Integrated task and data parallel support for dynamic applications

James M. Rehg [a,*], Kathleen Knobe [a], Umakishore Ramachandran [b], Rishiyur S. Nikhil [a] and Arun Chauhan [c]

[a] *Cambridge Research Laboratory, Compaq Computer Corporation, Cambridge, MA 02139, USA*
E-mail: {rehg,knobe,nikhil}@crl.dec.com
[b] *College of Computing, Georgia Institute of Technology, Atlanta GA 30332, USA*
E-mail: rama@cc.gatech.edu
[c] *Computer Science Department, Rice University, Houston, TX 77005, USA*
E-mail: achauhan@cs.rice.edu

There is an emerging class of real-time interactive applications that require the dynamic integration of task and data parallelism. An example is the Smart Kiosk, a free-standing computer device that provides information and entertainment to people in public spaces. The kiosk interface is computationally demanding: It employs vision and speech sensing and an animated graphical talking face for output. The computational demands of an interactive kiosk can vary widely with the number of customers and the state of the interaction. Unfortunately this makes it difficult to apply current techniques for integrated task and data parallel computing, which can produce optimal decompositions for static problems.

Using experimental results from a color-based people tracking module, we demonstrate the existence of a small number of distinct operating regimes in the kiosk application. We refer to this type of program behavior as *constrained dynamism*. An application exhibiting constrained dynamism can execute efficiently by dynamically switching among a small number of statically determined fixed data parallel strategies. We present a novel framework for integrating task and data parallelism for applications that exhibit constrained dynamism. Our solution has been implemented using *Stampede*, a cluster programming system developed at the Cambridge Research Laboratory.

*Corresponding author.

Scientific Programming 7 (1999) 289–302
ISSN 1058-9244 / $8.00

1. Introduction

There is an emerging class of real-time interactive applications that require dynamic integration of task and data parallelism for effective computation. The Smart Kiosk system [22,5] under development at the Cambridge Research Laboratory (CRL) is an example which motivates this work. Kiosks provide public access to information and entertainment. The CRL Smart Kiosk supports natural, human-centered interaction. It uses camera and microphone inputs to drive the behavior of a graphical talking face which speaks to its customers.

The CRL Smart Kiosk has features that we believe are typical of an emerging class of scalable applications. It is both reactive and interactive, since it must respond to changes in its environment as new customers arrive and it must interact with multiple people. It is computationally demanding due to the need for real-time vision, speech, and graphics processing. It is also highly scalable, both at the task level in supporting a variable number of users and functions, and at the data level in processing multiple video and audio streams.

Effective utilization of the task and data parallelism inherent in the CRL Smart Kiosk is critical for its successful implementation on commodity hardware such as clusters of SMPs. Unfortunately, existing techniques for integrated task and data parallel computing [2,4,9, 19,3] are not well-suited to this type of application. As an example of the current state-of-the-art, the system described in [19] employs extensive off-line analysis of the memory requirements, communication patterns and scalability aspects of the tasks. This information is then used to make a static determination of the ideal distribution of work among processors.

In contrast, the computational requirements of even simple vision algorithms for the kiosk are proportional to the number of customers, which cannot be predicted in advance. This variability has a direct influence on the optimal resource assignment, as we demonstrate in Section 6. Furthermore, computational requirements

that are unrelated to sensing tasks (such as database access, remote content retrieval, and conversation with the customer) will also vary as customers use the kiosk, impacting the resources available for sensing. Some previous work has been done on the dynamic integration of task and data parallelism for scientific applications [3]. However, that work focused on parallel numerical algorithms such as can be found in ScaLAPACK.

In this paper, we describe a novel approach to the dynamic integration of task and data parallelism for interactive real-time applications like the CRL Smart Kiosk. This work takes place within the context of *Stampede*, a cluster programming system under development at CRL. Stampede is aimed at making it easy to program this emerging class of applications on clusters of SMPs, the most economically attractive scalable platform. Stampede provides a task-parallel substrate: Dynamic cluster-wide threads together with Space-Time Memory, a high-level, flexible mechanism by which threads can communicate time-sequenced data such as video frames.

We present a general framework for integrating data parallelism into a task parallel substrate such as Stampede. It is based on an architecture for embedding data parallel decompositions into a task graph. The architecture supports dynamic, on-line changes in the data parallel strategy. We introduce a new notational scheme for describing these embedded data parallel architectures. In addition, we discuss an approach to changing the data parallel strategy during execution in response to changes in the application.

We show experimental results for a color-based tracking task that demonstrate the existence of distinct operating regimes in the kiosk application. Within each regime, a different data parallel strategy is required for optimal performance. We say that the kiosk exhibits *constrained dynamism*, since the optimal strategy changes during execution, but only at the boundaries of a small number of operating regimes. Our framework for integrated task and data parallelism is ideally suited to applications with constrained dynamics.

In Section 2 we describe the CRL Smart Kiosk application which provides the motivation for this research. We present the Stampede system in Section 3. Sections 4 and 5 form the core of paper. They contain our framework for dynamic integration of task and data parallelism. Experimental results demonstrating the existence of constrained dynamism can be found in Section 6.

2. The CRL Smart Kiosk: A dynamic multimedia application

This work is motivated by the computational requirements of a class of dynamic, interactive computer vision applications. We introduce this class through a specific example: A vision-based user-interface for a Smart Kiosk [22,17] under development at the Cambridge Research Laboratory. A Smart Kiosk is a free-standing computerized device that is capable of interacting with multiple people in a public environment, providing information and entertainment.

Conventional kiosks, such as ATM machines, are based on a touch-screen interface. The market for these kiosks is currently growing rapidly. We are exploring a social interface paradigm for kiosks. In this paradigm, vision and speech sensing provide user input while a graphical speaking agent provides the kiosk's output. Results from a recent public installation of our prototype kiosk can be found in [5]. A related kiosk application is described in [7].

Fig. 1 shows a picture of the Smart Kiosk prototype. The camera at the top of the device acquires images of people standing in front of the kiosk display. The kiosk employs vision techniques to track and identify people based on their motion and clothing color [17]. The estimated position of multiple users drives the behavior of an animated graphical face, called DECface [21], which occupies the upper left corner of the display.

Vision techniques support two kiosk behaviors which are characteristic of public interactions between humans. First, the kiosk greets people as they approach the display. Second, during an interaction with multiple

Fig. 1. The Smart Kiosk.

users DECface exhibits natural gaze behavior, glancing in each person's direction on a regular basis. Future versions of the kiosk will include speech processing and face detection and recognition.

There is currently a great deal of interest in vision- and speech-based user-interfaces (see the recent collections [6,8]). We believe the Smart Kiosk to be representative of a broad class of emerging applications in surveillance, autonomous agents, and intelligent vehicles and rooms.

2.1. Computational properties

A key attribute of the Smart Kiosk application is the real-time processing and generation of multimedia data. Video and speech processing combined with computer graphics rendering and speech synthesis are critical components of a social interface. The number and bandwidth of these data streams results in dramatic computational requirements for the kiosk application. However, there is both significant task parallelism as a result of the loose coupling between data streams and significant data parallelism within each data stream. These sources of parallelism can be exploited to improve performance. However, the complex data sharing patterns between tasks in the application make the development of a parallel implementation challenging.

One source of complexity arises when tasks share streams of input data which they sample at different rates. For example, a figure tracking task may need to sample every frame in an image sequence in order to accurately estimate the motion of a particular user. A face recognition task, in contrast, could be run much less frequently. Differences in these sampling rates complicate the recycling and management of the frame buffers that hold the video input data.

The dynamics of the set of tasks that make up the kiosk application is a second source of complexity. These dynamics are a direct result of the interactive nature of the application. A task such as face recognition, for example, is only performed if a user has been detected in the scene. Thus, whether a task in the application is active or not can depend upon the state of the external world and the inputs the system has received. This variability also complicates frame buffer management.

2.2. Color-based tracking example

The Smart Kiosk application can be viewed as a dynamic collection of tasks that process streams of input data at different sampling rates. To explore this point further, we focus on a subpart of the Smart Kiosk application that tracks multiple people in an image sequence based on the color of their shirts.

Fig. 2 shows the task graph for a color-based person tracking algorithm taken from [17]. It was used in our first Smart Kiosk prototype [22]. It tracks multiple people in the vicinity of the kiosk by comparing each video frame against a set of previously defined histogram models of shirt colors. There are four distinct tasks: *digitizer*, *change detection*, *histogram*, and *target detection*, which are shown as circular nodes in the diagram. The inputs and outputs for these tasks are shown as cylindrical "pipes". For example, the *histogram* task reads video frames and writes color models (histograms). The *target detection* task is based on a modified version of the standard histogram intersection algorithm described in [20].

Fig. 3 illustrates the flow of data in the color tracker by following a single image through the task graph. Processing begins at the *digitizer* task, which generates the video frame shown in Fig. 3(a). The *change*

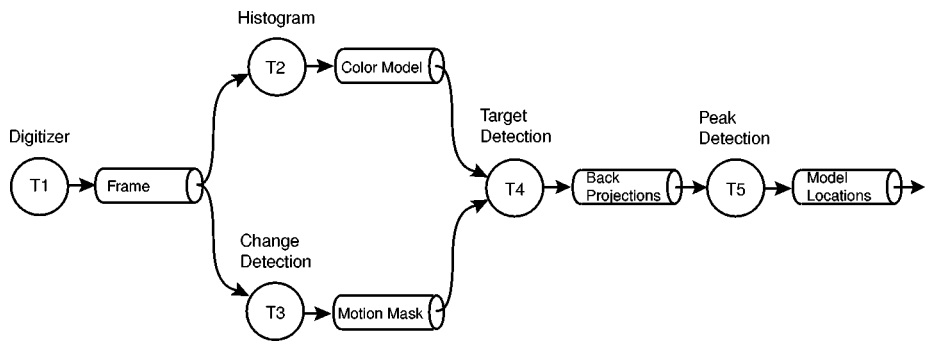

Fig. 2. Task graph for the color-based tracker. Circles denote *tasks*, implemented as threads. Cylindrical "pipes" denote *channels* which hold streams of data flowing between tasks.

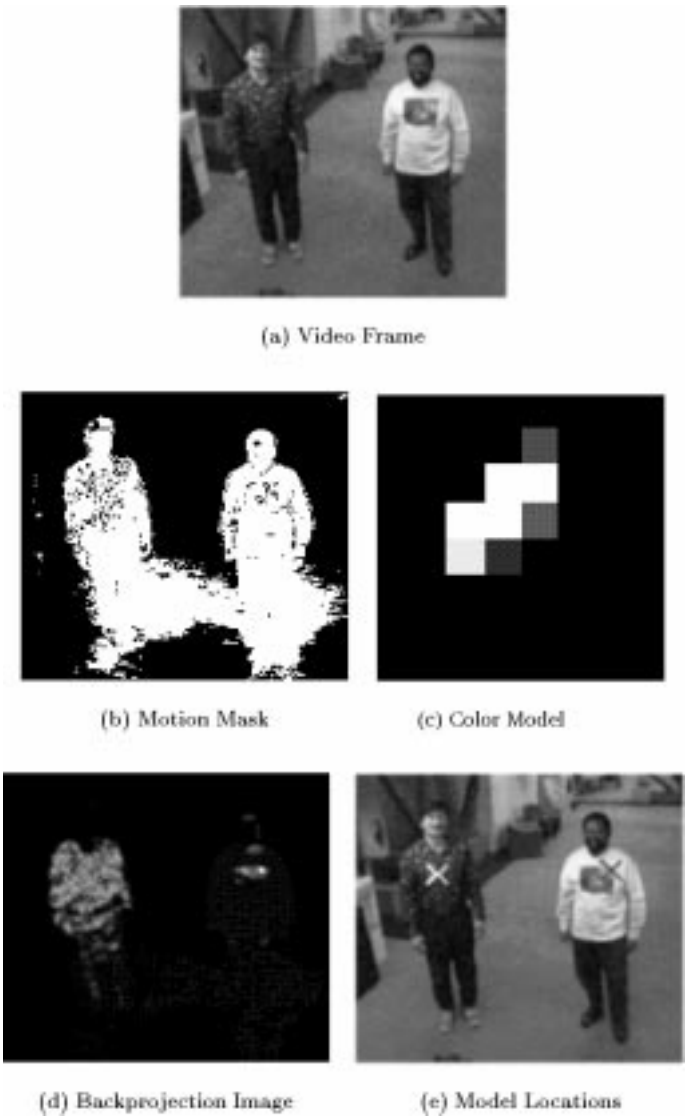

Fig. 3. Data flow in the color tracker of Fig. 2 during a search for two models, corresponding to the two figures in the input frame (a). The final output is the positions of the detected targets in (e). Intermediate results are shown in (b)–(d).

detection task subtracts a previously acquired background image from this frame to produce a motion mask (b) showing the foreground objects and their shadows. Similarly, the *histogram* task produces a color model (c) of the video frame (a). Fig. 3(c) shows an 8 by 8 bin histogram of the normalized red and green pixel values from the input frame, rendered as an intensity image. The brightness of each square bin reflects the number of pixels it contains.

Each instantiation of the *target detection* task compares the image histogram in Fig. 3(c) to a previously defined model histogram, resulting in a backprojection image. There is a separate backprojection image for each model. Fig. 3(d) shows the sum of the two backprojection images for the two targets. Each pixel in a backprojection image encodes the likelihood that its corresponding image pixel belongs to a particular color model. Connected component analysis and peak detection on the smoothed backprojection images results in the detected positions of the two models, shown with crosses in (e).

Parallelism at both the task and the data level are visible in the diagram of Fig. 2. Task parallelism arises when distinct tasks can be executed simultaneously. It is most obvious in the *change detection* and *histogram* tasks, which have no data dependencies and can there-

fore be performed in parallel. It is also present in the form of pipelining, where for example the *histogram* and *target detection* tasks can be performed simultaneously on different frames of an image sequence.

Data parallelism occurs when a single task can be replicated over distributed data. The *target detection* task is data parallel, since it performs the same operation for each color model in the application. The search for a set of models can be performed in parallel by multiple instances of the *target detection* task. For example, Fig. 2 illustrates a parallel search for two models. Similarly, data parallelism at the pixel level can be exploited in many image processing tasks, such as *change detection* or *histogram*, by subdividing a single frame into regions and processing them in parallel.

In designing a parallel implementation of the color tracker we could focus on task parallelism, data parallelism, or some combination of the two. Our experimental results in Section 6 confirm our hypothesis that in this application's dynamic environment, we need to combine task and data parallelism and, moreover, that the combined structure must vary dynamically.

3. Stampede

In this section we briefly describe Stampede, is the programming system within which we explore integrated task and data parallelism. (A more detailed description may be found in [15,14,18].) Stampede is currently based entirely on C library calls, i.e., it is implemented as a run-time system, with calls from standard C.

Stampede extends the well-known POSIX dynamic threads model [11] from SMPs to clusters of SMPs, which constitute the most economically attractive scalable platform today. It provides various "shared-memory" facilities for threads to share data uniformly and consistently across clusters.

More pertinent to the current discussion, Stampede provides a high-level, concurrent, distributed data structure called *Space-Time Memory* (STM) [18], which allows threads to produce and consume time-sequenced data in flexible ways, addressing the complex "buffer management" problem that arises in managing temporally indexed data streams as in the Smart Kiosk application. There are four sources of this complexity:

- Streams become temporally sparser as we move up the analysis hierarchy, from low-level vision processing tasks to high-level recognition tasks.
- Threads may not access items in strict stream order.
- Threads may combine streams using temporal correlation (e.g., stereo vision, or combining vision and sound).
- The hierarchy itself is dynamic, involving newly created threads that may re-examine earlier data.

Traditional data structures such as streams, queues and lists are not sufficiently expressive to handle these features.

Stampede's Space-Time Memory (STM) is our solution to this problem. The key construct in STM is the *channel*, which is a location-transparent collection of objects indexed by time. The API has operations to create a channel dynamically, and for a thread to *attach* and *detach* a channel. Each attachment is known as a *connection*, and a thread may have connections to multiple channels and even multiple connections to the same channel.

Fig. 4 shows an overview of how channels are used. A thread can *put* a data item into a channel via a given output connection using the call:

```
spd_channel_put_item (o_connection, timestamp,
                      buf_p, buf_size, ...)
```

The item is described by the pointer `buf_p` and its `buf_size` in bytes. Although multiple channels may contain items with the same timestamp at the same time, a given channel can contain only one item with a given timestamp. But this constraint does not imply that items be put into the channel in increasing or contiguous timestamp order. Indeed, to increase throughput, a module may contain replicated threads that pull items from a common input channel, process them, and put items into a common output channel. Depending on the relative speed of the threads and the particular events they recognize, it may happen that items are placed into the output channel "out of order". Channels can be created to hold a bounded or unbounded number of items. The `put` call takes an additional flag that allows it to block or to return immediately with an error code, if a bounded output channel is full.

A thread can *get* an item from a channel via a given connection using the call:

```
spd_channel_get_item (i_connection, timestamp,
                      & buf_p, & buf_size,
                      & timestamp_range, ...);
```

The `timestamp` can specify a particular value, or it can be a wildcard requesting the newest/oldest

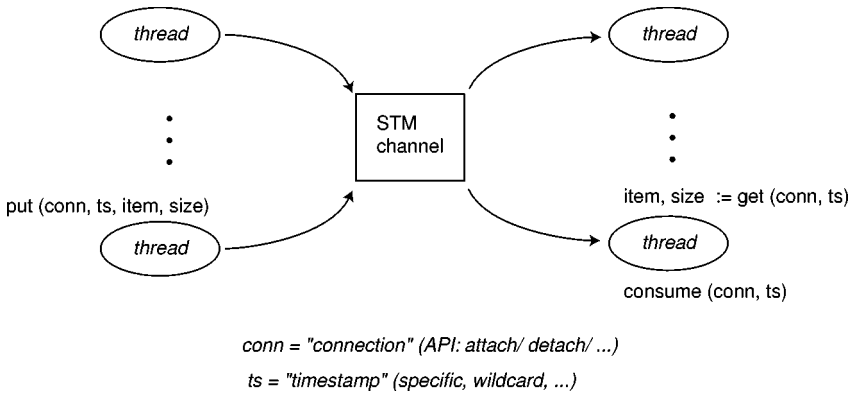

Fig. 4. Overview of Space-Time Memory channels in Stampede.

value currently in the channel, or the newest value not previously gotten over any connection, etc. As in the `put` call, a flag parameter specifies whether to block if a suitable item is currently unavailable, or to return immediately with an error code. The parameters `buf_p` and `buf_size` can be used to pass in a buffer to receive the item or, by passing NULL in `buf_p`, the application can ask Stampede to allocate a buffer. The `timestamp_range` parameter returns the timestamp of the item returned, if available; if unavailable, it returns the timestamps of the "neighboring" available items, if any.

The `put` and `get` operations are atomic. Even though a channel is a distributed data structure and multiple threads across the cluster may simultaneously be performing operations on the channel, these operations appear to all threads as if they occur in a particular serial order.

The semantics of `put` and `get` are copy-in and copy-out, respectively. Thus, after a `put`, a thread may immediately safely re-use its buffer. Similarly, after a successful `get`, a client can safely modify the copy of the object that it received without interfering with the channel or with other threads. Of course, an application can still pass a datum by reference – it merely passes a reference to the object through STM, instead of the datum itself. The notion of a "reference" can be based on any of the "shared-memory" mechanisms supported by Stampede, described in more detail in [14].

Puts and gets, with copying semantics, are of course reminiscent of message-passing. However, unlike message-passing, these are location-independent operations on a distributed data structure. These operations are one-sided: there is no "destination" thread/process in a `put`, nor any "source" thread/process in a `get`. The abstraction is one of concurrently putting items into and getting items from a temporally ordered collection, not of communicating between processes.

A related conception of space-time memory has been used in optimistic distributed discrete-event simulation [12,10]. In these systems, space-time memory is used to allow a computation to roll-back to an earlier state when events are received out of order. In contrast, we have proposed STM as a fundamental building block for a distributed application. Additional information about STM and related work can be found in [18,15].

4. Integration of task and data parallelism

In this section, we address the integration of task and data parallelism in the context of the Stampede system. We will discuss both static and dynamic integration strategies, using the color tracker application from Fig. 2 as an illustrative example.

For tasks like the color tracker there is a basic performance tradeoff between latency (input to output time per single frame) and throughput (output frames per unit time). Since the digitizer can generate images significantly faster than the downstream tasks can process them, pipeline parallelism can be used to increase throughput. Alternatively, a data parallel implementation of the target detection task can be used to remove bottlenecks and reduce latency. The integration of task and data parallelism makes it possible to address latency and throughput in a single framework.

Task parallelism is captured in a natural fashion by the task graph representation used in Space-Time Memory.[1] We introduce a framework for data parallel

[1] As a result, a task parallel implementation of an application can be achieved simply by implementing it in Stampede and assigning each task to a separate thread.

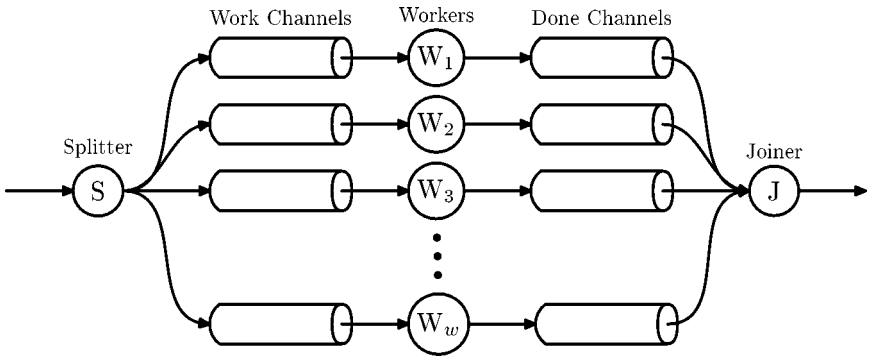

Fig. 5. Static data parallel architecture.

execution of a task in the STM environment. The basic idea is to replace a node in the task graph with a data parallel subgraph. This subgraph implements the work of the original node using multiple worker nodes operating in parallel. Each worker performs the same task as the original node but operates on a partition of the data.

We will describe the general form of the data parallel subgraph which can be applied to any potentially data parallel node in the task graph. The details of its operation will depend upon the application. For example, the color tracker operates on two data types, image frames and target color models. The tracker data space can be characterized as the cross product of frames, pixels in each frame, and models. Correspondingly, there are three possible strategies for exploiting data parallelism: distribute distinct whole frames, distribute parts of the same frame (i.e., regions of pixels), and distribute models.

Distributing distinct frames increases throughput but has no impact on per frame latency. Distributing regions and models would reduce latency. In distributing parts of the same frame, each data parallel thread searches in distinct regions of the frame for all of the models. Alternatively, in distributing models, each thread searches the entire frame for a distinct subset of target models. Combinations of these two approaches result in searching distinct regions of the frame for a subset of the models.

4.1. Static data parallel architecture

As an introduction to our framework, we first consider a static data parallel architecture. It has three basic components: *splitter*, *worker*, and *joiner* threads. The structure is illustrated in Fig. 5. For some task T, w data parallel worker threads execute the task concurrently, each on approximately one wth of the data. The splitter thread reads from the input channels for task T and converts the *chunk* of work specified by the inputs to T into w data parallel chunks, one for each of the workers. The joiner combines the w partial results from the workers into a single result, which it places on T's output channels. The splitter and joiner threads provide the interface between the worker threads and the rest of the task graph. They ensure that the data parallelism within T is not visible to the rest of the application.

The extent of the data parallelism employed is determined by the number of workers. A worker thread is a parameterized version of the original thread which has been designed to process a specific partition of the data. In the color tracker example, workers process fixed combinations of regions and models. Note that in the case where whole frames are distributed the worker threads can be direct copies of the original thread: The splitter simply reads frames and distributes them to the workers, and the joiner places the processed frames on its output channel.

The data parallel approach of Fig. 5 is *static* in that there is a fixed assignment of chunks to workers and a fixed number of worker threads. Note however that the splitter does not have to wait until one set of chunks has been completed before sending the next set of chunks to the workers.

4.2. Dynamic data parallel architecture

The static assignment of chunks to workers is unnecessarily restrictive. It limits the flexibility of the splitter to respond to changes in the task and makes it difficult to vary the number of workers. In the color tracking application, for example, the splitter's strategy should vary with the number of targets, as we demonstrate experimentally in Section 6.

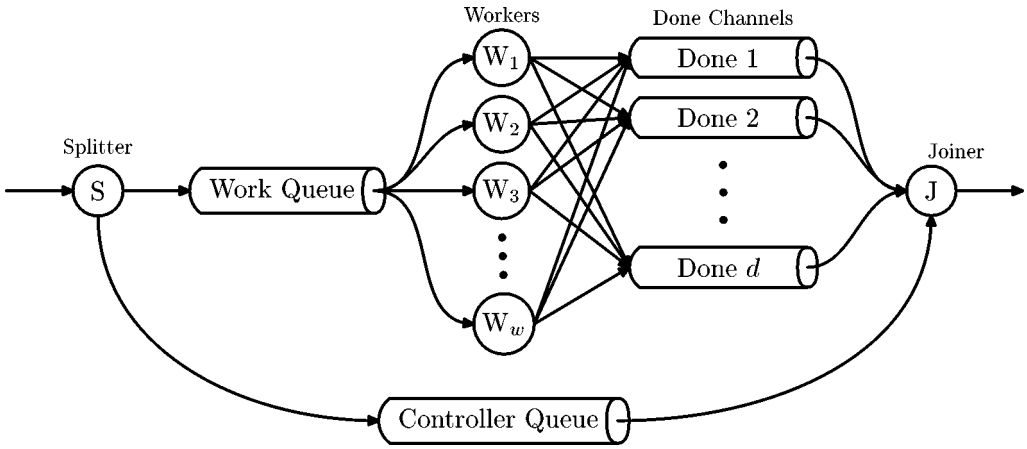

Fig. 6. Dynamic data parallel architecture.

Fig. 6 illustrates a dynamic data parallel architecture that avoids the limitations of the static approach. Here a single *work queue* acts as the source of chunks for all of the worker threads, supporting the dynamic assignment of chunks based on worker availability. The work queue provides load balancing in the situation where the number of chunks does not equal the number of workers. This strategy further minimizes latency when the time to complete a chunk varies over time. It also makes it easier to vary the number of worker threads, w, during execution.

The splitter divides an item of work into M chunks. In contrast to the static case, we no longer require $M = w$. In fact, M may vary with each item. In the static case, the joiner knew the number of chunks for each item and where to find them. Here the splitter communicates its data parallel strategy for each item (e.g., by frames, regions, models, or a combination) including the number of chunks, to the joiner through the *controller queue*.

The workers communicate with the joiner through the *done channels* illustrated in Fig. 6. The splitter tags each chunk with its associated done channel (e.g., chunk i goes to done channel i). This mechanism allows any worker to process any chunk since the done channels act as a sorting network for the results. After receiving the strategy from the splitter, the joiner reads the partial results from the appropriate done channels, combines them, and outputs the complete result. Note that the number of allocated done channels, d, is an upper bound on the number of chunks generated by splitter for any item.

In contrast to the static case, this architecture gives the splitter the ability to dynamically alter its strategy in response to changes in the task. A framework for changing strategies is described in Section 4.4.

4.3. Data parallel notation

The general *data parallel architecture* above is used to transform a task parallel application into an integrated task and data parallel version which is called the *DP application architecture*. This process could be specified by replacing one or more nodes in the original task graph by the subgraph of Fig. 6. As a notation, this graphical representation quickly becomes unwieldy due to the visual complexity of the expanded task graph. As an alternative, we introduce a new *data parallel notation* that concisely describes the data parallel aspects of the DP application architecture. Using our notation, the replacement of a node T by a data parallel subgraph is written $\{^{w}\ \mathrm{T}\ ^{d}\}$. Here T identifies the task, w indicates the number of workers, and d indicates the number of done channels. For example, $\{^{8}\ \mathrm{T}\ ^{4}\}$ denotes a data parallel structure with 8 workers and 4 done channels. The open and close brackets can be thought of as representing the splitter and joiner respectively.

Notice that this notation only specifies the DP application architecture. It does not describe the strategy used by the splitter in processing an item. For example, it does not specify how many chunks an item will be divided into or what those chunks correspond to. The splitter can dynamically modify its strategy based on the items it receives. It is constrained only by the number of available workers and done ports, as specified in the notation. For instance, the number of done ports d is an upper bound on the number of chunks.

Up to now we have viewed the general data parallel architecture in Fig. 6 as a means of *replicating* a single node in an existing task graph into multiple worker nodes contained between a splitter and a joiner. Note,

however, that any single-entry/single-exit subgraph of the original task graph can be replicated in the same manner. We will use the term *replica* to refer to such a replicated subgraph. The entry node of a replica will remove items from the work queue. The exit node will put the result on the appropriate done channel. Each replica is simply a copy of tasks and channels from the original task graph that is designed to work on some partition of the data.

In order to describe replicas using our data parallel notation, we first need a method for specifying subgraphs. We denote the subgraph with entry node T_i and exit node T_j as T_i::T_j. Then a DP application architecture containing w replicas of this subgraph can be written $\{^{w}\ T_i::T_j\ ^{d}\}$. Note that this form is not self-contained, but assumes that the description of the subgraph from T_i to T_j is available from the original task graph. For example, given Fig. 2 the color tracker can be written T1::T5.

In our earlier discussion of replication we made no distinction between replication within an SMP and replication across nodes in the cluster. This distinction has a critical effect on performance. It is expressed in our data parallel notation as follows: The curly brackets introduced above indicate that replicas are on distinct nodes. Alternatively, square brackets indicate that replication occurs within a given SMP. An example of the latter is $[^{w}\ T\ ^{d}]$. Table 1 gives a summary of our notation, including some examples of hierarchical replication which are described in Section 5.

4.4. Data parallel strategy

We have described the general data parallel architecture and a notation for specifying its use to create a DP application architecture. In the context of the color tracker, we have presented several data parallel *strategies* which our architecture supports (e.g., distributing regions or models). We will now describe an approach to changing the strategy on-line in response to changes in the application.

We define the *state* of the system to be the set of variables that determine the choice of data parallel strategy. In the color tracker example, the state is simply the number of people being tracked.

The key observation is that our class of applications is not arbitrarily dynamic. It exhibits a characteristic which we call *constrained dynamism* that is defined by the following properties:

(1) Changes in state are infrequent. More precisely, the number of items processed between state changes is large.
(2) The number of distinct states is small.
(3) Changes in state can be detected.

The first property implies that the splitter will need to change strategies infrequently. As a result, the overhead associated with an on-line change in strategy can be amortized over long sequences of processed items. The second property implies that it is feasible to specify the optimal data parallel strategy for each state. The optimal strategy for each state can be determined through off-line analysis. This results in a *look-up table* which can be used by the splitter during execution to select the correct data parallel strategy. The third property ensures that the splitter can react to on-line changes in the system state. For example, in the color tracker application the splitter would be informed whenever a person entered or left the scene and could change strategies if necessary.

Note that measurements of the system state may derive from computations performed by the application on its inputs. This can result in latency between the time at which the state changes and the time at which the change becomes known to the splitter. For example, in the color tracker the number of people in a given frame is not known until after it has been processed! If this number changed with each frame it would be impossible to select the optimal strategy. However, property one ensures that on average only a small number of items will be processed using a suboptimal strategy. The color tracker possesses this property in most situations.

To summarize, we propose to enumerate all possible states and for each state to determine the ideal data parallel strategy during off-line analysis. This information is encoded in a look-up table. The splitter's strategy is based on the measured state. When the state changes, the splitter interrogates the table to determine the best strategy for the new state. The splitter adopts this strategy until the next state change. The splitter communicates the strategy it is using for each item to the joiner via the control channel.

Finally, we note that although our presentation in this section took place in the context of the Stampede system, our approach to integrating task and data parallelism is quite general. It is applicable to any macro dataflow pipeline architecture, such as the *AVS \Express* visualization system from Advanced Visual Systems, Inc. [1].

Table 1
Summary of data parallel notation

Example	Interpretation
$[^{w}\ \mathrm{T}\ ^{d}]$	Replication of task T within an SMP w is the number of workers d is the number of done channels
$\{^{w}\ \mathrm{T}\ ^{d}\}$	Replication of task T across nodes of a cluster
$[^{w}\ \mathrm{T}_i{::}\mathrm{T}_j\ ^{d}]$	Replication of a subgraph within an SMP T_i is the entry node of the subgraph T_j is the exit node of the subgraph
$\{^{w_1}\ [^{w_2}\ \mathrm{T}\ ^{d_2}]\ ^{d_1}\}$	Hierarchical replication of a single task T
$\{^{w_1}\ \mathrm{T}_i{:}\ [^{w_2}\ \mathrm{T}_m{::}\mathrm{T}_n\ ^{d_2}]\ {:}\mathrm{T}_j\ ^{d_1}\}$	Hierarchical replication of subgraph T_m to T_n within a replication of subgraph T_i to T_j

5. Hierarchical decompositions

The approach to integrating data-parallel computations described in Section 4 has a natural, recursive extension. Networks of the form of Fig. 6 can be nested recursively, creating layered replications. These more complex topologies arise naturally from attempts to reduce latency or increase throughput by applying data parallelism to bottle-neck tasks.

A particularly natural hierarchical decomposition for the color tracker is shown in Fig. 7. This decomposition is designed to improve both latency and throughput in a cluster setting. We will introduce the general idea of hierarchical decomposition through this example. The figure illustrates the direct incorporation of the data parallel notation into the task graph, thereby specifying the integration of task and data parallelism. Note that this is a self-contained description of the DP application architecture.

In the example of Fig. 7, the digitizer creates a sequence of frames which are distributed across nodes in the cluster. This distribution is denoted by the left curly bracket, where w_1 is the number of nodes. Each node will process approximately $1/w_1$ of the frames. This outer replication addresses throughput. Notice that since this replication does not divide up a given item (i.e., frame) there is no need for a joiner task or a controller channel. All of the outputs are sent to a single done channel, in this case model locations. This is denoted by the right curly bracket for which $d_1 = 1$. The STM will order the items on the done channel based on their time-stamps, effectively acting as a joiner in merging the outputs from distinct nodes.

All of the tasks contained within the outer curly brackets in Fig. 7 are replicated across nodes in the cluster. In Section 4.3 we observed that any single-entry/single-exit subgraph can be replicated.[2] Minor modifications may be required to convert an arbitrary subgraph to single-entry/single-exit form. In this example, the original subgraph had two entry nodes, T2 and T3 (see Fig. 2). By adding the *auxiliary node* T_a (which is shaded in the figure) and an associated channel we can easily convert the original subgraph to single entry form. This outer replication can be written using our data parallel notation as $\{^{w_1}\ \mathrm{T}_a{::}\mathrm{T5}\ ^{d_1}\}$.

Continuing with our example, we can further improve performance by addressing the latency within a node. This involves a further decomposition within the replicated subgraph T_a::T5. The bottleneck for this subgraph (and the color tracker as a whole) is the target detection task, T4. Pixel-level data parallelism in this task can be exploited within each SMP. This results in the inner replication $[^{w_2}\ \mathrm{T4}\ ^{d_2}]$ where w_2 and d_2 are the usual parameters. This replication is illustrated with square brackets in Fig. 7. The final result is a hierarchical decomposition with two levels of data parallelism. It can be written $\{^{w_1}\ \mathrm{T}_a{:}\ [^{w_2}\ \mathrm{T4}\ ^{d_2}]\ {:}\mathrm{T5}\ ^{d_1}\}$.

Another example of a hierarchical decomposition, which we will discuss further in Section 6, is T1: $\{^{w_1}\ [^{w_2}\ \mathrm{T4}\ ^{d_2}]\ ^{d_1}\}$:T5. This minimum latency solution distributes the single bottleneck task, T4, both across and within nodes. It is interesting because while the outer splitter sends items to a work queue as usual, the task that reads those items is not an application task as in the previous example. Instead, because the two repli-

[2] Although we require a single task at entry and exit of the replicated subgraph, we allow multiple channels to the entry and from the exit. These can be handled straightforwardly. For example, the splitter can get an item from each of n channels and put one item that is an n-tuple onto the work queue.

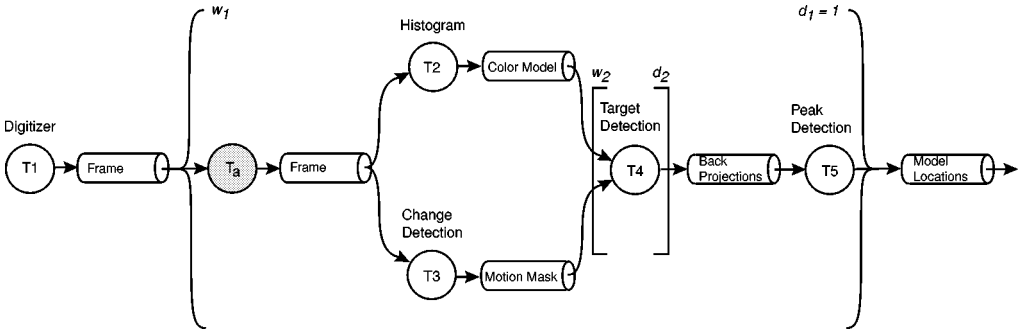

Fig. 7. Hierarchical decomposition of the color tracker in Fig. 2.

cations are tightly nested, the reader of the outer work queue is another splitter which writes to an inner work queue. One might think that this DP application architecture is equivalent to one splitter with w_1 * w_2 workers. In reality this two level decomposition can be much more effective due to the cost difference between inter- and intra-node communication. For example, the inner splitter communicates with its w_2 workers using hardware shared memory, while the outer splitter uses the cluster interconnect.

Finally, we note that the hierarchical examples in this section demonstrate the flexibility of the data parallel notation introduced in Section 4.3. A summary of this notation can be found in Table 1.

6. Experimental results

We implemented the color tracker application from Section 2.2 within the Stampede framework and conducted two sets of experiments. The goal of these experiments was to demonstrate the need to vary the data parallel strategy dynamically in order to obtain optimum performance. We show that for both one and two node (four and eight processor) implementations of the color tracker, the optimal data parallel decomposition depends directly on the number of targets being tracked. We establish this result by measuring the performance of the application as a function of the strategy and the number of models on two different DP application architectures.

We conducted these experiments on a two node cluster of AlphaServer 4100's (four processor SMPs) interconnected by Memory Channel and running DIGITAL UNIX 4.0. STM channels were used to implement the communication between the splitter, worker, and joiner threads in the general data parallel architecture of Fig. 6. In our current implementation these threads are explicitly created at the application level. In future work we plan to automate the generation of these subgraphs using a high-level description of the data parallel strategy.

6.1. SMP experiment

The first experiment focused on data parallel strategies for the target detection task, T4, within a single SMP. This task is the performance bottleneck in the color tracker (see Fig. 2). The cost of target detection results from the histogram backprojection algorithm [20] which generates the back projection images depicted in Fig. 3(e). In histogram backprojection, each pixel in the input image is compared to each of the target models, which are specified by color histograms. This comparison step results in a set of images, one for each target, in which each pixel location has been labeled with the likelihood that it came from that target model. After smoothing these images to reduce the effects of image noise, connected components analysis is used to identify blobs corresponding to the location of each target in the image.

In parallelizing the backprojection algorithm for a single frame we can divide the data by target models and by image regions. The number of target models varies as customers appear and disappear. For a small number of target models, distribution over regions is the only option, but as the number of models increases there is a choice. We would expect distributing over regions to result in increased overhead over processing whole frames. For example, there are certain set-up costs involved in processing a region of pixels regardless of its size. In addition, there are book-keeping costs involved in partitioning a frame of pixel data into regions for processing.

The DP application architecture used in this experiment can be written [^{4} T4 ^{d}] (see Section 4.3),

Table 2
Timing results in seconds/frame for the target detection task, T4, with one and eight target models

	Total models		
	1	8	
Partitions	MP = 1	MP = 8	MP = 1
FP = 1	0.876 (1)	1.857 (8)	6.850 (1)
FP = 4	0.275 (4)	2.155 (32)	2.033 (4)

where there is one worker for each of four CPU's and the number of done channels d equals the number of chunks in the data parallel strategy. We implemented four different strategies in order to explore the effect of partitioning by models and partitioning by regions.

Table 2 gives the results for these trials. It shows the total time to detect all targets in a single frame. There were two regimes, in which the total number of models was one and eight. MP gives the number of partitions of the models and FP the number of partitions of the frames. The total number of chunks in each trial (shown in parentheses) is the product of MP and FP. For example, in the case MP $=$ 8 and FP $=$ 1 each of eight chunks searched for a single model across the entire frame.

In the case where the total number of models is one (first column), we tested the sequential approach of searching the entire frame in one chunk (FP $=$ 1) against the data parallel strategy of dividing the frame across four chunks (FP $=$ 4). The parallel approach was faster, as expected, by more than a factor of three.

In the case of eight total models (second column) we tested four combinations of two strategies, corresponding to partitioning the data across models (MP $=$ 8) and across frames (FP $=$ 4). The three parallel approaches corresponding to 8, 32, and 4 chunks were each more than three times faster than the sequential approach in the upper right of the table, which corresponds to a single chunk. As expected, the division across models was faster (by 17 percent) than the division across pixels, presumably due to the increased overhead in the region case.

The results in Table 2 indicate the need for a dynamic decomposition strategy. When the number of models is small, partitioning frames is the only way to obtain significant parallelism. For larger numbers of models, the optimal decomposition is to partition the models. This decision must be taken on-line as the number of models changes in response to the appearance and disappearance of people in the scene.

While these experiments do not reflect the cost of looking up and implementing an on-line change in the data parallel strategy at each change of state, they do include the overhead incurred by the controller channel at each item.

Table 3
Timing measurements for three decompositions of the target detection task on a two node cluster

	Decomposition	Latency (secs)	Throughput (frames/sec)
A:	Outer – Frames Inner – Regions	2.44	1.005
B:	Outer – Models Inner – Regions	1.44	0.935
C:	Outer – Regions Inner – Regions	1.62	0.862

6.2. *Cluster experiment*

We next examined the performance of data parallel decompositions in a cluster setting. Using a two node cluster we tested the decomposition $\{^2\ [^4\ \mathrm{T4}\ ^4]\ ^d\}$ discussed at the end of Section 5. We implemented three different data parallel strategies with respect to this DP application architecture. In all three cases, the inner (square bracket) partition was over pixels. Four image regions were distributed to four worker threads within each SMP. There were eight color models (targets). The three strategies differed in the partitioning that was done at the outer (curly bracket) level. The number of replicas was two in all cases, since we were using a two node cluster.

Timing results for decompositions *A*, *B*, and *C* are shown in Table 3. Decomposition *A* corresponds to distributing frames at the outer level, *B* to distributing color models, and *C* to distributing regions. For each decomposition, we measured average latency and throughput. Latency was measured as the time (in seconds) between the generation of a frame at the digitizer and the receipt of the computed target positions at the user-interface. Thus all latency numbers include sequential task times before and after T4. To assess throughput, we measured the time between the arrival of successive outputs at the interface. The throughput number reported in the table is the reciprocal of this measured inter-arrival time.

These latency and throughput measurements are steady-state measurements which exclude transients at system start-up. For the latency measurements, we sent single frames through the task graph, using a feedback mechanism to ensure that a new frame enters the system only when the previous frame has been

processed. This is necessary to differentiate the time spent processing an item from the time it spends in channels waiting to be processed. The latter depends on the rate at which frames are produced and the effectiveness of the thread scheduling mechanism, and can be made arbitrarily large. For the throughput measurements, we injected 20 input frames into the system, which filled the channels to capacity, and recorded inter-arrival times after the transient died out.

Decomposition *A* distributes frames at the outer level in order to improve throughput and distributed regions at the inner level. Thus latency remains as it was in the single SMP case since in both cases each frame is processed by a single node. So this case exhibits the highest latencies in the table. The times are similar to the single node implementation of this identical strategy which is reported in Table 2 in the cell (FP $=$ 4, MP $=$ 1). This decomposition has the best throughput.

Decomposition *B* distributes models at the outer level in order to improve the latency of a given frame. There is a small amount of overhead in merging the results so the throughput is slightly reduced. The latency has improved significantly. However, it does not show a speedup of two on two nodes because of the inter-node communication required.

Decomposition *C* distributes regions instead of models at the outer level to improve latency. There is more overhead in dividing regions vs. dividing models. Distributing regions at the outer level leads to less improvement in both latency and throughput than distributing models, when there are enough models to distribute. The difference in performance between these two strategies is even more substantial here than in the SMP case. This further underscores the importance of dynamically choosing the optimal decomposition strategy.

These results demonstrate that dynamic data parallel strategies are required for optimal performance even for a relatively simple vision algorithm. In this example the optimal strategy depends upon the number of targets. Below some threshold we should divide over regions, above it we should divide over models.

The color tracker itself makes up only a small part of the complete kiosk application. As additional vision, speech, and graphics functionalities are implemented, an even more complex set of dynamic choices will result.

7. Conclusions and future work

There is a class of emerging applications that requires the integration of task and data parallelism. The current state of the art in integrating task and data parallelism optimizes the decomposition for static problems. However, our applications are not well suited to these techniques because they exhibit wide variability over time.

The key insight that makes optimization of data parallel decompositions possible in our dynamically varying applications is that the wide variability is over a small number of distinct regimes each of which is amenable to static techniques. For example, in the case of the color tracker, each regime corresponds to a specific number of people in front of the kiosk. We have experimentally demonstrated that the optimal decomposition varies dynamically even for fairly simple algorithms such as the vision-based color tracker.

We have described and implemented a mechanism for integrating task and data parallelism that can effectively exploit dynamically varying decompositions. We have also introduced a notation for specifying DP application architectures in this framework. A preliminary version of this work first appeared in [16].

Currently, the DP application architecture has only been implemented directly at the application level. Our next goal is to automate the generation of these architectures. The input to this process would include the task graph, a data parallelism specification in our data parallel notation, and parameterized splitter, worker, and joiner methods. The abstraction would automatically create the necessary channels and threads by invoking these application-provided methods to generate the structure depicted in Fig. 6. To change the data parallel aspects of the application one would simply change the data parallel specification and regenerate the implementation.

The main source of inefficiency remaining in the current system is thread scheduling. The system currently relies on pthreads to schedule tasks. But the pthreads system does not understand the specific scheduling issues for STM based applications and can easily generate an inefficient schedule. Our approach to scheduling and some experimental results are described in [13].

References

[1] Advanced Visual Systems, Inc. *International AVS Users Conference and Exhibition*, Boston, MA, annual from

1992 to 1995. See the websites www.avs.com (AVS) and www.iavsc.org (AVS user's organization).

[2] R. Bagrodia, M. Chandy and M. Dhagat, UC: A set-based language for data parallel programs, *J. Parallel Distrib. Computing* **28** (Aug. 1995), 186–201.

[3] S. Chakrabarti, J. Demmel and K. Yelick, Models and scheduling algorithms for mixed data and task parallel programs, *J. Parallel Distrib. Computing* **47** (1997), 168–184.

[4] K.M. Chandy, I. Foster, K. Kennedy, C. Koelbel and C.-W. Tseng, Integrated support for task and data parallelism, *Intl. J. Supercomputer Appl.*, 1994.

[5] A.D. Christian and B.L. Avery, Digital smart kiosk project, in: *ACM SIGCHI '98*, Los Angeles, CA, April 18–23, 1998, pp. 155–162.

[6] R. Cipolla and A. Pentland, eds., *Computer Vision for Human-Machine Interaction*, Cambridge University Press, 1998.

[7] T. Darrell, G. Gordon, J. Woodfill and M. Harville, A virtual mirror interface using real-time robust face tracking, in: *Proc of 3rd Intl. Conf. on Automatic Face and Gesture Recognition*, Nara, Japan, April 1998, pp. 616–621.

[8] *Proc. of Third Intl. Conf. on Automatic Face and Gesture Recognition*, Nara, Japan, April 14–16, 1998, IEEE Computer Society.

[9] I. Foster, D. Kohr, R. Krishnaiyer and A. Choudhary, A library-based approach to task parallelism in a data-parallel language, *J. Parallel Distrib. Computing*, 1996.

[10] K. Ghosh and R.M. Fujimoto, Parallel discrete event simulation using space-time memory, in: *20th International Conference on Parallel Processing (ICPP)*, Aug. 1991.

[11] IEEE, Threads standard POSIX 1003.1c-1995 (also ISO/IEC 9945-1:1996), 1996.

[12] D.R. Jefferson, Virtual time, *ACM Trans. Programming Languages and Systems* **7**(3) (July 1985), 404–425.

[13] K. Knobe, J.M. Rehg, A. Chauhan, R.S. Nikhil and U. Ramachandran, Dynamic task and data parallelism using space-time memory, Technical Report 98/10, Compaq Computer Corp. Cambridge Research Lab, November 1998. Abstract in: *Proc. of Eighth Scalable Shared Memory Multiprocessors Workshop*, Atlanta, GA, May 1999.

[14] R.S. Nikhil, U. Ramachandran, J.M. Rehg, R.H. Halstead, Jr., C.F. Joerg and L. Kontothanassis, Stampede: A programming system for emerging scalable interactive multimedia applications, in: *Proc. Eleventh Intl. Wkshp. on Languages and Compilers for Parallel Computing*, Chapel Hill, NC, Aug. 7–9, 1998, pp. 95–109. See also Technical Report 98/1, Cambridge Research Lab., Compaq Computer Corp.

[15] U. Ramachandran, R.S. Nikhil, N. Harel, J.M. Rehg and K. Knobe, Space-time memory: A parallel programming abstraction for interactive multimedia applications, in: *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Atlanta, GA, May 1999, pp. 183–192.

[16] J.M. Rehg, K. Knobe, U. Ramachandran and R.S. Nikhil, Integrated task and data parallel support for dynamic applications, in: *Fourth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, D. O'Hallaron, ed., Pittsburgh, PA, May 28–30, 1998 (Springer), pp. 167–180.

[17] J.M. Rehg, M. Loughlin and K. Waters, Vision for a smart kiosk, in: *Computer Vision and Pattern Recognition*, San Juan, Puerto Rico, 1997, pp. 690–696.

[18] J.M. Rehg, U. Ramachandran , R.H. Halstead, Jr., C. Joerg, L. Kontothanassis and R.S. Nikhil, Space-time memory: A parallel programming abstraction for dynamic vision applications, Technical Report 97/2, Compaq Computer Corp. Cambridge Research Lab, April 1997.

[19] J. Subhlok and G. Vondran, Optimal latency – throughput tradeoffs for data parallel pipelines, in: *Proc. 8th Symposium on Parallel Algorithms and Architecture (SPAA)*, June 1996.

[20] M. Swain and D. Ballard, Color indexing, *Intl J. Computer Vision* **7**(1) (1991), 11–32.

[21] K. Waters and T. Levergood, An automatic lip-synchronization algorithm for synthetic faces, *Multimedia Tools and Applications* **1**(4) (Nov. 1995), 349–366.

[22] K. Waters, J.M. Rehg, M. Loughlin, S.B. Kang and D. Terzopoulos, Visual sensing of humans for active public interfaces, in: *Computer Vision for Human-Machine Interaction*, R. Cipolla and A. Pentland, eds., Cambridge University Press, 1998, pp. 83–96.