

# Stampede: A Cluster Programming Middleware for Interactive Stream-oriented Applications

Umakishore Ramachandran, Rishiyur Nikhil, James Matthew Rehg,  
Yavor Angelov, Arnab Paul, Sameer Adhikari, Kenneth Mackenzie, Nissim Harel,  
Kathleen Knobe

## Abstract

Emerging application domains such as interactive vision, animation, and multimedia collaboration display dynamic scalable parallelism, and high computational requirements, making them good candidates for executing on parallel architectures such as SMPs and clusters of SMPs. Stampede is a programming system that has many of the needed functionalities such as high-level data sharing, dynamic cluster-wide threads and their synchronization, support for task and data parallelism, handling of time-sequenced data items and automatic buffer management.

In this paper, we present an overview of Stampede, the primary data abstractions, the algorithmic basis of garbage collection and the issues in implementing these abstractions on a cluster of SMPs. We also present performance studies including micromasurements and representative multimedia applications implemented on top of Stampede, through which we demonstrate the low overhead of this runtime and that it is suitable for the streaming multimedia applications.

## Index Terms

Middleware, Cluster Computing, Streaming applications, Garbage Collection, Virtual Time

• R. S. Nikhil is presently at Sandburst Corporation, Email: [nikhil@sandburst.com](mailto:nikhil@sandburst.com). This work was done while he was at Compaq CRL

• Kathleen Knobe is with HP CRL, Email: [kath.knobe@hp.com](mailto:kath.knobe@hp.com)

• U. Ramachandran, J. Rehg, Y. Angelov, A. Paul, S. Adhikari, N. Harel and K. Mackenzie are with College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA. Email: {rama, rehg, yavor, arnab, sameera, kenmac}@cc.gatech.edu

The work has been funded in part by an NSF ITR grant CCR-01-21638, NSF grant CCR-99-72216, HP/Compaq Cambridge Research Lab, the Yamacraw project of the State of Georgia, and the Georgia Tech Broadband Institute. The equipment used in the experimental studies is funded in part by an NSF Research Infrastructure award EIA-99-72872, and Intel Corp.

## I. INTRODUCTION

Emerging application domains such as interactive vision, animation, and multimedia collaboration display dynamic scalable parallelism, and high computational requirements, making them good candidates for executing on parallel architectures such as SMPs and clusters of SMPs. There are some aspects of these applications that set them apart from scientific applications that have been the main target of high performance parallel computing in recent years. First, time is an important attribute in such emerging applications due to their interactive nature. In particular, they require the efficient management of temporally evolving data. For example, a stereo module in an interactive vision application may require images with corresponding timestamps from multiple cameras to compute its output, or a gesture recognition module may need to analyze a sliding window over a video stream. Second, both the data structures as well as the producer-consumer relationships in such applications are dynamic and unpredictable at compile time. Existing programming systems for parallel computing do not provide the application programmer with adequate support for such temporal requirements.

To address these problems we have developed an abstraction for parallel programming called **Space-Time memory (STM)** – a dynamic concurrent distributed data structure for holding time-sequenced data. STM addresses the common parallel programming requirements found in most interactive applications, namely, inter-task synchronization, and meeting soft real-time constraints. These facilities are useful for this application class even on an SMP. However, in addition, our system provides the STM abstraction transparently across clusters. Reclamation of STM's time-sequenced data items is an unusual problem quite different from the usual memory address based garbage collection. It is further complicated because of the spread of the computation over a cluster. We present an algorithmic basis for automatic garbage collection across the cluster. We also discuss the issues in implementing these data abstractions on a cluster of SMPs.

STM was first implemented on a cluster of Alpha SMPs (running Digital Unix 4.0) interconnected by Memory Channel. Recently, we have ported Stampede to run on x86-Linux, StrongArm-Linux, and x86-NT platforms as well. We have used STM to implement the following applications so far: (1) a system for the analysis and synthesis of video textures which identifies transition points in a video sequence, and uses them to indefinitely extend the duration of a video clip, (2) a color-based vision tracking component for an interactive multimedia application called the *Smart Kiosk*, which was developed at the Compaq Cambridge Research Laboratory, (3) an image-based rendering application [8] at Compaq CRL, and (4) the distributed data management

in an audio/video meeting application at Georgia Tech.

The key contributions of this paper are :<sup>1</sup>

- the presentation of the STM abstraction for parallel programming,
- a demonstration of ease of use in the context of programming interactive multimedia applications, and
- a performance study using this abstraction on a cluster of SMPs. In particular, we show that STM's significant programming advantage (over, say, direct message-passing) incurs only low performance overheads.

We begin by giving the application context, in Sec. II. In Sec. III, we enumerate the parallel programming requirements engendered by interactive multimedia applications. The Space-Time Memory abstraction, and the unusual garbage collection problem in this class of applications are discussed in Sec. IV. Ease of use of STM is demonstrated via programming examples in Sec. IV-G. We discuss design rationale in Sec. IV-H, and present related work in Sec. IV-I. A brief discussion of the implementation of Stampede is discussed in Sec. V. Performance studies are presented in section VI and concluding remarks are given in Sec. VIII.

## II. APPLICATION CONTEXT

To set the context for the emerging application classes for which our cluster programming system is targeted, we briefly describe a new type of public computer device called the *Smart Kiosk* [25], [4] which has been developed at the Cambridge Research Laboratory of Compaq Computer Corporation. The Smart Kiosk could be located in public spaces such as a store, museum, or airport and is designed to interact with multiple people in a natural, intuitive fashion. For example, we envision Smart Kiosks that entertain passers-by while providing directions and information on local events. The kiosk may initiate contact with customers, greeting them when they approach and acknowledging their departure.

A Smart Kiosk may employ a variety of input and output devices for human-centered interaction: video cameras, microphones, infrared and ultrasonic sensors, loudspeakers, and touch screens. Computer vision techniques are used to track, identify and recognize one or more customers in the scene [19]. A future kiosk will use microphone arrays to acquire speech input from customers, and will recognize customer gestures. Synthetic emotive speaking faces [24]

<sup>1</sup>An overview of the Stampede programming system first appeared in a workshop [16]. The channel abstraction of Stampede and arguments of ease of use were presented in a conference [17]. Details of the garbage collection problem in Stampede were presented in another conference [15]. Discussion of the complete system details of Stampede, and the performance study reported in this paper (Sections VI and VII) have not appeared in any other forum.

and sophisticated graphics, in addition to Web-based information displays, are currently used for the kiosk's responses.

We believe that the Smart Kiosk has features that are typical of many emerging scalable applications, including mobile robots, smart vehicles, intelligent rooms, and interactive animation. These applications all have advanced input/output modes (such as computer vision), very computationally demanding components with dynamic structure, and real-time constraints because they interact with the real world.

### III. APPLICATION PROGRAMMING REQUIREMENTS

The parallel structure of the Smart Kiosk is highly dynamic. The environment in front of the kiosk (number of customers, and their relative position) and the state of its conversation with the customers affect which threads are running, their relative computational demands, and their relative priorities (*e.g.*, threads that are currently part of a conversation with a customer are more important than threads searching the background for more customers). There are a number of other applications (such as interactive animation, and distributed audio/video meetings) that have similar characteristics to the Smart Kiosk.

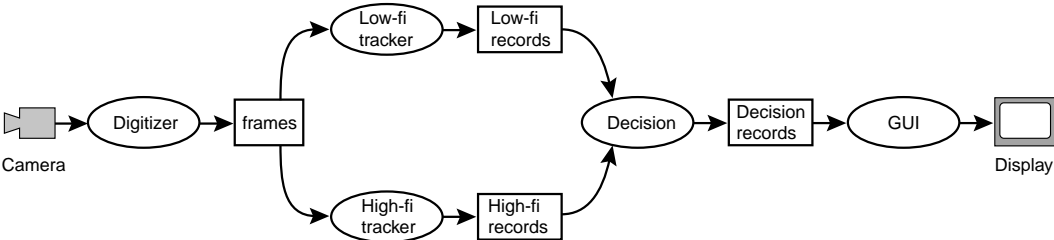


Fig. 1. A simple vision pipeline.

A major problem in implementing these kinds of application is “buffer management”. This is illustrated in the simple vision pipeline shown in Fig. 1. The *digitizer* produces digitized images every 30th of a second. The *Low-fi tracker* and the *Hi-fi tracker* analyze the frames produced by the digitizer for objects of interest and produce their respective tracking records. The *decision module* combines the analysis of such lower level processing to produce a decision output which drives the *GUI* that converses with the user. From this example, it should be evident that even though the lowest levels of the analysis hierarchy produce regular streams of data items, four things contribute to complexity in buffer management as we move up to higher levels:

- Threads may not access their input datasets in a strict stream-like manner. In order to conduct a convincing real-time conversation with a human a thread (*e.g.*, the Hi-fi tracker)

may prefer to receive the “latest” input item available, skipping over earlier items. The conversation may even result in canceling activities initiated earlier, so that they no longer need their input data items. Consequently, producer-consumer relationships are hints and not absolute, complicating efficient data sharing especially in a cluster setting.

- Datasets from different sources need to be combined, correlating them temporally. For example, stereo vision combines data from two or more cameras, and stereo audio combines data from two or more microphones. Other analyzers may work multi-modally, *e.g.*, by combining vision, audio, gestures and touch-screen inputs.
- Newly created threads may have to re-analyze earlier data. For example, when a thread (*e.g.*, a Low-fi tracker) hypothesizes human presence, this may create a new thread (*e.g.*, a Hi-fi tracker) that runs a more sophisticated articulated-body or face-recognition algorithm on the region of interest, beginning again with the original camera images that led to this hypothesis. This dynamism complicates the recycling of data buffers.
- Since computations performed on the data increase in sophistication as we move through the pipeline they also take more time to be performed. Consequently, not all the data that is produced at lower levels of the processing will necessarily be used at the higher levels. As a result, the datasets become temporally sparser and sparser at higher levels of processing because they correspond to higher- and higher-level hypotheses of interesting events. For example, the lowest-level event may be: “a new camera frame has been captured”, whereas a higher-level event may be: “John has just pointed at the bottom-left of the screen”. Nevertheless, we need to keep track of the “time of the hypothesis” because of the interactive nature of the application.

These algorithmic features bring up two requirements. First, data items must be meaningfully associated with time and, second, there must be a discipline of time that allows systematic reclamation of storage for data items (garbage collection).

In addition to the buffer management issue, specific tasks within these applications lend themselves very nicely to data parallelism. Consider, for example the High-fi tracker in the vision pipeline shown in Fig. 1. The latency for processing a frame by this tracker could well exceed the rate at which the Digitizer (upstream) may produce frames for analysis. In such situations, an obvious approach would be to apply multiple processors to the tracking task operating in data parallel mode on distinct image frames or on parts of the same frame.

#### IV. SPACE-TIME MEMORY

The *Stampede* project addresses the parallel programming requirements posed by interactive multimedia applications such as those discussed in Sec. II. Stampede allows the creation of multiple address spaces in the cluster and an unbounded number of dynamically created application threads within each address space. The threading model within an address space is basically standard OS threads such as *pthreads* (POSIX threads) on Tru64 Unix and Linux [6], and Win32 threads on Windows NT. Stampede provides high-level data sharing abstractions that allow threads to interact with one another without regard to their physical locations in the cluster, or the specific address spaces in which they execute.

A novel component of Stampede is Space-Time Memory (STM), a distributed data structure that addresses the complex “buffer management” problem that arises in managing temporally indexed data items as in the Smart Kiosk application. Traditional data structures such as streams and lists are not sufficiently expressive to handle the requirements enumerated in the previous section.

STM *channels* provide random access to a collection of time-indexed data items, while STM *queues* give a FIFO access to a similar collection. We will first describe the channel and then remark on the similarities and differences between channels and queues. STM channels can be envisioned as a two-dimensional table. Each row, called a *channel*, has a system-wide unique id. A particular channel may be used as the storage area for an activity (e.g. a digitizer) to place the time-sequenced data records that it produces. Every column in the table represents the temporally correlated output records of activities that comprise the computation. For example, in the vision pipeline in Fig. 1, the digitizer produces a frame  $F_t$  with a timestamp  $t$ . The Low-fi tracker produces a tracking record  $LF_t$  analyzing this video frame. The decision module produces its output  $D_t$  based on  $LF_t$ . These three items are on different channels and may be produced at different real times, but they are all temporally correlated and occupy the same column  $t$  in the STM. Similarly, all the items in the next column of the STM channel table have the timestamp  $t+1$ . Fig. 2 shows an example of how the STM channels may be used to orchestrate the activities of the vision processing pipeline introduced in Fig. 1. The rectangular box at the output of each activity in Fig. 1 is an STM channel. The items with timestamp 1 ( $F_1$ ,  $LF_1$ ,  $HF_1$ , and  $D_1$ ) in each of the four boxes in Fig. 1 is a column in the STM.

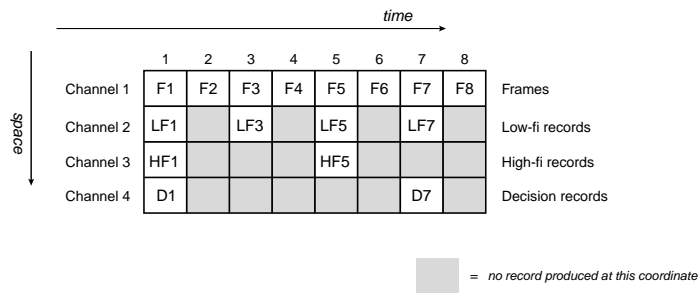


Fig. 2. Mapping the vision pipeline to STM channels.

### A. API to the STM Channel Abstraction

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 multiple connections to the same channel. Fig. 3 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. A channel cannot have more than one item with the same timestamp, but there is no constraint 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 either to block or to return immediately with an error code if a bounded output channel is full.

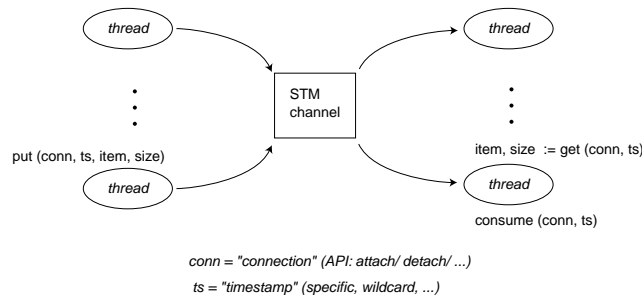


Fig. 3. Overview of Stampede channel usage (relationship of a channel to threads)

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, for example, the newest/oldest value currently in the channel, or the newest value not previously gotten over any connection. 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 on multiple address spaces may simultaneously be performing operations on a given 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.

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 putting items into and getting items from a temporally ordered collection, concurrently, not of communicating between processes.

## B. STM Queues

The primary reason for providing the STM *queue* abstraction is to support data parallelism in a cluster. As we mentioned earlier, the targeted application classes provide plenty of opportunities for exploiting data parallelism. For example, in the vision pipeline (see Fig. 1), data parallel instances of the tracker could operate in parallel on distinct image frames or on parts of the same frame. STM queues are provided for this purpose. Similar to the channel, a queue has a system-wide unique id. The queue abstraction supports the same set of calls as a channel: `get`, `put`, and `attach`. The runtime system allows a “timestamp” attribute to be associated with an item in a queue just as in the case of a channel. Aside from the “timestamp” attribute associated with a queue item the `get/put` operations on a queue are semantically the same as the `enqueue/dequeue`

operations on a traditional queue data structure. The nature of a traditional queue data structure coupled with the fact that the queue items have a timestamp attribute leads to the following differences between an STM queue and an STM channel:

- a *get* on a queue gives an item in strictly *FIFO* order (*i.e.*, irrespective of the timestamp order of the queue items); the runtime provides the timestamp and ticket associated with this item to the getting thread,
- a queue item *has* to be gotten exactly once (otherwise it will never go away as we will see in the next subsection), and cannot be gotten more than once; a channel item *may* be gotten 0 times or as many times as the number of connections to that channel (modulo any reference count specification for that item, see Section IV-H),
- multiple items with the *same* timestamp can be *put* into the queue; this may be necessary in a vision pipeline for instance, if the strategy for data parallelism is to carve out a given image frame into smaller segments; the runtime system associates a tag (called a *ticket*) with each fragment that uniquely identifies a particular item in a queue.

### C. STM Registers

In addition to channels and queues, Stampede also provides cluster-wide abstraction called *registers*. A thread can attach and detach to a register just like channels or queues. A register can be used like a cluster-wide shared variable. Writing to a register over-writes its previous contents. A register read operation returns successfully if a new value is written onto it. A thread can block on such a read until a new write happens. The *full/empty* synchronization semantics provides a mechanism to implement inter-thread signaling and event notification.

### D. Garbage Collection

In dealing with timestamped data in this application domain we encounter an unusual notion of garbage collection, where “reachability” concerns timestamps and not memory addresses. If physical memory were infinite, STM’s *put* and *get* primitives would be adequate to orchestrate the production and access to time-sequenced data in any application. However, in practice it is necessary to garbage collect data items that will no longer be accessed by any thread. When can we reclaim an item from a timestamp-indexed collection? The problem is analogous to the classical “space leak” situation where, whenever a table is reachable from the computation, no item in that table can be garbage collected on the basis of reachability alone, even if there are items that will never be accessed subsequently in the computation. A complication is the fact that

application code can do arithmetic on timestamps. Timestamp-based GC is orthogonal to any classical address-based GC of the STM's host language. This section discusses the guarantees provided by the STM for producing and accessing time-sequenced data, and the guarantees that the application must provide to enable garbage collection.

To enable garbage collection of an STM item, the API provides a `consume` operation by which the application declares to STM that a specific STM item<sup>2</sup> is garbage from the perspective of a particular connection. A queue item has an implicit reference count of *one*. So as soon as the thread that got that item calls `consume` on that item, STM can safely garbage collect it. Although `get` semantics is copy-out, as we shall see in section IV-E, because of items that may involve embedded pointers, it is mandatory that `consume` be explicitly called. Garbage collection is a little more involved in the case of a channel. STM can safely garbage collect an item once it has determined that the item can no longer be accessed through any existing connection *or any future connection* to this channel. So the discipline imposed by STM on the application programmer is to get an item from a channel, use it, and mark it as consumed. An object  $X$  in a channel is in one of three states with respect to each input connection  $ic$  attaching that channel to some thread. Initially,  $X$  is “unseen”. When a `get` operation is performed on  $X$  over connection  $ic$ , then  $X$  is in the “open” state with respect to  $ic$ . Finally, when a `consume` operation is performed on the object, it transitions to the “consumed” state. We also say that an item is “unconsumed” if it is unseen or open. The contract between the runtime system and the application is as follows: The runtime system guarantees that an item will not be garbage collected at least until it has been marked consumed on all the connections that have access to it. An application thread has to guarantee to mark each item on its input connections as consumed. The `consume` operation can specify a particular object (*i.e.*, with a particular timestamp), or it can specify all objects up to and including a particular timestamp. In the latter case, some objects will move directly into the consumed state, even though the thread never performed a `get` operation on them.

Similarly, there are rules that govern the timestamp values that can be associated with items produced by a thread on a connection (be it a channel or a queue). A thread can associate a timestamp  $t$  with an item it produces so long as this thread has an item  $X$  with timestamp  $t$  currently in the open state on one of its input connections. This addresses the common case (*e.g.*, the Low-fi tracker thread in Fig. 1) where a thread gets an item from its input connection, processes it, produces a timestamped output (correlated to the timestamp of the item

<sup>2</sup>In the case of a channel, the  $(connection, timestamp)$  pair uniquely identifies a particular item. In the case of a queue, the  $(connection, ticket)$  pair uniquely identifies a particular item.

it is processing, possibly even the same timestamp) as a result of the processing, and marks the item consumed. We say that the output item *inherits* the timestamp of the input item.

However, there are situations where timestamped output may have to be generated without getting an item from the STM channel. This is in general true for application “source” threads that have no input connections (*e.g.*, the digitizer thread in Fig. 1, with the corresponding code fragment shown in Fig. 6), or a root thread in a task connectivity graph that drives the whole computation. For this purpose, the STM maintains a state variable for each thread called *virtual time*. An application may choose any application-specific entity as the virtual time. For example, in the vision pipeline (Fig. 1), the frame number associated with each camera image may be chosen as the virtual time. The current *visibility* of a thread is the minimum of its virtual time and the timestamps of any items that it currently has open on any of its input connections. When a thread puts an item, it can give it any timestamp  $\geq$  its current visibility. When a thread creates a new thread, it can initialize the child thread’s initial virtual time to any value  $\geq$  its own current visibility. When a thread creates a new input connection to a channel, it implicitly marks as consumed on that connection all items  $<$  its current visibility. A thread can explicitly change its own virtual time to any value  $\geq$  its current visibility. In most cases, a thread can set its own virtual time to the special value INFINITY because the timestamps of items it puts are simply inherited from those that it gets.

These rules enable the runtime system to transitively compute a global minimum  $ts_{min}$ , which is the minimum of:

- virtual times of all the threads,
- timestamps of all items on all queues, and
- timestamps of all unconsumed items on all input connections of all channels.

This is the smallest timestamp value that can possibly be associated with an item produced by any thread in the system. It is impossible for any current thread, or any subsequently created thread, ever to refer to an object with timestamp less than the global minimum. Thus, all objects in all channels with lower timestamps can safely be garbage collected. Stampede’s runtime system has a distributed algorithm that periodically recomputes this value and garbage collects dead items. To ensure that this global minimum advances and thus garbage collection is not stymied a thread must guarantee that it will advance its virtual time, for which STM provides an API call:

```
spd_set_virtual_time (new_virtual_time).
```

The `consume` call is reminiscent of reference counting. However, this gets complicated in the case of an STM channel because the number of consumers of an item is unknown— a thread may

skip over items on its input connections, and new connections can be created dynamically. These interesting and subtle issues, as well as our distributed, concurrent garbage collection algorithm are described in greater detail in a separate paper [15].

### *E. Communicating Complex Data Structures through STM*

The `put` and `get` mechanisms described above are adequate for communicating contiguously allocated objects through channels and queues, but what about linked data structures? In the Smart Kiosk vision code, for example, a “color model” data structure is actually a complex of four or more separately allocated components linked with C pointers. We wish to treat them as a single unit that can be communicated through STM. The C pointers are of course meaningless in a different address space.

To solve this problem, Stampede extends the basic STM system with a notion of “object types”. The following call:

```
spd_dcl_type (type, marshall_hook, unmarshall_hook, ...)
```

declares a new object type (represented by an integer), with an associated set of methods, or procedures. Two of these are hooks that assist in marshalling and unmarshalling objects for transmission between address spaces.

A variant of the channel/queue `put` procedure is based on types instead of object sizes. Its parameters include a pointer to the data structure and its type instead of its size (which is not particularly meaningful for a linked data structure). Similarly, a variant of the channel/queue `get` call returns a pointer to the linked data structure, and its type instead of size. Figure 4 shows an overview of how these facilities are used. Stampede takes care of the marshalling, communication and unmarshalling of the data structure, using the supplied hooks to decompose and reconstitute the “object”. These actions are done lazily, *i.e.*, only when a consumer actually attempts to `get` an item, and intermediate results are cached (at the producer and the consumers) to avoid repeating this work in the presence of multiple `get`'s. The normal garbage collection process, described in the previous subsection, is extended to reclaim any such cached intermediate results.

If we implement Stampede in a language with a richer type system, the application programmer could perhaps be relieved of the burden of specifying these hooks (*cf.* “serializer” mechanisms in Java). However, even in this case, it would be useful to have the ability to override these default methods. For example, image data structures in the Smart Kiosk vision code include a linked list of attributes which can, in fact, be recomputed from the object during unmarshalling, and therefore do not need to be transmitted at all. Further, the image data itself can be compressed during marshalling and decompressed during unmarshalling. Such application- and type-specific

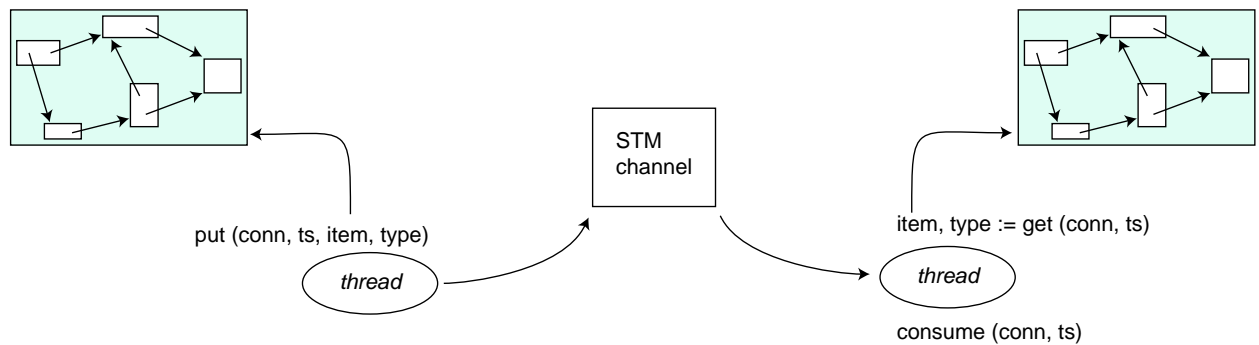


Fig. 4. Communicating complex objects through channels, based on ‘types’

generalizations of “marshalling” and “unmarshalling” cannot be provided automatically in the default methods.

In addition to serialization routines, the application can install specific garbage-handler routines to clean up such complex items. Although, `get` has a copy-out semantics, it may copy out just a pointer to such an item. Therefore, typically the runtime has no idea of when the item can be garbage collected. Only after an explicit `consume` call is made, the runtime can run the installed routine to reclaim the memory.

#### F. Synchronization with Real-time

The virtual time and timestamps described above with respect to STM are merely an indexing system for data items, and do not in of themselves have any direct connection with real time. For pacing a thread relative to real time, Stampede provides an API for loose temporal synchrony that is borrowed from the Beehive system [22]. Essentially, a thread can declare real time “ticks” at which it will re-synchronize with real time, along with a tolerance and an exception handler. As the thread executes, after each “tick”, it performs a Stampede call attempting to synchronize with real time. If it is early, the thread waits until that synchrony is achieved. If it is late by more than the specified tolerance, Stampede calls the thread’s registered exception handler which can attempt to recover from this slippage. Using these mechanisms, for example, the digitizer in the vision pipeline can pace itself to grab images from a camera and put them into its output channel at 30 frames per second, using absolute frame numbers as timestamps.

#### G. Programming Examples

In this section, we show some STM programming examples. Fig. 5 shows the relationship of an application thread to the STM abstraction. The only interaction it has with the other threads

in the application is via the Stampede data abstractions it is connected to on the input and output sides. Other than the specific calls to the STM to get, put, or consume an item, the thread executes its sequential algorithm.

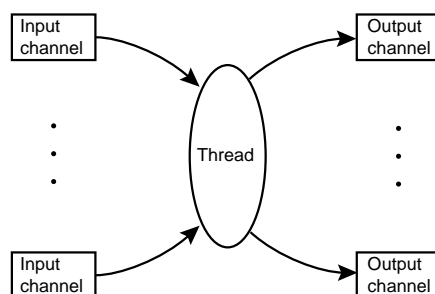


Fig. 5. Relationship of an application thread to STM.

For the vision pipeline in Fig. 1, we present code fragments for the digitizer thread and a tracker thread in Figs. 6 and 7, respectively.

It can be seen from the code fragments that the extent of application modification required to use the STM is small and localized to the specific regions where a thread would need to communicate with its peers under any parallel programming regime. More importantly, all such communication and synchronization are encapsulated in the get, put, and consume calls. The threads never have to synchronize explicitly with other threads, nor do they have to know the existence of other threads in the applications. All that a particular thread needs to know is the names of the channels it should expect inputs from and the channels to which it should send its outputs (see Fig. 5). Thus STM relieves the application programmer from low level synchronization and buffer management. Moreover, the virtual time and timestamp mechanisms of the STM provide a powerful facility for the application programmer to temporally correlate disparate data items produced at different real times by different threads in a complex application.

Space limitations prevent us from presenting more elaborate programming examples here. The program represented by the code fragments in Figs. 6 and 7 could perhaps have been written using straight message-passing, except that the STM code is still simpler because of its location-independence (producer and consumer need not be aware of each other), and because the consumer has the capability of transparently skipping inputs (using the `STM_LATEST_UNSEEN` flag in its get call). A more elaborate example would involve dynamic thread and channel creation, dynamic attachments to channels, multiple producers and consumers for a channel with complex production and consumption patterns *etc.*. These features, along with STM's automatic garbage collection, would be difficult to reproduce with message-passing code.

In addition to the Smart Kiosk system we have used throughout as a motivating example,

### Digitizer thread

```
...  
  
/* create an output connection to an STM channel */  
oconn = spd_attach_output_channel(video_frame_chan)  
  
/* specify mapping between vt tick and elapsed real-time */  
spd_tg_init(TG_DIGITIZE, 33)  
  
/* frame_count will be used as the virtual time marker  
   for the digitizer */  
frame_count = 0  
while (True) {  
    frame_buf = allocate_frame_buffer()  
    frame_buf ← digitize_frame()  
  
    /* put a timestamped output record of the frame */  
    spd_channel_put_item(oconn, frame_count, frame_buf)  
  
    /* advance digitizer's virtual time */  
    frame_count++  
  
    /* announce digitizer's new virtual time to STM */  
    spd_set_virtual_time(frame_count)  
  
    /* synchronize digitizer's virtual time with real-time */  
    spd_tg_sync_vt_with_rt(TG_DIGITIZE)  
}
```

Fig. 6. Digitizer code using the STM calls.

### Tracker thread

```
...
/* announce to STM that the thread's virtual time is
   +infinity for the purposes of garbage collection */
spd_set_virtual_time(+infinity)

/* create an input connection to the STM channel for
   getting video frames from the digitizer */
iconn_frame = spd_attach_input_channel(video_frame_chan)

/* create an output connection to an STM channel for
   placing tracker output records */
oconn = spd_attach_output_channel(model_location_chan)

while (True) {
    location_buf = allocate_location_buffer()

    /* get the most recent frame produced by the digitizer,
       and record its timestamp in  $T_k$  */
    (frame_buf,  $T_k$ ) = spd_channel_get_item(iconn_frame,
                                             STM_LATEST_UNSEEN)

    /* tracker algorithm for detecting target model
       in video frame */
    location_buf ← detect_target(frame_buf)

    /* put the location of the detected target in STM channel
       corresponding to tracker's output records */
    spd_channel_put_item(oconn,  $T_k$ , location_buf)

    /* mark the video frame consumed */
    spd_channel_consume_items_until(iconn_frame,  $T_k$ )
}
```

Fig. 7. Tracker code using the STM calls.

Stampede is also being used in another application called image-based rendering [8], [20] at CRL. At Georgia Tech, it has been used to implement an audio/video meeting application, and a video texture generation application. A variant of the STM model has been investigated at Rice University for irregular scientific computations [3].

#### H. Design Rationale

In designing the STM abstraction, we have attempted to keep the interface simple and intuitive. We provide the reasoning behind some of the design choices we made along the way:

- **Virtual versus Real timestamps:** Despite the fact that the primary intent of this abstraction is to support interactive applications, we chose an application-derived quantity to be used as timestamps. While some applications may benefit by using real-time for temporal correlation, it was not clear that in general the runtime could make correlations (using real-time) between independent streams that may use different sampling rates on input data (*e.g.*, voice versus video). Further, data may be captured at some real time but processed at a much later real time. By virtualizing time, the same timestamp index can be associated with both the raw and processed data thus empowering applications to propagate temporal causality. We chose to allow the application to specify the mapping of the virtual time ticks to real time, and use that relationship purely for scheduling the threads (*i.e.*, pacing an individual thread’s activity) and not for temporal correlation.

There could be application scenarios in which there are streams (possibly with different time bases) that are semantically independent of one another from the application perspective. Clearly, the garbage collection of such streams should be independent of one another. However, garbage collection in Stampede relies on a single value, namely, the *global virtual time*. Thus from the point of view of garbage collection, data items that are in such independent streams will be deemed by the runtime system as temporally related to one another if they happen to have the same timestamp. This is clearly a limitation of the current system. One possibility for circumventing this limitation is to define independent virtual time zones, and we will explore this possibility in future evolutions of the system.

- **Virtual Time Management:** As mentioned in Sec. IV-D a “source” thread (with no input connections) must manage its virtual time explicitly, purely for the purpose of garbage collection, whereas most other threads implicitly inherit time based on what is available on their input connections. A more complex and contrived alternative would have been to let source threads make input connections to a “dummy” channel whose items can be regarded

as “time ticks”.

- **Connections to Channels and Queues:** A design choice is to allow operations directly on channels and queues instead of via explicit connections, thus simplifying the API. However, there are two reasons why we chose a connection-based access to channels and queues.
  - The first reason has to do with flexibility. Our approach allows a thread to have multiple connections to the same channel. Such a flexibility would be valuable for instance if a thread wants to create a debugging or a monitoring connection to the same channel in addition to the one that it may need for data communication. While the same functionality could be achieved by creating a monitoring thread, we think that connections are a more intuitive and efficient way to achieve this functionality.
  - The second reason has to do with performance. Connections can play a crucial role in optimizing communication especially in a cluster setting by providing a hint to the runtime system as to who may be potential consumers for a data item produced on a channel (so that data can be communicated early).
- **Garbage Collection:** STM provides transparent garbage collection by performing reachability analysis on timestamps. In a cluster, this could be quite expensive since the put and get operations on a channel or a queue are location transparent, and can be performed by threads anywhere in the cluster that have connections to that channel or queue. The alternative would have been to associate a reference count and garbage collect a channel item as soon as its reference count goes to zero. There are two issues with this alternative:
  - As we discussed earlier, not all produced items may necessarily be used in a dynamic application such as interactive vision. Thus an item that was skipped over by all potential consumer threads will never be garbage collected since its reference count will never go to zero.
  - Further, in many such dynamic applications a producer may not know how many consumers there may be for an item it produces.

We do, however, allow a channel put operation to specify an optional reference count (a special value indicates that the consumer count is unknown to the producer). The runtime employs two different algorithms. The first algorithm uses reference counts<sup>3</sup>. A second algorithm based on reachability analysis to garbage collect channel items with unknown reference counts is run less frequently.

<sup>3</sup>Recall that a queue item has an implicit reference count of one.

## I. Related Work

The STM abstraction may be viewed as a form of structured shared memory. In this sense it is related to recent distributed shared memory systems (such as Cashmere [10], Shasta [21], and Treadmarks [9]). DSM systems typically offer the same API as any hardware SMP system and therefore are too low level to simplify programming of the complex synchronization and communication requirements found in interactive multimedia applications (as mentioned earlier, STM is useful even on an SMP). Further, from a performance perspective DSM systems are not particularly well-suited for supporting applications with highly dynamic sharing characteristics.

There have been several language designs for parallel computing such as Linda [1] (and its more recent derivatives such as JavaSpaces [12] and T-Spaces [26]), Orca [2], and Cid [13]. The data sharing abstractions in these languages are at a lower level than STM; of course, they could be used to implement STM.

Temporal correlation of independent data streams is a key distinguishing feature of our work from all prior work. The work most closely related to ours is the Beehive [22] software DSM system developed by one of the authors and his colleagues at the Georgia Institute of Technology. The delta consistency memory model of Beehive is well-suited for applications that have the ability to tolerate a certain amount of staleness in the global state information. Beehive has been used for real-time computation of computer graphical simulations of animated figures. STM is a higher level structured shared memory that can use the lower-level temporal synchronization and consistency guarantees of Beehive.

The idea of Space-Time memory has also been used in optimistic distributed discrete-event simulation [7], [5]. The purpose and hence the design of Space-Time memory in those systems is very different from ours. In those 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 this paper, we have proposed Space-Time Memory as the fundamental building block around which the entire application is constructed.

## V. IMPLEMENTATION

Stampede was originally implemented (at Compaq CRL) on a cluster of 4-way Alpha SMPs interconnected by Memory Channel and running Tru64 Unix. Since then it has been ported to clusters of x86-Linux, x86-Solaris, StrongArm-Linux, and Windows nodes. The Stampede runtime systems assumes a reliable messaging layer underneath. We have built Stampede on top of two such messaging layers: *MPI* [11], and *CLF* [14]. Both MPI and CLF provide basic

message transport. While MPI uses TCP/IP for reliability, CLF implements its own packet-based reliability layer on top of UDP. Stampede facilitates the creation of any number of address spaces in each node of the cluster, and threads within each address space. The channels/queues can be created in any of the address spaces and have system-wide unique ids allowing transparent access to them by a thread running anywhere in the cluster. The runtime implements caching of items fetched from remote channels and queues for transparent sharing by Stampede threads collocated on the same node of the cluster. Detailed discussion of the implementation details of the Stampede runtime library is beyond the scope of this paper.

## VI. BASELINE PERFORMANCE OF STAMPEDE

In addition to simplifying programming, STM has the potential to provide good performance on clusters, for several reasons. First, synchronization and data transfer are combined in STM, permitting fewer communications. Second, connections provide useful hints for optimizing data communication across clusters. Third, sharing across address spaces is orchestrated via the STM abstraction which can therefore optimize it in a more targeted manner than the indiscriminate sharing that can occur in a DSM system for dynamic applications.

We have conducted two sets of experiments to evaluate the performance of Stampede: first a set of micro-benchmarks and next a set of applications. In the microbenchmarking experiments we measured the latency and bandwidth of `put/get` calls in Stampede. Comparison of the latencies taken by the Stampede calls against the latencies taken by the messaging layer underneath reveals that STM incurs only nominal marginal processing cost. Similarly, recording the sustained bandwidth at the STM runtime level reveals that STM offers bandwidth comfortably above what is required for continuous display of moderate resolution camera images at 30 frames/second. Due to space restrictions we provide detailed description of the experiments and the results in Appendix-A.

In Section VII, we present performance studies for representative multimedia applications implemented using Stampede.

The computing platform for all the experiments is a cluster of SMP nodes running Linux. The hardware consists of 17 Dell 8450 servers each with eight 550MHz Pentium III Xeon CPUs, 2MB of L2 cache per CPU and 4GB of memory per node. The 8450 uses the Intel ProFusion chipset which provides two 64-bit/100MHz system (front-side) busses, one for each bank of four CPUs. The nodes are interconnected with doubled Gigabit Ethernet through a dedicated switch.

## VII. Application Level Performance

In this section we describe a set of experiments for evaluating the performance of two interactive multimedia applications on the Stampede cluster. The first is a *video textures* application. This application takes an input video sequence and calculates a set of cut points which can be used to “loop” the video sequence indefinitely, by randomly transitioning between the computed cuts. This requires calculating the best matches across all pairs of frames in order to identify the best places to cut. This is a computationally-demanding batch video processing task.

The second application is a *color tracker* that operates on a live video stream and tracks the locations of multiple moving targets on the basis of their color signatures. This application requires real-time color analysis of the video frames and comparison against multiple color models. These two applications are typical of the kinds of computations that new multimedia systems will require. They exhibit an interesting range of computational properties, which are illustrated in the experiments that follow. Due to space restriction, we present the second application and its performance in Appendix-B.

The software is Linux-based with Intel-provided libraries. The operating system is Linux with the 2.4.9 kernel. The system scheduler in this kernel is oblivious to the 8450’s split system bus. The compiler is GCC version 2.96 with optimization set to -O2. The video textures application uses the `ippiNormDiff_L2_8u_C1R()` procedure in Intel’s Integrated Performance Primitives (IPP) library, version 1.1 at the core of its processing. For these application level studies we use Stampede on top CLF.

### A. Experiments with Video Textures

The overall application pipeline is shown in Figure 8. The computationally intensive part of the application is the box labeled *Analysis*. This represents the kernel of the application that we parallelize on the cluster. The distinguishing characteristic of this application kernel is that the input dataset of  $N$  images are all produced by a *digitizer thread* that is on one node of the cluster (the images come from one camera source which can be thought of as attached to that cluster node). The core computation in the kernel is a comparison of every image with every other image in the sequence. Thus the total work to be done in the kernel is the total number of image comparisons:  $W = N(N - 1)/2$ . The actual computation is an L2 differencing<sup>4</sup> between every two images. Note that every pair of image comparison is independent of other comparisons.

<sup>4</sup>The L2 norm of the difference of two vectors  $u$  and  $v$  is:  $\sqrt{\text{sum}((u - v)^2)}$ . Each image is “unwrapped” in raster scan order to form a vector.

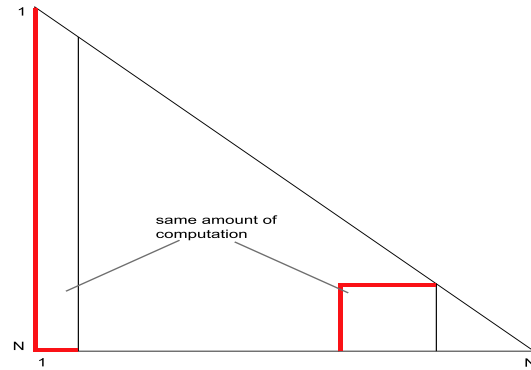
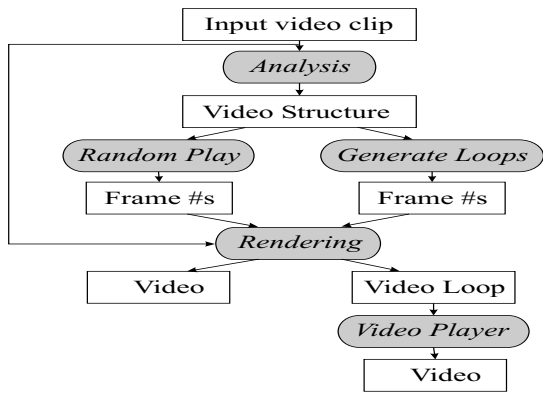


Fig. 8. *Video Textures System Overview.* An input video clip is fed into the *Analysis* component, which finds good transition points where the video can be looped back on itself. These transitions (the *Video Structure*) are fed to one of two *Synthesis* components: either *Random Play*, which sequences the transitions stochastically; or *Generate Loops*, which finds a set of transitions that together equal number of image comparisons. However, the column create a single overall video loop of a given length. The *Rendering* component takes the generated sequence of frames, together with the original video clip, for comparison against one image while the square tile and produces either an infinite video texture sequence, or a video loop that can be played indefinitely by a standard *Video Player* in “loop” mode.

Fig. 9. *Video Textures Computation Space.* The lower triangular nature of the computation space has a direct impact on choice of work distribution. The column on the left and the square tile on the right perform an equal number of image comparisons. However, the column would require fetching all  $N$  images from the digitizer and requires only a subset of images and reuses them.

In this sense the kernel is embarrassingly parallel. The tricky part of parallelizing this kernel is determining an efficient partitioning scheme to minimize the data distribution costs, improve cache performance and thus optimize the overall execution time on the cluster.

Figure 9 shows the input dataset and the computation space of the kernel. In the experimental study we use a total of 316 images amounting a total of 49770 image comparisons. Each image is of size  $640 \times 480$  color pixels, approximately 900KB.

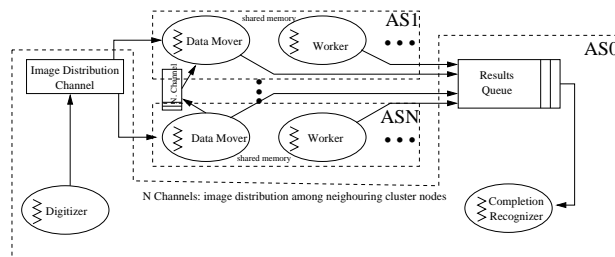


Fig. 10. *Video Texture Implementation*

1) *Mapping the Video Texture Analysis onto Stampede:* We use the Stampede programming library to implement the video texture analysis kernel. Figure 10 shows the implementation using Stampede threads, channels, and queues. AS, AS1, ..., ASN denote the cluster nodes. Stampede threads within the same node share memory. The digitizer thread, a *channel* for image distribution, a *queue* for collecting the correlation results, and a *completion recognizer* thread that is notified when the analysis is complete are all located on AS0. Each of the other cluster nodes participating in the computation has one *data mover* thread and some number of *worker* threads.

The data mover thread prefetches images from the image distribution channel, and depending on the data distribution scheme (to be discussed shortly) may also pass them around to other nodes of the cluster via Stampede channels. The worker threads carry out the image comparisons. The digitizer gives sequential numbers as the “timestamp” for an image that is *put* on a channel, and the data mover and worker threads use this timestamp to *get* a particular image from the channel. The timestamp essentially serves as an index into the image array that is contained in the image distribution channel. This is an interesting and unintended use of the Stampede timestamp mechanism, and has two consequences. The first is a programming convenience in that the channel appears logically like shared memory across the cluster nodes. The second is a performance consequence in that an image that is prefetched into a cluster node by the data mover thread is shared by all the worker threads that are on this node via the Stampede *get* operation, since Stampede runtime caches items that are fetched from remote nodes. Stampede “registers” are used for event notification purposes (not shown in the figure) and to exchange global information.

2) *Performance Concerns*: While it is straightforward to implement the video texture kernel using Stampede, it is non-trivial to optimize the kernel for performance. To address the performance concerns related to inter-node distribution of data we introduce a work distribution strategy similar to multicast on top of unicast (Figure 11). To address memory hierarchy performance concerns (particularly cache hit ratio), we compare images in stripes of pixels, rather than as a whole, and apply heuristics to order the memory requests.

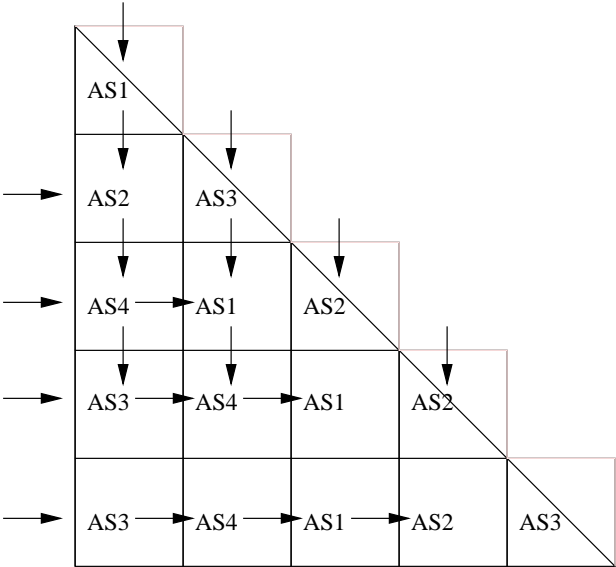


Fig. 11. Conceptual diagram of the *tiling-with-chaining* work distribution strategy.

3) *Experimental Setup*: The input dataset is 316 images of 900KB each. We conducted the following experiments:

- *Stripe size*. The intent is to determine the speedup on a single node with varying stripe size and number of worker threads.

Even with a single worker thread, we would expect performance improvement as the stripes become smaller, since the memory bus may not be able to serve even a single CPU doing full-size image comparison. The experiments included varying the number of worker threads from 1 to 8, and the size of stripes from 2 to 480 lines of pixels, where each line is 1920 bytes. The optimum stripe size is non-obvious since the intermediate results have to be cached and later combined.

- *Efficiency of inter-node distribution strategy*. This experiment is intended to test the scalability of the inter-node distribution strategy. Each node employs a single worker thread. The number of nodes is increased from 2 to 14, with node AS0 serving as the source node for the images.
- *Overall speedup*. This experiment is intended to assess the overall speedup as the number of nodes is increased from 2 to 14, and the number of worker threads is increased from 1 to 8.

4) *Results and discussion*: Figure 12 shows the results of the stripe-size experiment. The curve for a stripe size of 480 lines represents fetching entire images; the execution time flattens out at about 27 seconds (a speedup of about 3.5) corresponding closely to the memory bandwidth limit of the 8450 system.

The other three curves represent different choices in stripe sizes intended to fit in the cache and the results reveal an optimum stripe size of two. The results for stripe size of four comes close, which at first glance is surprising since the worst case cache capacity needed for this stripe size is 2370 KB ( $4 \times 1920 \times 316$ ), which is more than the available cache size of 2MB. However, due to the incomplete tiles along the edges of the triangle the average cache size needed is much less than the worst case leading to this result. The best speedup achieved is 5.75 for eight workers, which is pretty good considering that there is a data mover thread in addition to the workers taking up some cycles on the node.

Clearly the optimum stripe size is a function of the problem size (i.e. the number of images being compared), and the available cache on the processor. With smaller tile sizes and appropriately chosen stripe sizes, machines with smaller cache sizes will be able to minimize the memory bottleneck.

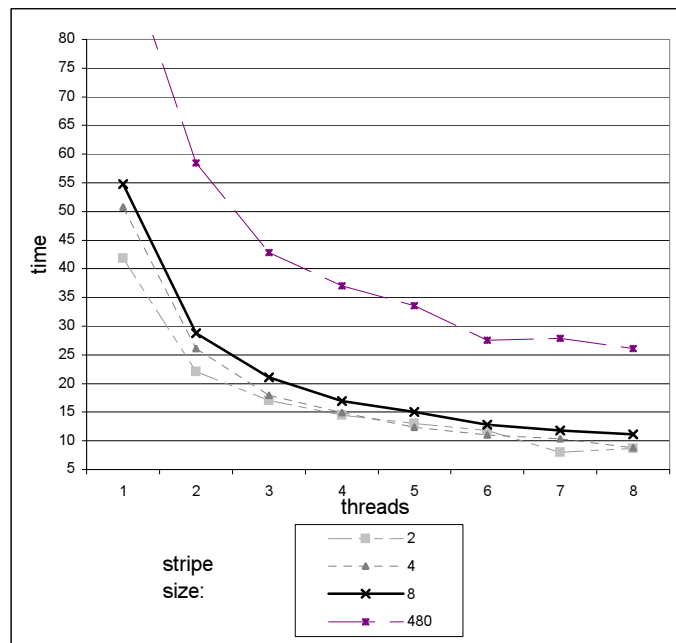


Fig. 12. Running times (seconds) for different configurations of stripe sizes and number of threads. A stripe size of 2 corresponds to performing comparisons one line of pixels (3840 bytes) at a time.

In Figure 13 and Figure 14 we show speedup results for the networked distribution and for the single-source distribution. For these results we have not used cache stripe optimization but compare full images. As the number of nodes are increased, the single source becomes a bottleneck limiting scalability. This is understandable since with a problem size of 316 images, the single source has to pump in the worst case 278 MB to each node of the cluster. Of course with tiling the actual number of images needed by each node is smaller than this worst case. The maximum delivered bandwidth we have observed using Stampede is 45 MB/s on Gigabit Ethernet. Thus with 14 nodes it would take  $\approx 100sec$  just for data distribution in the worst case.

As a point of comparison, the entire computation (without the cache stripe optimization), takes about 90s using a single CPU on a single node.

Figures 13 and 14 also show the ideal algorithm-limited speedup [23] due to the load imbalance inherent in our choice of tiles. Recall that the tile size decreases with the number of nodes. As can be seen, the bigger the tiles, the better the *compute : communication* ratio, yet the more jagged but steep is the algorithmic speedup curve. Because the algorithmic speedup accounts only for load balancing but not communication costs, if the tile size is 1, the algorithmic speedup curve will be a 45-degree line, since work can be distributed perfectly evenly. Needless to say this tile size is also the worst possible in terms of communication since all the nodes will need all the images.

The line labeled tiling-with-chaining shows the overall performance in the presence of networked distribution. With one worker per node (Figure 13), we observe speedup of about 7.1 for 14 nodes. As can be seen the corresponding algorithmic limit is 10.9. For eight workers per node we observe speedup of about 4.22 (Figure 14), while the algorithmic limit is 11.29.

Since the base with eight threads had a speedup of 5.75 that translates to an overall speedup of 24.26 on 112 processors.

We attribute the difference between 1-thread and 8-thread performance (7.1 vs. 4.22) to communication costs. Figure 15 compares the communication costs for one and eight threads as we vary the number of nodes in the application. As can be seen each thread in the 8-thread configuration spends roughly the same amount of time as the singleton thread in the 1-thread configuration on network I/O, leading to a reduction in overall speedup. The communication time is relatively stable (due to the application level multicast distribution tree) instead of increasing as is the case with a single source.

With some detailed instrumentation of the Stampede system we have been able to ascertain that most of the communication inefficiency is limited to prefetching the initial tile. During this phase, there can obviously be no overlap of computation with communication. This effect is exacerbated with eight workers per node over the effect with one worker per node and hence the corresponding disparity in the speedup curves. However, once the initial tile has been brought in there is good overlap of computation and communication due to the data mover thread at each node.

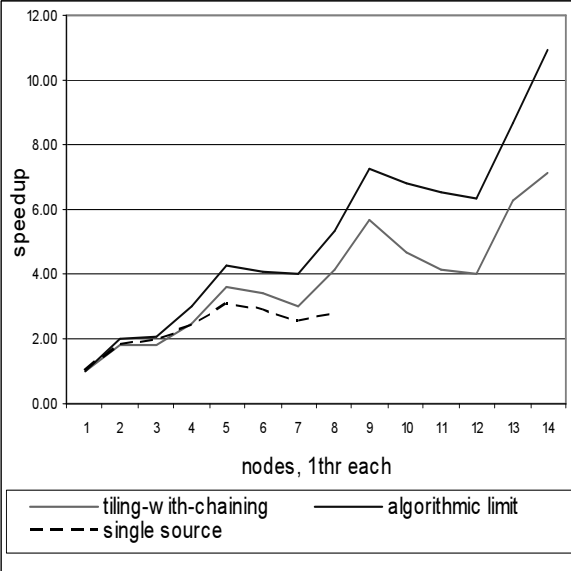


Fig. 13. Speedup across nodes for 1 thread per node

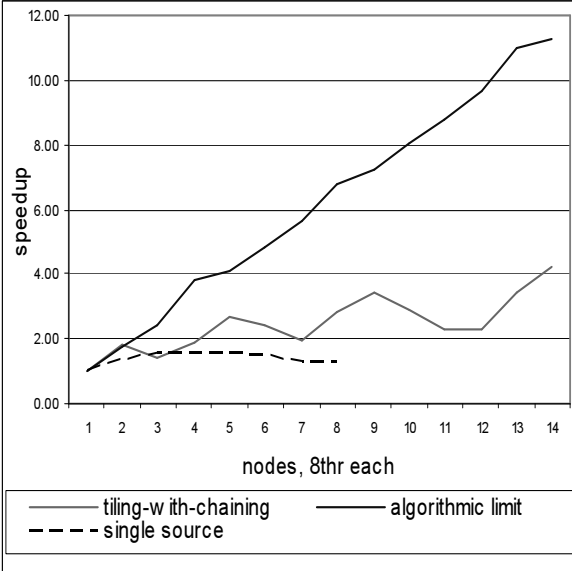


Fig. 14. Speedup across nodes for 8 thread per node

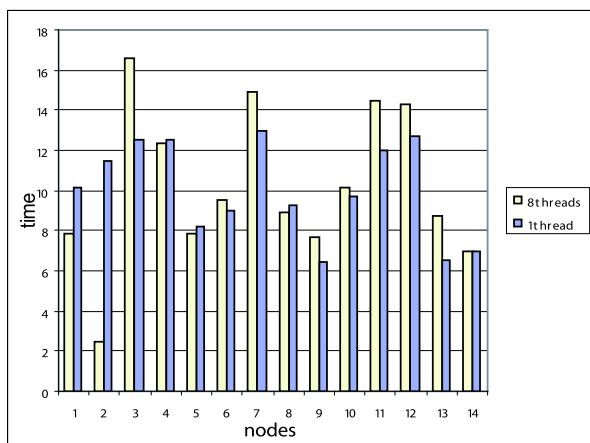


Fig. 15. *Inter-node communication (in seconds)*. Time spent by each thread in network I/O for 1-thread and 8-thread per node configurations as we vary the number of nodes from 1 to 14.

### VIII. CONCLUDING REMARKS

Stampede is a cluster parallel programming system with novel data abstractions designed to support emerging classes of complex interactive stream-oriented multimedia applications. Space-time memory (with its two variants channel and queues) provides a rich high level programming support to alleviate the programmer from low level details in developing such applications on a cluster computing platform. There are non-trivial systems issues in implementing this abstraction (most notably the garbage collection problem) efficiently in a cluster. We presented the details of the STM abstraction, programming examples to demonstrate its ease of use, and performance studies on a cluster of SMPs to show the implementation efficiency.

Directions for future research include asynchronous notification of item arrival on channels and queues, and multicasting support at the level of the abstractions.

### IX. ACKNOWLEDGMENTS

A number of people have contributed to the Stampede project. Bert Halstead, Chris Jeorg, Leonidas Kontothanassis, and Jamey Hicks contributed during the early stages of the project. Dave Panariti developed a version of CLF for Alpha Tru64 and Windows. Members of the “ubiquitous presence” group at Georgia Tech continue to contribute to the project.: Bikash Agarwalla, Matt Wolnetz, Hasnain Mandviwala, Phil Hutto, Durga Devi Mannaru, Namgeun Jeong, and Ansley Post deserve special mention. Russ Keldorph and Anand Lakshminarayanan developed an audio and video meeting application. Irfan Essa, and Arno Schoedl provided the background necessary for understanding the video texture application.

## REFERENCES

- [1] S. Ahuja, N. Carriero, and G. David. Linda and Friends. *IEEE Computer*, 19(8):26–34, August 1986.
- [2] H. E. Bal, A. E. Tanenbaum, and M. F. Kaashoek. Orca: A Language for Distributed Programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.
- [3] A. Chauhan and K. Knobe. A compiler driven execution model for irregular applications. In *Compilers for Parallel Computers (CPC2000)*, Aussois, France, January 4-7 2000.
- [4] A. D. Christian and B. L. Avery. Digital Smart Kiosk Project. In *ACM SIGCHI '98*, pages 155–162, Los Angeles, CA, April 18–23 1998.
- [5] K. Ghosh and R. M. Fujimoto. Parallel Discrete Event Simulation Using Space-Time Memory. In *20th International Conference on Parallel Processing (ICPP)*, August 1991.
- [6] IEEE. Threads standard POSIX 1003.1c-1995 (also ISO/IEC 9945-1:1996), 1996.
- [7] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.
- [8] S. B. Kang. A Survey of Image-based Rendering Techniques. Technical Report CRL 97/4, Cambridge Research Lab., Digital Equipment Corp., August 1997.
- [9] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. Winter Usenix*, 1994.
- [10] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott. VM-Based Shared Memory on Low-Latency Remote-Memory-Access Networks. In *Proc. Intl. Symp. on Computer Architecture (ISCA) 1997, Denver, Colorado*, June 1997.
- [11] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, May 1994. [www.mpi-forum.org](http://www.mpi-forum.org).
- [12] S. Microsystems. Javaspaces specification. In <http://java.sun.com/products/javaspaces/>, 1998.
- [13] R. S. Nikhil. Cid: A Parallel “Shared-memory” C for Distributed Memory Machines. In *Proc. 7th. An. Wkshp. on Languages and Compilers for Parallel Computing (LCPC)*, Ithaca, NY, Springer-Verlag LNCS 892, pages 376–390, August 8–10 1994.
- [14] R. S. Nikhil and D. Panariti. CLF: A common Cluster Language Framework for Parallel Cluster-based Programming Languages. Technical Report (forthcoming), Digital Equipment Corporation, Cambridge Research Laboratory, 1998.
- [15] R. S. Nikhil and U. Ramachandran. Garbage Collection of Timestamped Data in Stampede. In *Proc. Nineteenth Annual Symposium on Principles of Distributed Computing (PODC 2000)*, Portland, Oregon, July 2000.
- [16] 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 (LCPC 98)*, Chapel Hill NC, August 7-9 1998.
- [17] 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. ACM Principles and Practices of Parallel Programming*, May 1999.
- [18] J. M. Rehg, K. Knobe, U. Ramachandran, R. S. Nikhil, and A. Chauhan. Integrated task and data parallel support for dynamic applications. *Scientific Programming*, 7(3-4):289–302, 1999. Invited paper, selected from 1998 Workshop on Languages, Compilers, and Run-Time Systems.
- [19] J. M. Rehg, M. Loughlin, and K. Waters. Vision for a Smart Kiosk. In *Computer Vision and Pattern Recognition*, pages 690–696, San Juan, Puerto Rico, June 17–19 1997.
- [20] 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 CRL 97/2, Digital Equipment Corp. Cambridge Research Lab, April 1997.
- [21] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proc. 7th. ASPLOS, Boston MA*, October 1996.

- [22] A. Singla, U. Ramachandran, and J. Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *9th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1997.
- [23] A. Sivasubramaniam, A. Singla, U. Ramachandran, , and H. Venkateswaran. An application-driven study of parallel systems overheads and network bandwidth requirements. *IEEE Transactions on Parallel and Distributed Systems*, March 1999.
- [24] K. Waters and T. Levergood. An Automatic Lip-Synchronization Algorithm for Synthetic Faces. *Multimedia Tools and Applications*, 1(4):349–366, Nov 1995.
- [25] K. Waters, J. M. Rehg, M. Loughlin, S. B. Kang, and D. Terzopoulos. Visual Sensing of Humans for Active Public Interfaces. In R. Cipolla and A. Pentland, editors, *Computer Vision for Human-Machine Interaction*, pages 83–96. Cambridge University Press, 1998.
- [26] P. Wyckoff, S. McLaughly, T. Lehman, and D. Ford. T Spaces. In *IBM Systems Journal*, August 37(3): 453-474, 1998.

**Umakishore Ramachandran** received his Ph. D. in Computer Science from the University of Wisconsin, Madison in 1986, and is currently a Professor in the College of Computing at the Georgia Institute of Technology. His fields of interest include parallel and distributed systems, computer architecture, and operating systems. Currently, he is leading an NSF-ITR funded project that investigates the programming idioms and runtime systems for a distributed sensing infrastructure. He is the recipient of an NSF PYI Award in 1990, the Georgia Tech doctoral thesis advisor award in 1993, and the College of Computing Outstanding Senior Research Faculty award in 1996.

**Rishiyur S. Nikhil** has been at Sandburst Corporation since September 2000, directing the development of Bluespec (TM), a new language for ASIC design and verification. Previously, he was at the Cambridge Research Laboratory (DEC/Compaq), including a brief stint as Acting Director. Earlier, he was an Associate Professor of Computer Science and Engineering at MIT. He has published on functional programming, dataflow and multithreaded architectures, parallel processing, and high-level languages for ASIC systems design. He is a member of ACM, IEEE, and IFIP WG 2.8 on Functional Programming, and received his Ph.D. from the University of Pennsylvania.

**James M. Rehg** received his Ph.D. from Carnegie Mellon University in 1995. From 1996 to 2001 he led the computer vision research group at the Cambridge Research Laboratory of the Digital Equipment Corporation, which was acquired by Compaq Computer Corporation in 1998. In 2001, he joined the faculty of the College of Computing at the Georgia Institute of Technology, where he is currently an Associate Professor. His research interests include computer vision, machine learning, human-computer interaction, computer graphics, and distributed computing.

**Yavor Angelov** is a doctoral student in the College of Computing, Georgia Tech. He holds a BA in CS from the American University in Bulgaria and MS in CS from Georgia Tech. His interests are in Parallel and Distributed computing.

**Arnab Paul** finished his undergraduate in Computer Science and Engineering from Jadavpur University, India, and Master's from Indian Institute of Science, Bangalore. He is presently a Ph.D. student at College of Computing, Georgia Tech, Atlanta. His interests include distributed and parallel computing, fault tolerant computing and storage systems and information and coding theory.

**Sameer Adhikari** is a Ph.D. student in College of Computing, Georgia Institute of Technology since 1998. He finished his B.Tech from Indian Institute of Technology, Kharagpur, India. His research interest is experimental computer systems, primarily distributed systems for pervasive computing.

**Kenneth M. Mackenzie** received S.B., S.M. and Ph.D. degrees from the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology in 1990, 1990 and 1998, respectively. From 1998 to 2003 he was an Assistant Professor in the College of Computing at the Georgia Institute of Technology where he received an NSF CAREER award in 1999. In 2003 he joined the technical staff of Reservoir Labs, New York, NY. His research interests are in the areas of parallel systems, embedded computing and compilers

**Nissim Harel** is a Ph.D. student at the College of Computer Science at the Georgia Institute of Technology. He received his B.Sc. in electrical engineering from the Technion, in Haifa, Israel, in 1990, and his M.S. in computer science and M.Sc. in electrical engineering from the Georgia Institute of Technology in 1998 and 1999, respectively. He also co-founded in 2000, ClickFox, LLC, which develops and provides a decision support tool that helps optimizing interfaces of information systems in general, and web sites and IVR systems in particular.

**Kathleen Knobe** worked at Massachusetts Computer Associates (AKAC ompass) for over ten years where she worked on Fortran compilers for Alliant, Thinking Machines and MasPar. She subsequently returned to complete her education and received her PhD from MIT in 1997 with Professor Wm. Dally. She currently works at Cambridge Research Lab (Digital/Compaq/HP). Her research interests include compilation, runtime systems, streaming, parallelism and models of computation. She is currently investigating a new models of parallel computation based on streaming.

## APPENDIX

### A. Micromasurements

In this section we measure the performance of STM through a series of micromasurements. The intent is to quantify the overhead that STM incurs on top of the underlying messaging layers (CLF and MPI). Since we are targeting distributed multimedia applications, it is useful to consider the demands in such a scenario. A typical video camera delivers at 30 frames per second, where each image consists of  $320 \times 240$  pixels each having 24 bits of color. Thus each frame has 230400 Bytes and the total bandwidth at this rate is 6.912 MB/s. Typical inter-frame latency is 33.3 milliseconds.

**Experiment 1:** We compare CLF and STM latencies in this experiment. Two threads ( $T_1, T_2$ ) are spawned on two different nodes of the cluster.  $T_1$  sends a packet to  $T_2$ ; upon receiving this packet  $T_2$  sends back a packet of the same length to  $T_1$ . This send/receive is repeated for a large number of iterations and then one-way latency is computed at one of the ends as half the total time spent in those iterations. For CLF, the two threads communicate using its own send/receive API. A similar experiment for STM, uses two *channels* and *put* and *get* APIs for communication.  $T_1$  puts data into a channel collocated with  $T_2$  on one address space, and gets (and consumes) data from a second channel collocated with itself.  $T_2$  functions similarly. The latency is a measure of time taken from the starting point of a put (for an item) to the end of the consume operation for the same item. Figure 16 compares the latency (in microseconds) experienced in the CLF layer as a function of datasize with that incurred in STM. We varied the packet size up to 8KB which is the MTU for CLF. For each data size, we have plotted the minimum value of the latencies recorded over a large number of runs. Since STM is implemented on top of CLF, STM-latency is expected to be more than CLF latency. In particular, for one such latency cycle, the STM operations involve two or more roundtrip communications (via CLF) and number of thread synchronizations and context switches<sup>5</sup>. CLF-latency varies from 200 usecs (for 0.5 KB) to 730 usecs (8KB), while STM-latency varies from 420 usecs up to 1130 usecs for the same range. Although the processing overhead of STM is not constant across this range, the small difference between the slope of the two curves clearly suggests that STM incurs only nominal marginal processing cost with increasing data size. Figure 17 shows similar numbers comparing STM with MPI, and tells a similar story. The graph of MPI latency is little jagged; we ascribe

<sup>5</sup>Manipulating channel data structure involves acquire and release of locks, and remote requests to channels are fielded by server threads

this to the behaviour of MPI over the particular hardware. However, the overall conclusion about the performance of Stampede remains the same.

**Experiment 2:** We quantify the bandwidth usage through this experiment. A producer and a consumer are started on two different nodes. The producer keeps sending data packets to the receiver and the bandwidth is measured at the receiver's end as the rate at which the receiver gets the data. For the messaging layers, we also investigate one more variant of this setting where the receiver sends a zero byte ack to the producer every time a packet is received. The next packet is sent only after this ack is received. Notice, that any cross-address-space STM operation involves such roundtrip communication and hence we decided to employ this variant to make more meaningful comparisons. The STM scenario is implemented by two threads communicating via a channel collocated with the receiver. While the producer keeps putting data items on the channel, the consumer gets and consumes those items. Figure 18 plots the bandwidth numbers for implementation over CLF. It shows that CLF offers up to 30 MB/s of bandwidth. However, with an *ack* for every data packet, the bandwidth is reduced to 10 MB/s, while STM in that range of message size offers up to 4.9 MB/s. The reduction in STM is due to the overhead of byte copying and other synchronization incurred in the STM layer. Figure 19 captures a similar story for STM implemented on top of MPI. MPI offers a maximum bandwidth of around 21 MB/s at 8KB, and up to 9MB/s with an *ack*. The maximum bandwidth obtained from STM on top of MPI is around 4 MB/s. However, the processing overhead in the STM layer does not increase significantly with the data size as we have already seen in Experiment 1. For larger packet size, when transport latency dominates over the processing cost, STM attains the maximum bandwidth offered by the underlying messaging layer. This is illustrated in Figure 20 which shows the STM bandwidth obtained at image size payloads. Over CLF, STM achieves almost 30 MB/s and almost 21 MB/s over MPI, showing that it catches up with the maximum bandwidth offered by the respective messaging layers. In either case, these bandwidths are comfortably above 6.912 MB/s (bandwidth required for sustaining moderate resolution camera images at 30 frames/sec.).

### *B. Experiments with Color-Based Tracking*

Figure 21 shows the task graph for the Color Tracker application. Images are continually captured, digitized, and passed down the pipeline for analysis and target matching. The figure of merit in this application is two-fold: the latency for an individual frame to make it through the

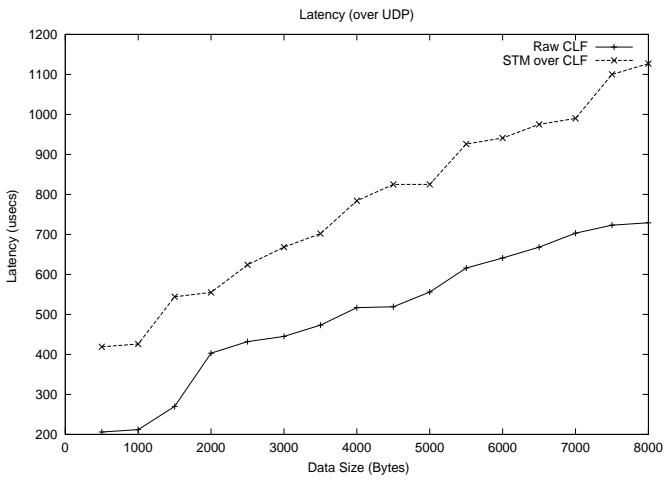


Fig. 16. Comparison of latency (CLF, and STM over CLF)

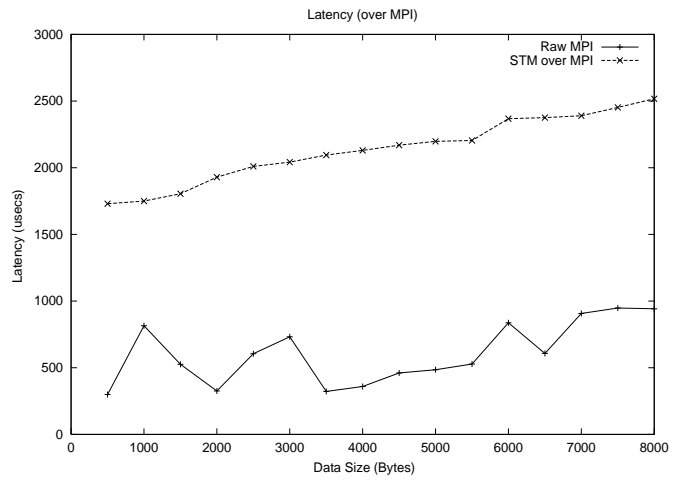


Fig. 17. Comparison of latency (MPI, and STM over MPI)

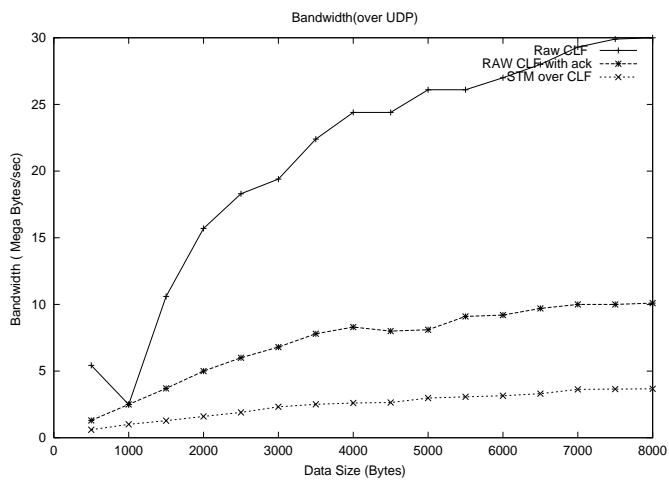


Fig. 18. Comparison of max. Bandwidth (CLF, and STM over CLF); Ack indicates CLF send with an ack from the receiver

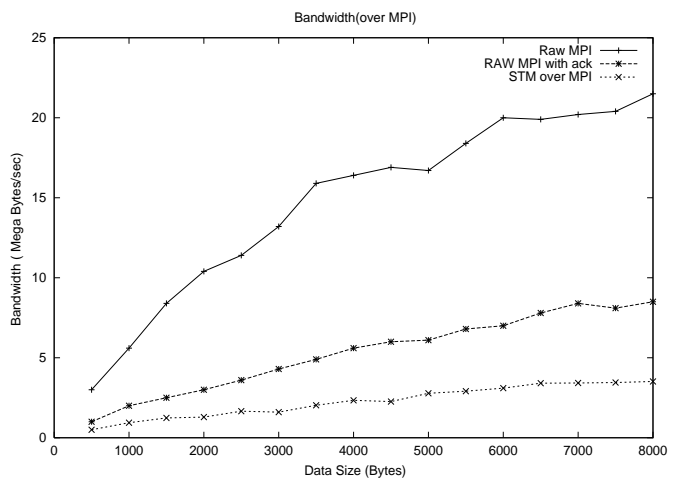


Fig. 19. Comparison of max. Bandwidth (MPI, and STM over MPI); Ack indicates MPI send with an ack from the receiver

entire pipeline, and the sustained frame rate that can be obtained. Thus effective utilization of the task and data parallelism inherent in the application is critical for its successful implementation on commodity hardware such as clusters of SMPs.

In an earlier work [18], we proposed an approach for an integrated approach to addressing task and data parallelism for such applications. We showed experimental results that validated this approach using a two node cluster of AlphaServer 4100's (four processor SMPs) interconnected by Memory Channel and running DIGITAL UNIX 4.0. In this subsection, we report the results of a new set of experiments which extend the analysis of this application in two ways. First, the experiments are based on a much larger, state-of-the-art cluster. Second, we provide much more extensive analysis of the bottlenecks in the application and the challenges they pose for efficient implementation using Stampede.

1) *Mapping the Color Tracker onto Stampede*: The color tracker's pipeline (see Figure 21) runs a number of different computational tasks, such as digitizer, histogram, change detection, and target detection. As a consequence of the varying computational requirements of these tasks and their interdependencies, it is not immediately obvious how to achieve maximum performance. This is in contrast to the video texture application, where the image comparison operation is clearly the bottleneck in a non-parallelized implementation. As a result, the tracker requires much more thorough analysis in order to determine how to improve latency and throughput. The first step in an efficient mapping is to exploit function parallelism by converting the separate tasks in Figure 21 into threads and introducing Stampede channels between them. The second step is to exploit data parallelism by determining the "bottleneck" stages in the pipeline and dividing their associated image processing tasks across multiple threads.

2) *Performance Concerns*: Data parallelization is achieved by replacing a thread with a group of  $K+2$  threads, where the  $K$  threads perform equivalent task to the original thread but on *chunks* of the input, rather than on the whole input, and the two additional threads are required to split the work into chunks (called a *splitter thread*), and to assemble back the work and produce a single output (called a *joiner thread*). There are several factors in determining the mapping of the original thread (assume 1 CPU) into a group of threads that scales up as additional resources are added (ideally, running  $K$  times faster):

- *Granularity of computation*. In other words, we determine how much the computation space can be divided among workers. In the tracker application, the amount of work at the Peak detection phase is  $O(frames \times models \times imagesize)$  which gives us 3 possible dimensions to partition. Since the time-sensitive dimension is *frames*, to minimize latency, we need to

partition across that dimension. However this strategy will only make the latency as low as that of a single node. In order to further lower the latency, the same frame may need to be sent to more than one worker at the expense of throughput. Once we have done partitioning across *frames*, further exploitation of data parallelism can be pursued by partitioning the other two dimensions.

- *Distribution of inputs and outputs.* If the thread to be data parallelized requires significant network bandwidth, then it is likely that the new  $K$  threads will face bandwidth problems as well. For example, if the original thread receives 30 frames per second, each of the new  $K$  threads will need to receive, depending on the partitioning strategy, the same 30 frames per second (in which case communication bottlenecks may develop, either at the splitter or at the original source of the data), or different pieces of each frame (which may cause the splitter *and* joiner threads to become a bottleneck, since the work of splitting/joining the regions of pixels may significantly limit the speedup). If the  $K$  threads are mapped on the same node of the cluster (as well as the splitter and joiner threads), then the amount of inter-node communication will be the same as with the original thread; if they are not, then we are facing a variant of the image distribution problem discussed in the Video Textures application and one possible solution would be to de-centralize the communication (which is what the distribution network really does).

In [18] the bottleneck was guesstimated to be the “target detection” task, and it was parallelized at two levels - across the nodes of the cluster (using an *outer* splitter and joiner) and further inside each node using an *inner* splitter and joiner. Using our measurement infrastructure in Stampede we measured all “put” operations in every thread of the pipeline and confirmed that the slowest rate was at the “target detection” task (called Peak Detection in our pipeline).

Since parallelizing entirely on a single machine would still not make the target detection run fast enough to keep up with the faster threads earlier in the pipeline, we preserved the outer/inner splitting strategy. For each node that we ran the inner splitter and joiner we placed 6 inner worker threads. The inputs to the outer splitter (which had to be sent to each *inner* splitter), were  $2 \times 500\text{KB}$  images and a small control structure; therefore the inner splitters were configured to read the images directly from the histogram and the background detection threads bypassing the outer splitter and the outer splitter would only create “chunks” describing the work to be performed. After timing the various other threads we concluded that all threads, other than the ones running on the “inner” nodes, could be placed on a single node without creating a bottleneck.

Figures 22 and 23 give the actual layout of the pipeline in terms of Stampede abstractions.

3) *Experimental Setup*: The input is a continuous stream of images each of size 1 MB and a set of 8 models to track in each image. The space to be partitioned is the cross product of the models and the pixels of each frame; for that purpose we have adopted the notation MP and RP to denote the number of subsets the *Models* are partitioned into, and the number of *Regions* a frame is divided into, respectively. Due to the two levels of splitting, each parallelization strategy was described by the values of Out\_MP, Out\_RP, In\_MP and In\_RP. We did not want to parallelize parts of different frames because this would increase the overall latency of the pipeline. The models are partitioned into sets and the frames are partitioned into “regions” (horizontal stripes, similar to the VT application). Experimentally we have determined that it is better to partition the models into as many sets, called *model partitions*, as possible, so each node will get fewer models in its model partition. Consequently, we set the *Out\_MP* as high as possible and have experimented only with 1, 2 and 4 node configurations that divides the set of 8 models evenly. In addition, because we wanted to minimize latency on the outer level, the number of chunks created by the outer splitter ( $Out\_MP * Out\_RP$ ) had to match the number of inner splitters.

Therefore, for the space of configurations we wanted to explore (1,2,4 nodes varying 1 to 6 threads in each), the parallelization strategy always is to divide the models into as many sets as there are inner nodes, and on the inner level, to set the In\_RP to the number of threads in the particular configuration.

Since each inner splitter reads in 1 MB/frame, as the frame rate scales up and approaches the network capacity we expect the speedups to plateau.

#Nodes	#Threads per node					
	1	2	3	4	5	6
1	0.78	1.26	2.09	2.81	3.43	4.08
2	1.53	2.40	4.03	5.23	5.28	5.32
4	2.65	4.46	5.20	5.31	5.30	5.26

TABLE I

FRAME RATES (FPS) FOR THE COLOR TRACKER

4) *Results and Discussion*: The frame rate and end-to-end latency results for the tracker are summarized in Tables I and II. A quick look at Table I shows a bottleneck at  $\approx 5$  fps. Table II gives the end-to-end latency of the pipeline which corresponds nicely to the frame rate. The best figures of  $\approx 221$  msecs is achieved by the same set of configurations we get the best frame rate from. For all columns or rows where the frame rates is below this peak rate the speedup is nearly linear – the 1-node configuration scales up for 6 threads to 5.23, the one thread and two

#Nodes	#Threads per node					
	1	2	3	4	5	6
1	1577.9	1052.6	593.0	436.4	358.3	298.0
2	815.2	517.5	296.7	226.7	221.9	221.8
4	472.7	276.0	225.6	221.1	222.7	220.9

TABLE II

END-TO-END LATENCIES (MILLISECONDS) FOR THE COLOR TRACKER

thread configurations scale up for 4 nodes to 3.4 and 3.54 respectively.

The 5 fps plateau is clearly not due to network limitation ( $5 \text{ fps} \times 4 \text{ nodes} \times 1 \text{ MB} = 20 \text{ MB/s}$  is well below the achievable throughput of Stampede on CLF as presented in Section A). Therefore, we investigated the measured rates of execution for all the threads in the pipeline. This investigation revealed that the histogram thread (which precedes the parallelized threads in the pipeline), was executing at  $\approx 5$  fps.

Although we have found a set of configuration for which the pipeline runs at peak performance (2 nodes, 4 threads per node), we see 2 possibilities for further exploiting computation resources to improve the performance:

- *Improve latency.* This would require parallelizing the histogram thread using a similar strategy of replacing it with a set of threads, a splitter and a joiner.
- *Get more work done.* This would simply mean to increase the problem size without affecting the performance of the histogram. One example is increasing the number of models, which will decrease the frame rate of the target detection part below the 5fps bottleneck, since more work needs to be done, and configurations with more than 2 nodes will need to be explored.

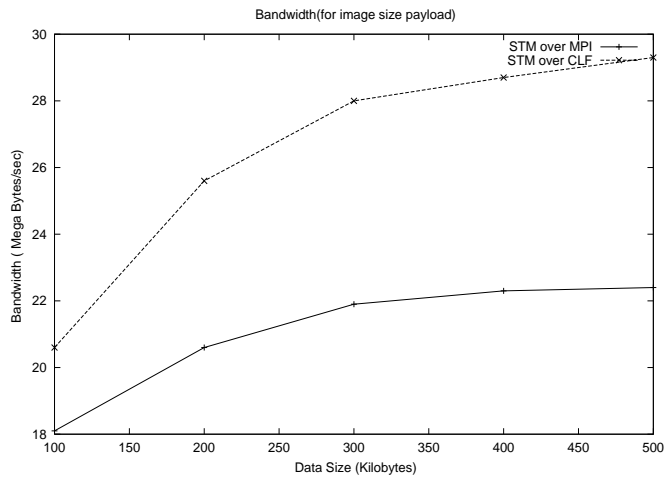


Fig. 20. Bandwidth offered by STM at image size payloads

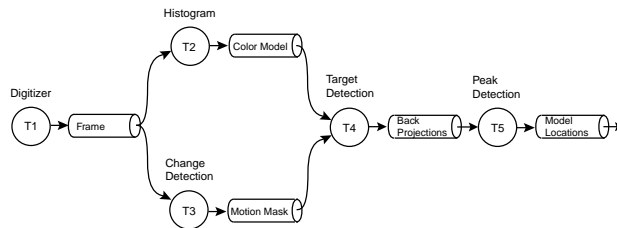


Fig. 21. Task graph for the color-based tracker. Ellipses denote *tasks*, implemented as threads. Rectangles denote *channels* which hold streams of data flowing between tasks.

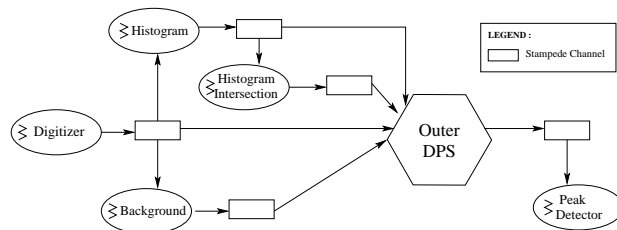


Fig. 22. Tracker Pipeline: See Figure 23 for Outer DPS

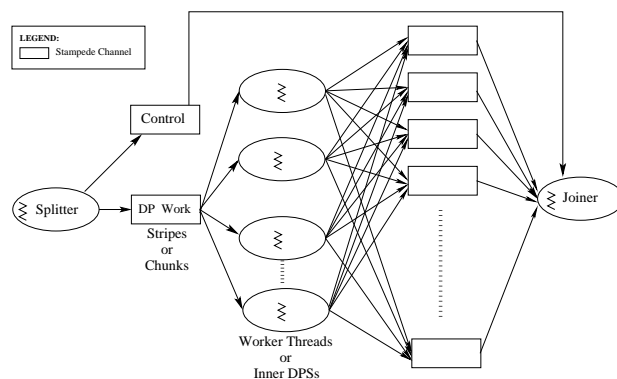


Fig. 23. Tracker Data Parallel Sub-graph: Used for both the Outer DPS and the Inner DPS