for which it is the primary (Fig. 3).

In an attempt to avoid 'hot spots', this hierarchy may be different for each process in a system, and is defined as part of the process environment. The number of processors in a level cannot be fixed, and varies along with the number of levels to reflect the available process parallelism and number of active nodes in a system.

When the evaluation of a graph of one or more task tokens and streams is requested, the evaluation is performed locally. As opportunities for data and control parallelism become apparent, the tasks will be assigned to additional processing nodes. This assignment utilizes a (possibly erroneous) model of the processing resources in the system. This model is built when a processing node first contacts a system, and is updated as changes (such as the average load on remote machines) are detected.

The assignment of tasks to processing resources is not done in a single step. Rather, the assignment is done hierarchically. When a task is to be applied to a stream, and the stream and task parameters indicate that it may be 'split' to provide parallelism, the task is subdivided and dispatched to processors in the next level of the hierarchy. Upon evaluation, these processors may further subdivide the task and dispatch it to even lower levels.This has two effects:

- The computational and network I/O load of performing the task/stream splitting and assignment is distributed. Larger systems have a larger number of processors at each level of the hierarchy, as well as more levels in the hierarchy.
- The computational resources being modeled at the initial (and more distant from the worker processors) task subdivision are aggregates. When (and if) a finer subdivision is performed in the next step it utilizes a more local (and hopefully more accurate)

[2] The logical organization need not consider the actual network topology. If, however, there is a hierarchy of communication bandwidths, the mapping of logical organization to physical processors should reflect it.

model of available resources. This attempts to ameliorate the effects of errors in the model of system processor resources.

Once the scheduling, or mapping onto currently available processor resources is performed, the graph is executed relatively independently of the originating node. The serial portions of the application will be scheduled for execution on the primary node, but may migrate to alternates (which become primaries and name new alternates) if needed (e.g., due to overloading or failure of the originating node).

9. An example application

A simple example of a stream graph – representing a 2D motion predictive video encoder comprised of three tasks and five streams – is shown in Fig. 4. The square elements in the graph represent tasks, and the circles represent buffers inserted into streams in the graph by the resource manager. A stream may have multiple output ports (each with its own access pattern) utilizing the same stream buffers. The algorithm consists of a task which estimates the 2D planar motion of the 'New' video frame relative to the 'Old' video frame, followed by a Motion Compensation task that immediately selects the region in the 'Old' frame indicated by the motion vector. An error signal is generated and output, along with the estimated motion vectors.

Typical characteristics for each task stream input or output are shown in Table 1. The variable *BS* is used to represent the size (in any dimension) of the blocks used for motion prediction, and *SS* is used to represent the size (in any dimension) of the area searched for the optimal match (typically a small integer multiple of *BS*). For example, the video data input to the stream graph (the output of the In task) is a stream fragment that varies along the x, y, and t (time) dimensions. Its extent in the x dimension is E_x, along the y, E_y, and it is unbounded in time.

The access pattern of the Motion Estimation task, shown graphically in Fig. 5, shows considerable overlap in the 'New' stream between successive task evaluations. Depending on the relative amount of state memory available in the processing element, this task may be pipelined to greatly reduce memory and communications needs. Current specialized motion estimation processors tend to provide enough state for pipelining in only a single dimension, whereas software implementations may attempt to store enough data in local cache for the two dimensional pipelining shown in Fig. 5.

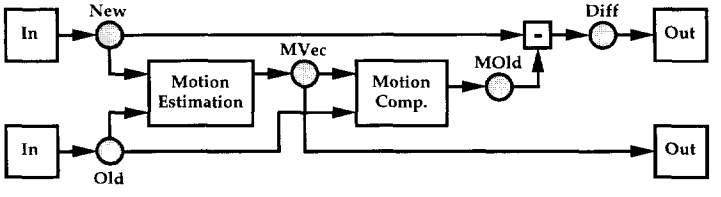


Fig. 4. A 2D motion predictive coder.

Table 1
Motion predictive encoder stream parameters

Task:	New	Old
Port:	output	output
Dimensions	$D\{x, y, t\}$	$D\{x, y, t\}$
Extent	$E\{E_x, E_y, \infty\}$	$E\{E_x + 2BS, E_y + 2BS, \infty\}$
Step	$S\{1, E_x, E_x \times E_y\}$	$S\{1, E_x + SS - BS,$ $(E_x + SS - BS) \times (E_y + SS - BS)\}$
Order	$O\{1, 2, 3\}$	$O\{1, 2, 3\}$

Task:	Motion Estimation			Motion Compensation		
Port:	New	Old	MVec	MVec	Old	MOld
Dimensions	$D\{x, y\}$	$D\{x, y\}$	$D\{s\}$	$D\{s\}$	$D\{x, y\}$	$D\{x, y\}$
Extent	$E\{BS, BS\}$	$E\{SS, SS\}$	$E\{2\}$	$E\{2\}$	$E\{SS, SS\}$	$E\{BS, BS\}$
Step	$S\{BS, BS\}$	$S\{BS, BS\}$	$S\{1\}$	$S\{1\}$	$S\{BS, BS\}$	$S\{BS, BS\}$
Order						

Task:	Subtract	Diff	MVec
Port:	all	input	input
Dimensions	scalar	$D\{x, y, t\}$	$D\{s, t\}$
Extent		$E\{E_x, E_y, \infty\}$	$E\{2, \infty\}$
Step		$S\{1, E_x, E_x \times E_y\}$	$S\{1, 1\}$
Order		$O\{1, 2, 3\}$	$O\{1, 2\}$

The characteristics of the stream buffers are not fixed, instead being determined at runtime to reflect both the capabilities of the machine and the structure of the algorithm. The size of the 'Old' streambuffer, for example, depends rather directly on either the

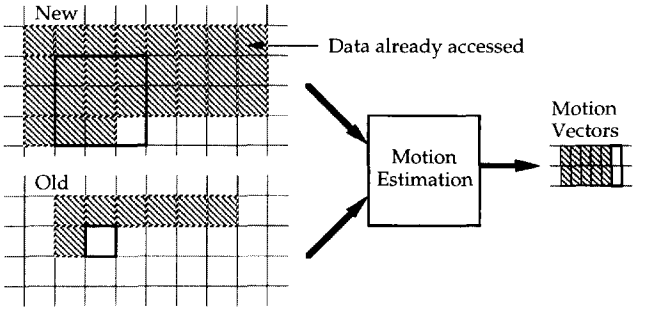


Fig. 5. Stream access patterns of the motion estimation task.

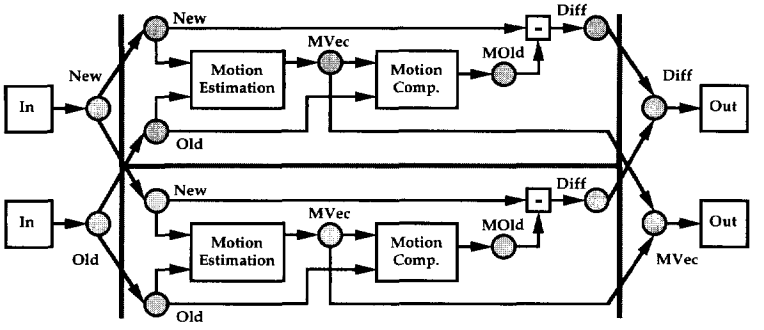


Fig. 6. A partitioning of the motion predictive coder.

computational latency of the Motion Estimation processor (when using fine-grained synchronization) or on the synchronization block size. The 'MOld' stream buffer size is directly determined by the synchronization block size used.

The graph may be subdivided for execution, either if multiple processing elements are available in the system or if the local state on a selected processor is inadequate for optimum pipelining of a given task. A subdivision of the 2D Motion Predictive encoder for a system with two similar nodes is shown in Fig. 6. While both data and process parallelism are available in this example, the chosen partition on a homogenous machine will most likely be data partitioning to minimize communication between processing nodes. As indicated earlier, memory constraints may force partitions smaller than the most efficient to be used.As discussed elsewhere [6], it is also possible to construct dependency graphs which may (due to feedback loops) only be evaluated using synchronization data blocks of limited size.

While stream access overlap allows pipelining, it also increases the communications cost of obtained data parallelism. Partitioning along the t dimension of the input and output streams in this example(where there is no overlap) is also feasible, but given the input and output streams this would result in increased buffer sizes and longer computational latency.

10. Conclusion

Through the stream mechanism we decouple the memory access from the actual processing of the data, simplifying the design of specialized processors and allowing them efficiently to share a single memory interface. In addition, the stream mechanism allows the data and functional parallelism present in an application to be manipulated to match the parallelism available in a particular system.

Digital media provide a computational domain whose demands will continue to increase for many years to come. Because this is after all a consumer application, solutions must be compact, low-cost, and easily programmable, and must support differing hardware scales and architectures. We feel that the system we have outlined here offers a practical framework for design of such products.

Acknowledgements

This work was supported by the Digital Life consortium of the MIT Media Laboratory.

References

[1] Arvind, R.A. Iannucci, Two fundamental issues in multiprocessing, in: Proc. of DFLVR Conf. on Parallel Processing in Science and Eng., 1987. Also in: D.J. Lilja (Ed.), Architectural Alternatives for Exploiting Parallelism, IEEE Computer Society Press, Los Alamitos, CA, 1991.

[2] Arvind, R.E. Thomas, I-structures: An efficient data type for functional languages, Technical report LSC/TM-178, MIT Laboratory for Computer Science, 1980.

[3] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, W.W. Wadge, Multidimensional Programming, Oxford University Press, New York, 1995.

[4] V.M. Bove, Jr., Multimedia based on object models: Some whys and hows, IBM Systems Journal 35 (3–4) (1996).

[5] V.M. Bove, Jr., B.D. Granger, J.A. Watlington, Real-time decoding and display of structured video, in: Proc. IEEE Int. Conf. on Multimedia Computing and Systems '94, Boston, MA, May 1994.

[6] J.T. Buck, E.A. Lee, Scheduling dynamic dataflow graphs with bounded memory using the token flow mode, in: Proc. IEEE 1993 Int. Conf. on Acoustics, Speech and Signal Processing, April 1993, pp. 429–432.

[7] D. Culler, G. Papadopoulos, The explicit token store, Journal of Parallel and Distributed Computing 10 (4) (1990) 289–308.

[8] S.A. Guccione, M.J. Gonzalez, A data-parallel programming model for reconfigurable architectures, in: Proc. IEEE Workshop on FPGAS for Custom Computing Machines, Napa, CA, 1993, pp. 79–87.

[9] R.A. Iannucci, Toward a dataflow/von Neumann hybrid architecture, in: Proc. 15th Annual Int. Symposium on Computer Architecture, ACM, 1988, pp. 131–140.

[10] R. Jagannathan, E.A. Ashcroft, Fault tolerance in parallel implementations of functional languages, in: Proc. 21 th Int. Symp. on Fault-Tolerant Computing, Montreal, Quebec, Canada, June 1991, pp. 256–263.

[11] K.H. Kim, C. Subbaraman, Fault-tolerant real-time objects, Communications of the ACM 40 (1) (1997) 75–82.

[12] M. Kunt, A. Ikonomopoulos, M. Kocher, Second-generation image-coding techniques, Proceedings of the IEEE 73 (4) (1985) 549–574.

[13] R. Lee, Subword parallelism with MAX-2, IEEE Micro 16 (4) (1996) 51–59.

[14] H.M. Levy, Capability-Based Computer Systems, Digital Press, Bedford, MA, 1984.

[15] H.G. Musmann, Object-oriented analysis–synthesis coding of moving objects, Signal Processing: Image Communication 1 (1989) 117–138.

[16] R.S. Nikhil, G.M. Papadopoulos, Arvind, *t: A multithreaded massively parallel architecture, in: Proc. 19th Annual Int. Symposium on Computer Architecture, 1992, pp. 156–167.

[17] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, Y. Koumura, Thread-based programming for EM-4 hybrid dataflow machine, in: Proc. 19th Annual Int. Symposium on Computer Architecture, 1992, pp. 146–155.

[18] P. St.-Hilaire, S.A. Benton, M. Lucente, Synthetic aperture holography: A novel approach to three dimensional displays, Journal of the Optical Society of America A 9 (11) (1992) 1969–1977.

[19] A.S. Tanenbaum, S.J. Mullender, R. van Renesse, Using sparse capabilities in a distributed operating system, in: Proceedings 6th Annual Int. Conf. on Distributed Computing Systems, New York, 1986, pp. 553–563.

[20] M. Tremblay, J.M. O'Connor, V. Narayanan, H. Liang, VIS speeds new media processing, IEEE Micro 16 (4) (1996) 51–59.

[21] J.A. Watlington, V.M. Bove, Jr., Stream-based computing and future television, SMPTE Journal 106 (4) (1997).

[22] J.A. Watlington, M. Lucente, C.J. Sparrell, V.M. Bove, Jr., I. Tamitani, A hardware architecture for rapid generation of electro-holographic fringe patterns, in: SPIE Proc. #2406-23 – Practical Holography IX, Bellingham, WA, 1995.

[23] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Ghosh, PRISM-II compiler and architecture, in: Proc. IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, 1993, pp. 9–16.