A Distributed and Parallel Component Architecture for Stream-oriented Applications

P. Barthelmess and C.A. Ellis

Department of Computer Science, University of Colorado at Boulder, Campus Box 430, Boulder, CO 80309-0430, USA. {barthelm,skip}@colorado.edu

University of Colorado Technical Report CU-CS-989-04

Abstract. This paper introduces ThreadMill - a distributed and parallel component architecture for applications that process large volumes of streamed (time-sequenced) data, such as is the case e.g. in speech and gesture recognition applications.
Many stream-oriented applications offer ample opportunity for enhanced performance via concurrent execution, exploring a wide variety of parallel paradigms, such as task, data and pipeline parallelism. ThreadMill addresses the challenges of development and evolution of parallel and distributed applications in this domain by offering a modeling formalism, a programming framework and a runtime infrastructure. Component development and reuse, and application evolution are facilitated by the isolation of communication, concurrency, and synchronization concerns promoted by ThreadMill: 1) communication between components is mediated, so that components are oblivious to who their peers in an application are, allowing them to be composed in unanticipated ways, in different contexts; 2) concurrency is exogenous to components and can be controlled via an integrated staging mechanism that affords detailed control of distribution across multiple address spaces and/or concurrency within each address space; 3) synchronization is effected via an extensible set of orchestration operators, that embed recurrent coordination patterns that are matched to the requirements of stream-oriented applications.
A direct consequence of the novel mechanisms introduced by ThreadMill is that applications composed of reusable components can be re-targeted, unchanged, and made to run efficiently on a variety of execution environments. These environments can range e.g. from a single machine with a single processor, to a cluster of heterogeneous computational nodes, to certain classes of supercomputers. Experimental results show an eightfold speedup when using ten nodes of an AlphaServer DS20 cluster running a proof-of-concept 2D video-based tracker for hands and face of American Sign Language signers.

1 Introduction

ThreadMill is a distributed and parallel component software architecture that caters for the specific requirements of applications that process large volumes of streamed (i.e. time-sequenced) data. Such applications surface e.g. in the context of the analysis of streams generated by humans as they interact with each other and/or their environment.

Applications in this target domain can many times be naturally expressed in terms of a pipe-and-filter paradigm. From the point of view of concurrency these applications, once appropriately structured, can explore a variety of concurrency modalities, e.g.: the *pipeline parallelism* that emerges from the simultaneous processing of new elements in the first stages of the processing pipeline while previous elements are still being processed by other stages; *task parallelism* that surfaces when the same data is analyzed concurrently by more than one filter; and *data parallelism* associated to the concurrent processing of multiple sub-problems (corresponding e.g. to multiple hypotheses) by replicated instances of a filter.

These multiple opportunities for parallel processing offer a potential for enhancing the performance of this class of applications via distribution and concurrency. Many stream-oriented applications process large volumes of data and/or employ computationally demanding processing techniques. This in many cases results in lengthy processing times if a single processor is employed. Enhanced performance opens up the possibility for many applications to become usable in settings that require fast response time, such as e.g. advanced user interfaces and smart environments.

While the opportunities for concurrency abound, developing code that takes advantage of that is not in general easy. Parallel and distributed code many times embed assumptions about the structure and characteristics of the execution environments on which it runs. These assumptions tend to permeate the architectural scaffolding of applications, which makes it hard to reuse parts of an application in different contexts, e.g. in different execution environments.

ThreadMill offers a modeling formalism, a programming framework and a runtime infrastructure that aim at facilitating the development and evolution of efficient distributed and parallel stream-oriented applications. ThreadMill's goals are to promote the reuse of existing application code while at the the same time affording detailed control over performance aspects.

ThreadMill's approach to development, reuse and control over evolution aspects is based on mechanisms that promote the separation of communication, concurrency and synchronization concerns from application code. These three aspects are made exogenous to application components and can thus be manipulated in separate without affecting application code within components.

ThreadMill takes a three-pronged approach to the separation of communication, concurrency and synchronization concerns:

1. Components communicate exclusively by indirect exchange of messages. The recipient of a message is unaware of who the sender is; conversely, the sender of a message is unaware of who the recipient(s) are. Because components do not address each other directly, they can be composed in unanticipated ways, e.g. by having the results produced by a component be transparently delivered to multiple consumers without requiring any special action on the part of the component.
2. An integrated staging mechanism affords developers the detailed control over distribution of components among different address spaces and (thread-based) concurrency within each address space. This first class staging mechanism addresses the need for structuring application deployment in such a way that the communication of the commonly large data structures is optimized by locality and accounts for differences e.g. in processor and network speeds.
3. Synchronization functionality is factored out of components and embedded into reusable components, resulting in components that can be potentially used in a broader range of contexts. The open-ended nature of the architecture, that allows for new synchronization operators to be defined, is based on ThreadMill's reliance on a generic substrate offering introspection capabilities.

This paper's presentation is structured around the description of ThreadMill's three main aspects: 1) The *generic component model* and graphical formalism; 2)The *meta-machine*, in terms of which the semantics of the model are expressed; and 3) The *synchronization mechanism* tailored to the specific requirements of stream-oriented applications. This mechanisms is built on top of the generic functionality of the model, taking advantage of the architectural extensibility mechanisms of ThreadMill. These three aspects are presented in Sections 2 to 4.

The paper then presents results of experimental evaluation of the model and its implementation (Section 5) and related work (Section 6). The paper ends with Conclusions and Future Work (Section 7). An extended version of this paper is presented in [4].

2 The ThreadMill model

ThreadMill offers a generic component-based model that is the foundation both for the development of custom applications and for the implementation of domain-specific synchronization operators. In this section we present this generic model. The underlying semantics of the model presented here is then detailed in Section 3, in the context of ThreadMill's meta-machine. Section 4 shows how this generic model can be customized to handle the specific synchronization requirements of stream-oriented applications.

Developing a ThreadMill application involves the following phases, that in most cases will be iterated:

- **Component development:** application components are coded or existing components are identified and reused, including synchronization operators.
- **Application composition:** components are composed into a *configuration graph* that determines how components communicate, based on data-dependencies among them.
- **Staging definition:** based on the characteristics of a target execution environment (e.g. number of processors, processing speed, network speed), a specific *configuration version* is defined. Configuration versions add annotations to configuration graphs to determine threading and/or placement of components across one or more address spaces.
- **Code generation:** Configuration versions are processed by a ThreadMill compiler, that generates a SPMD (Single Process Multiple Data) style code, that is ready to be deployed in one or more processors.

In the following sections, we present the basic elements of the generic model (Section 2.1), and the show how applications can be composed from reusable components (Section 2.2). We then discuss how the execution behavior of an application can be defined through the integrated staging functionality (Section 2.3).

2.1 Fundamental elements

All communication in the model takes the form of asynchronous messages (or tuple exchanges) among components, as described in the following:

- **Components** implement application code and can be specialized to perform specific synchronization functions. A component is characterized by a signature $(s, \{ip\}, \{op\})$ where s is a state, and $\{ip\}, \{op\}$ are sets of input and output ports respectively. A component *produces* or *writes* tuples to its output ports and *accepts* or *consumes* tuples from its input ports.
  Components attach *handlers* to each of their input ports. These handlers apply unspecified transformations to the state and may post tuples to one or more of the component's output ports. The state is private to each instance of a component and accessible to all of the instance's handlers.
  In practice, defining a component is similar to defining a class in an object-oriented language. Writing a component therefore involves defining a state and writing a set of handler functions. Function signatures are required to comply to a ThreadMill mandated format. Any language can used to write components, provided that it supports C-style calling conventions.
- **Connectors** are conceptual entities that are said to *carry* tuples between a pair of component ports. A connector *receives* tuples from the output port of a component a and *relays* them to the input port of a component b.
  Multiple connectors can be attached to a single input port. Incoming tuples are in general dealt with by a consumer component as if they were relayed by a single connector.

The introspection mechanisms (Section 3.4) might be used by advanced components to distinguish messages sent by different components.
Conversely, multiple connectors might be associated to each output port. Tuples written by a component to one of its output ports are replicated into as many copies as there are attached connectors[1]. Selective delivery can be attained via instrospection mechanisms (Section 3.4). Again, the introspection mechanism can be used by advanced components to determine which components are connected to an output port. Messages then can be selectively delivered to one (or a sub-group) of components that are attached to an output port, rather than to all of them.
Notice that this architectural mechanism allows for results of a component to be propagated to multiple consumers without the need for the generating component to be directly concerned with this delivery.

– **Tuples** are units of data that are communicated among components. A tuple type determines a sequence of field names and field types. Tuples represent partial results that flow among components and are further processed and refined as they flow through a configuration graph, until some final processing objective is reached.
Tuples might represent complex data structures by embedding references to other structures (which in turn might include references themselves).

Communication among components is in general location and identity transparent. Components are not required to be aware of who the recipients of tuples they generate are and where these recipients are located. Conversely, components receiving a tuple can be oblivious of who the sender of the tuple is or where the sender is located. Those advanced components that require more control over the way the communication is effected can make use of ThreadMill's introspection facilities (Section 3.4) for instance to perform delivery to a selective group of connectors attached to a port.

2.2 Composing applications - configuration graphs

ThreadMill applications are defined by composing components into *configurations*. Configurations are directed graphs in which nodes represent components and edges represent connectors.

Figure 1 shows a configuration graph that represents a 2D vision-based Joint Likelihood Filter (JLF) tracker [28]. This application tracks the position of the hands and face of American Sign Language signers from frame to frame of a video input (acquired by *Reader*). The method consists of evaluating multiple hypotheses based on random samples around positions predicted by kalman filters against evidence provided by a skin-color filtered video frame (details are out of the scope of this paper - see [28] for details of the method and [6] for details of the ThreadMill implementation).

Components in Figure 1 are distinguished according to their types. Rectangles represent reusable application code. Ellipsoids (*channels*) and triangles (*join operators*) are synchronization operators. The different types of components are discussed in detail in Section 4. For the purpose of this section, it suffices to say that the nodes are components that perform actions on the tuples they receive through their input ports, and that may in turn produce tuples through their output ports, according to some logic that is private to the component, i.e. the tuple exchanges are the only observable results of component execution.

Ports are represented as small triangles embedded on one or more sides of a component. Input ports are indicated by triangles that point into components; output ports are indicated

[1] In practice, there is no copy involved unless a consumer is located in a different address space. Tuple flow is performed by transmitting references, so what actually takes place is that the same reference is sent to all consumers.

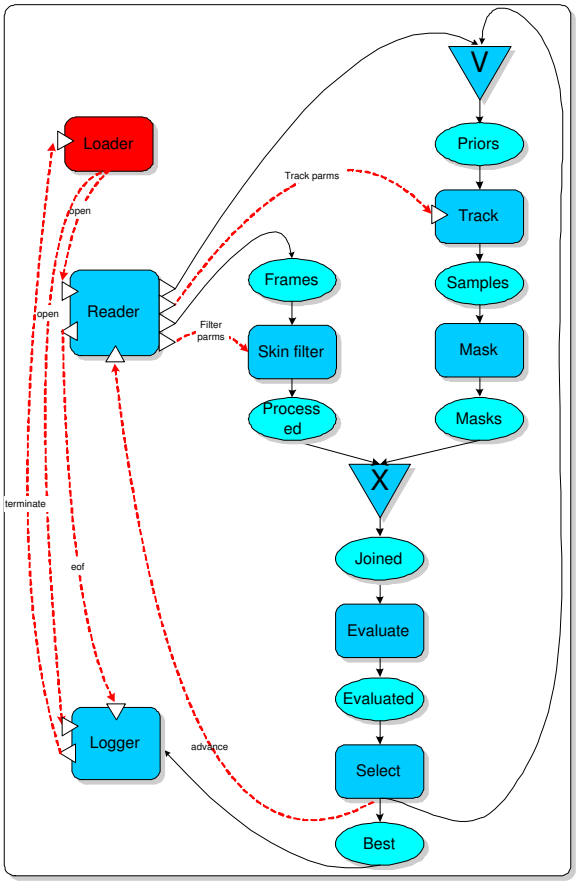

Fig. 1. ThreadMill representation of a JLF tracker.

by triangles pointing out of components. In many occasions explicit port representations are dropped and it is then assumed that connectors that are directed into components are made to input ports and that connectors originating on a component are attached to output ports.

2.3 Staging - configuration versions

The configuration graphs examined in the previous section (Section 2.2) indicate potential for concurrency through the partial ordering that is made explicit by the graph. ThreadMill does not associate a concurrency semantics to these graphs, though. Concurrency issues are dealt with in a separate phase, in which one or more *configuration versions* are defined. A configuration version is built by applying transformations to a configuration graph to specify the concurrency characteristics of an application. Configuration versions are matched to the characteristics of specific execution environments, determining for instance how many address spaces are to be created; which components are placed within each address space; how threads of execution are to be assigned within each address space; and how many instances of each component are to be created.

Currently, developers are responsible for specifying a desired concurrency semantics by creating versions that consider application and environment characteristics. Refinement and fine-tuning of configuration versions is facilitated by the ease with which these versions can

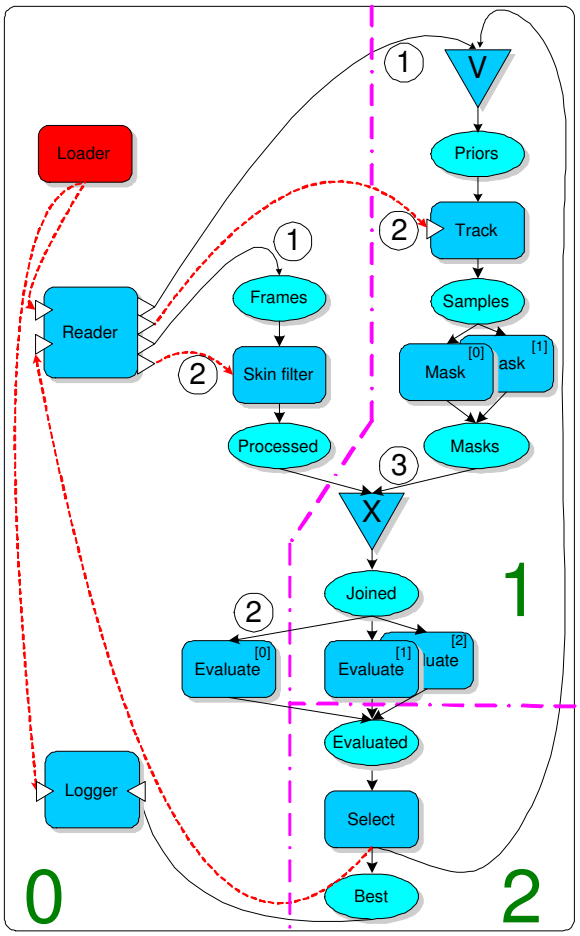

Fig. 2. A configuration version for the JLF tracker application.

be created in ThreadMill. That allows developers to experiment with multiple versions to determine effective ones. The reflective nature of ThreadMill's execution mechanism makes it possible in principle to explore self-tuning and dynamic execution (this is briefly considered in Section 7.

Staging related issues are treated as first class concerns by ThreadMill because of the impact that they have on performance. Efficiently implemented intra-address space communication (via hardware shared memory) can be orders of magnitude faster than communication between address spaces particularly if the communication has to be performed over a network connection. Placing components that communicate intensively and/or exchange large volumes of data into the same address space has therefore important efficiency consequences. In fact, given the same algorithm, this is the single most important factor determining whether the performance of an application will be acceptable or not.

The ideal concurrency and placement of components of an application are determined by a variety of factors that include characteristics of the application, such as computational costs of each of its stages, size of exchanged data structures, as well as characteristics of the execution environment, such as number of available processors, processor and network speeds, availabil-

ity of hardware supported shared memory. Configuration versions allow developers to use their knowledge of the domain and environment to determine efficient runtime organizations.

For each configuration graph describing an application, a family of configuration versions can be created. These different versions determine how the application is to be deployed under different execution circumstances. The same application can thus be tuned for execution in a single processor machine, or distributed over many multiprocessor nodes of a computational cluster without requiring any changes to component code.

Configuration graphs can be tuned by developers to optimally explore the resources available on a specific execution environment, be it a single-processor machine or a supercomputer. On the limit, an application can be made to execute strictly sequentially, under a single thread of execution under a single processor. Or this same application can be distributed so that each component executes under a private address space each running under its own processor. Most versions will in practice combine distribution of components with thread concurrency within each of the distributed address spaces.

Figure 2 shows a configuration version derived from the configuration graph of Figure 1. This version is tailored to execute distributed over three address spaces, which will most likely run under distinct computational nodes. The purple dotted lines in Figure 2 represent address space boundaries and therefore define the placement of components within these spaces. The number-labeled circles that annotate connectors represent thread assignments (only a few are shown). Notice as well that some components, namely *Mask* and *Evaluate* are instantiated multiple times, to take advantage of potential data parallelism offered by these specific stages.

ThreadMill supports the following staging operations:

- **Placement** specifies address space boundaries, determining which components are to be executed within each address space.
  Boundaries are indicated in Figure 2 by purple dotted lines; each address space is labeled with a tag 0, 1 and 2. Each address space can be mapped to a separate node of cluster, for instance.
  For multiprocessor machines that provide hardware shared memory support (such as commonly available Symmetric Multi Processor - SMP - machines), each address space can be executed under a single machine, independently of its number of processors. Multiple processors are taken advantage of via threading, and communication is optimized to make use of the fast shared memory accessible to threads executing under these multiple processors. Threading is specified by the *binding* operation.
- **Binding** determines the concurrency to be applied to elements of the same address space. Binding associates threads of execution to individual connectors, so that tuples can be concurrently processed by the consuming components.
  Binding is indicated in Figure 2 by numbered circle labels attached to connectors - only a few are actually shown in the figure for reasons of legibility. In practice, every connector is labeled with a binding tag.
  Thread assignment via binding as proposed by ThreadMill allows for a fine level of control over concurrency within an address space. All tuple processing can for instance be associated with independent threads, exploring maximum concurrency, or can be made to execute under a single thread, i.e. strictly sequentially. Intermediate solutions are also possible, for instance assigning independent threads for computationally intensive components and having other low priority components process tuples under one or more shared threads.
- **Unfolding** provides for components to be instantiated multiple times, to enhance concurrency of selected processing stages. The multiple instances will compete for tuples, that are therefore concurrently processed, rather than consumed one by one by a single consumer.

The actual mechanism that makes this distribution possible is embedded in a synchronization operator - the *channel* - that is detailed in Section 4.2.

Unfolding is graphically represented as multiple instances of an component, each of which is labeled with an index number within brackets (e.g. *Mask* and *Evaluate* in Figure 2).

Notice that *placement* and *binding* are orthogonal, in the sense that it is possible to place unfolded instances in multiple address spaces, as is the case for *Evaluate*, that has instances in spaces 0 and 1.

Configuration versions are processed by a ThreadMill compiler that converts them into tables containing meta-information that drives the execution of an application. The compiler generates a single-image, SPMD (Single Process Multiple Data) style of code that loads the meta-information and initiates execution of the runtime infrastructure. The runtime infrastructure is based on the semantics of a meta-machine, that we introduce in the following section (Section 3).

3 The meta-machine and runtime infrastructure

The semantics of the configuration versions described in the previous section (Section 2.3) are defined in terms of the operations of a reflective machine. In particular, the exogenous communication and concurrency that are essential to ThreadMill's approach to reuse and evolution are effected via operations of the machine. This semantics is in turn implemented by ThreadMill's runtime infrastructure, accessible to application code via a programming framework.

The ThreadMill's machine is an interpreter of configuration versions (Figure 3). The structure of the communication graph and annotations, represented as relations stored in a meta-store provide the instructions that drive the machine.

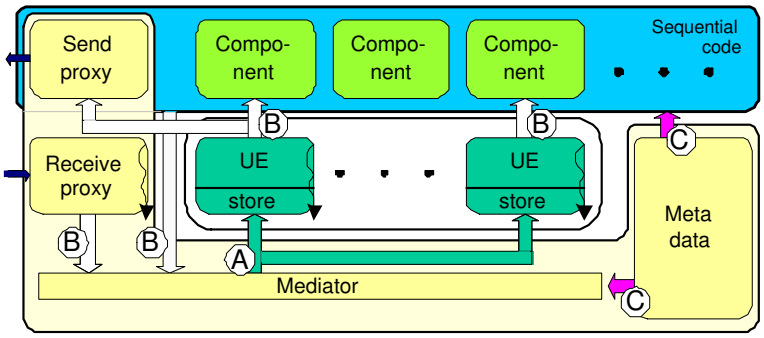

Fig. 3. The ThreadMill meta-machine.

We begin the presentation of the mechanism by introducing the data structures that represent configuration versions, as converted by the ThreadMill compiler (Section 3.1). Section 3.2 presents the machine's functional elements. Section 3.3 presents the machine's operational semantics. The introspection facilities provided by ThreadMill for writing of advanced components is presented in Section 3.4. Some of the practical aspects of the implemented infrastructure are presented in Section 3.5.

3.1 Reflective structures

A configuration version is expressed within the meta-machine in terms of five functions: *mapping, placement, binding, handler* and *state*:

- $Mapping(c, op)$ is a function that takes a component identifier c and an output port identifier op and returns a set of tuples $\{(c'_i, ip_i)\}$ that identify the components and their input ports to which op is attached by a connector.
- $Placement(c)$ identifies the address space within which a component c has been placed.
- $Handler(c, ip)$ maps input port ip of a component c to a function handler that is attached to the input port ip and processes messages that are written to ip.
- $Binding(c, op, c', ip)$ returns the thread identifier associated to the connector that attaches the output port op of component c to the input port ip of component c'.
  Notice that the binding allows for the assignment of individual threads to each connector.
- $State(c)$ returns a reference to the private state associated with component instance c.
- $SpaceUE(n)$ returns the number of *units of execution* that are to be created in a specific address space n.
- $Local$ is a constant that uniquely identifies an address space.

These different functions are used by the machine's mechanisms to obtain the information that is used to drive the execution of an application, as specified in the following section.

3.2 Functional elements

Figure 3's diagram represents the logical structure of a single address space. Execution involves one or more instances of such machine that communicate with each other to transparently deliver messages across address space boundaries. In this diagram, green arrows (labeled "A") represent asynchronous posting of tuples to multi-threaded units of execution; white arrows (labeled "A") represent function calls; purple arrows (labeled "C") represent accesses to the meta-information that drives the machine. Rectangles are functional blocks; crooked arrows attached to these blocks represent threads, i.e. the functional blocks marked with crooked arrows can execute concurrently. In this section, we informally describe the semantics. The detailed algorithms are discussed in Section 3.3.

1. The mediator is the core element of the machine, and acts as the intermediary between all components, effectively driving the execution as it processes (concurrently) the tuples that are posted by sequential application code, causing component handlers to be activated (via the units of execution), which in turn might result in further generation of tuples. The mediator is based on a *mediator/observer* pattern [14].
   Upon receipt of a tuple t, the mediator determines to whom it should be delivered by examining the *mapping* relation. Given a component c and an output port op through which the tuple was posted, the mediator determines the set of components and input ports $\{(c'_i, ip_i)\}$ to which the tuple is to be delivered. For each such recipient the mediator assembles an *activation record* $ar = (h, t, s)$ where $h = handler(c'_i, ip_i)$, and $s = state(c'_i)$.
   Each activation record is further associated to an *address space* $as = placement(c'_i)$ that determines under which address space the handler is to be processed. Activations that are to take place in an address space different from the current one are transfered to the *send proxy* (see item 3) for relaying to the foreign space. Activations that are local, i.e. take place in the current address space are placed into the store of the unit of execution $ue = binding(c, op, c'_i, ip_i)$ for processing (see item 2).
   Notice that multiple components (including the receive proxy) might call the mediator concurrently to have tuples delivered asynchronously to the units of execution. The mediator is reentrant and can handle these calls concurrently.

2. The *units of execution* implement the *asynchronous delegate* pattern [14] being therefore responsible for introducing concurrency into the execution. Each of them corresponds to a thread of execution (indicated in Figure 3 by the crooked arrow that annotates them).
   Units of execution retrieve activation records (h, t, s) placed by the mediator into their private stores and invoke the indicated component handler h, passing tuple t and the reference to the state s as parameters. This effectively implements the semantics of a connector, performing the transfer of a tuple to a component for processing under a specific binding. The execution of the handler may result in further messages being posted to the mediator, that will dispatch them according to what has been explained in item 1 above.
   A handler under execution has control of the thread associated to the unit of execution until its completion. Once the handler runs to completion, the unit of execution retrieves another activation record, if available, and repeats the process described above.
   The number of units of execution and by consequence the number of threads of a machine can be programmed individually for each address space, as part of the machine's initialization.
3. The *send proxy* handles the transmission of tuples to components located in other address spaces. The transmission is effected by sending a message to the *receive proxy* (item 4) of a machine controlling another address space. A marshaling mechanism automatically converts data to/from a wire format. This mechanism performs a recursive descent through pointer structures (using tuple-structure meta-data), sending out each part of a complex data structure to a remote space where they are reassembled. Embedded pointers are replaced by unique system-wide identifiers. Recipients patch pointers with local addresses of cached copies based on structure descriptors. ThreadMill adopts a single assignment policy (common in message-passing systems) to avoid costly replica-synchronization operations. This mechanism is further detailed in Section 3.5.
4. The *receive proxy* monitors inter-machine connections for messages. Once a message from a foreign address space is received, the receive proxy posts it to the local mediator for dispatching, thus transparently effecting the activation of a local handler under the command of a remote machine.

3.3 Operational semantics

Algorithms 1 to 5 present the pseudo-code of the machine operations that were briefly described in a previous section.

To start processing, a marker tuple $init$ is posted as part of a machine initialization routine (Algorithm 1). This tuple is by definition mapped to the initial behavior that is desired, and causes the activation of one or more component handlers which in turn produce further tuples that are translated by the mediator as described in Section 3.2. A variable number of *units of execution* (UEs) are created as part of a machine's initialization. The number of UEs to be created is given individually for each address space by the $SpaceUE(\cdot)$ function (Section 3.1). $Local$ is a constant that is set to a unique address space identifier under which a specific machine instance is executing.

Algorithm 1 Node initialization - units of execution

for $i = 1$ to $SpaceUE(Local)$ **do**
  $U \leftarrow U+$ new unit i
end for
$Post(init)$

Post is the main service activated by components to communicate (indirectly via the mediator) with each other (Algorithm 2).

Algorithm 2 Post(c, op, t)

```
for all (c', ip) ∈ Mapping(c, op) do
  n ← Placement(c')
  if n = Local then
    if m ≠ stop then
      u ← Binding(c, op, c', ip)
      s ← State(c')
      h ← Handler(c', ip)
      S[u] ← S[u] +_safe (h, t, s)
    else
      for all ue ∈ U do
        S[ue] ← S[ue] +_safe stop
      end for
    end if
  else
    Send(c, op, n, t)
  end if
end for
```

Postings are handled by the mediator, that reacts to a posting in three distinct ways, depending on whether a tuple is a regular tuple or the special marker $stop$, and whether the handling of a tuple takes place locally or remotely:

1. **Handling of a regular tuple taking place within the local address space managed by a machine instance**
   In this case, the mediator reacts by inserting zero or more *activation records* into the *stores* of zero or more UEs, according to the meta-data that establishes message translations. Because multiple components might post tuples concurrently, the insertion and extraction of tuples into/from the stores need to be concurrency-safe (e.g. using *locks*). This is represented in the concurrent algorithms by the symbol "$+_{safe}$" and "$-_{safe}$", representing concurrency-safe insertions and deletions respectively.
   For every local tuple (c', ip) associated to (c, op) by $Mapping(\cdot)$, an activation record (h, t, s) inserted into the *store* of unit of execution u, where $h = Handler(c', ip)$ is a handler within component c' that is to be activated, t is the tuple to be handled and $s = State(c')$ is the private state of c' and $u = Binding(c, op, c', ip)$ identifies the UE associated to the connector between the output port op of component c and port ip of component c'. *Post* returns as soon as the activation record is placed within a store $S[\cdot]$, i.e. the activation is handed over to the UE for asynchronous execution that does not dependent from this point on from the mediator.
2. **Handling of a stop marker tuple**
   A non-dispatchable message $stop$ is used to signal termination, and causes the dispatching loop of the UEs (discussed bellow) to terminate. The mediator propagates the stop marker by placing a $stop$ into every UE within the local address space.
3. **Tuples that are handled by components located in remote address spaces**
   In this case, a tuple needs to be exported to the remote space for processing. This is handled by a call to the *Send Proxy* that serves as the interface between machines distributed across address spaces.

Activation records placed in UE stores are dispatched concurrently by the UEs (Algorithm 3). UEs select nondeterministically an activation record (h, t, s) from their *stores* $S[.]$ and extract it. The nondeterminism reflects the lack of constraints on message ordering, which is important to account for transmission delays common in distributed systems. An UE activates a handler h of a component c specified in the activation record, passing it the tuple t and the state s as parameters. The process is repeated until there is a single marker tuple $stop$ in the store, which signals the end of processing. $Stop$ is never selected as part of Algorithm 3 and therefore eventually becomes the single content of the store[2]. This in turns causes the dispatching loop to exit, terminating execution of the unit.

Algorithm 3 Dispatching loop within a unit of Execution u

1: **repeat**
2: $(h, t, s) \leftarrow$ any activation record from $S[u]$
3: $S[u] \leftarrow S[u] -_{safe} (h, t, s)$
4: activate $h(s, t)$
5: **until** $S[u] = stop$

The *receive proxy* (Algorithm 4) is the element that handles messages that are exported from other machines into a local machine. The proxy employs an infinite reception loop (eventually terminated by other means as part of some machine finalization code). Within the loop, a blocking message reception instruction waits for messages and receives them when available.

Algorithm 4 Receive Proxy()

loop
block until a message m becomes available
$Post\_Local(m)$
end loop

Received messages are posted using a version of post that propagates tuples only to local handlers (Algorithm 5), to avoid that a message be repeatedly transmitted back and forth between address spaces. This undesirable situation might occur e.g. when $Mapping(c, op) = (c_1, ip_1), (c_2, ip_2)$, $n_1 = Placement(c_1), n_2 = Placement(c_2)$ and $n_1 \neq n_2$, i.e. a tuple is mapped to two components that are placed in distinct address spaces. Let t be originally generated by a component c_1 of n_1. The original posting (Algorithm 2) generates an activation record (h_1, t, s) handled locally and a call to $Send(c_2, n_2, t)$. The former causes a local activation of handler h_1 and the latter is sent through the send proxy to node n_2. Tuple t is eventually received by address space n_2, where it is processed by the receive proxy. Assume that instead of using $Post_{local}$ (Algorithm 5), the receive proxy called $Post$ (Algorithm 2). That would result in the symmetrical generation of an activation record (h_2, t, s) and a call to $Send(c_1, n_1, t))$. The same tuple t would then be bounced back to n_1 by the Send proxy, and an infinite loop would ensue. $Post_{local}$ omits the calls to the Send proxy, avoiding the undesirable situation.

[2] A more realistic scenario admits multiple $stop$ messages to be in the store.

Algorithm 5 $Post_{local}(m, c)$ - concurrent components

```
for all (c', ip) ∈ Mapping(c, op) do
  n ← Placement(c')
  if n = Local then
    if m ≠ stop then
      u ← Binding(c, op, c', ip)
      s ← State(c')
      h ← Handler(c', ip)
      S[u] ← S[u] +_safe (h, t, s)
    else
      for all ue ∈ U do
        S[ue] ← S[ue] +_safe stop
      end for
    end if
  else
    Do nothing
  end if
end for
```

3.4 Introspection facilities

To allow for the definition of advanced components, ThreadMill makes some meta-information available. Components can query the structure of a running configuration version and adapt their behavior according to the specifics of their context within this version:

- $WhoAmI$ returns the component identifier c of the caller.
- $InputPorts(c)$ returns the identifiers of the input ports of a component c.
- $OutputPorts(c)$ returns the identifiers of the output ports of a component c.
- $Producers(c, ip)$ returns the identifiers of the components whose output ports are connected to the input port ip of component c.
- $Consumers(c, op)$ returns the identifiers of the components whose input ports are connected to the output port op of component c.
- $ConnectedTo(c, p, c')$ returns the identifier of the (input or output) port of component c' that is connected to port p of component c.
- $Peers(c)$ returns the component identifiers of unfolded instances of c.
- $Rank(c)$ returns the relative unfolding index of a component c, i.e., it returns 1 for the first unfolded instance, 2 for the second and so on.
- $ThisSpace$ returns the identifier of the current address space.
- $Place(c)$ returns the identifier of the address space under which component c is placed.
- $Unit(c, op, c', ip)$ returns the identifier of the unit of execution under which tuples relayed via a connector are executed.
- $NumberUnits(as)$ returns the number of units of execution under address space as.

The above functionality is mainly used by synchronization operators that need to adapt their behavior according to their context of use. Regular application components rarely have the need to employ this functionality.

Notice that these introspection functions allow for a component to recursively navigate a whole configuration version graph, extracting all available information.

3.5 Infrastructure optimizations

Actual implementation faces specific challenges related to the need for high performance. In particular, two aspects impact the efficiency of an implementation: 1) the management of data,

given that structures are in many cases large and special care must be taken to avoid unnecessary copies and 2) the communication among components, that can be efficiently realized by taking advantage of shared memory within address spaces and by avoiding conversions whenever possible across address spaces. Sections 3.5 and 3.5 present some details of these two aspects respectively.

Data management The sizes of the structures processed by applications in the target domain are in many cases large - a single video frame can be as large as one megabyte. ThreadMill minimizes data moving and copying by caching single copies of data structures within each address space. Components that are co-located, i.e. deployed within same address space, access cached data via handles, rather than receiving a copy of it. Actual copies only take place when data has to be communicated to a different address space, and then only once per structure.

Tuples follow a *single assignment* (or write-once) policy. That means that once a structure is initialized, its contents ought not to be modified by components. In practical terms, that means simply that components treat data they receive as read-only. Results of processing are generated in a separate, new structure, or generated via copy-on-write. Single assignment simplifies data management because it eliminates the need for complex synchronization among distributed copies and simplifies distributed garbage collection. This is particularly relevant in distributed environments, where synchronization costs might become prohibitive because of communication latency among computational nodes.

Garbage collection is an essential service that must be performed by the infrastructure, given that components are by design unaware of how data they generate might be accessed by other components. ThreadMill performs this collection by keeping reference counts, aided by meta-information that identifies pointers within tuples.

Communication Communication among components that are co-located takes advantage of shared memory and is effected through synchronized queues.

Data flowing across address spaces is transparently processed and transmitted by ThreadMill, using a native format as the *wire* format (adopting the approach proposed by [8]). This reduces conversions in the common case in which address spaces are executing under similar platforms. Conversions among heterogeneous platforms are handled transparently by the recipient's infrastructure.

Tuples in ThreadMill might (and often do) embed references to other data structures (which might in turn embed other references). With the aid of the reflective meta-information, these embedded pointer structures are recursively visited by ThreadMill whenever a tuple needs to be transmitted across address space boundaries. The reflective information describes pointer locations and types within structures for each structure used by an application. A depth-first descent is performed through the pointers until leaf structures are reached. Leaf structures are contiguous byte regions that do not have embedded pointers or whose pointers point to structures that have already been visited. Leaf structures are transmitted to the intended address space, along with a minimal amount of meta-information necessary for the foreign address space to reconstruct the data structure on arrival. As the recursion unrolls, additional parts of a structure are sent piecewise to the destination address space.

As they are received, parts of a data structure are cached at the recipient's address space. Once the whole structure has been received, embedded pointers are patched to reflect the addresses of the cached parts in the recipient's address space.

Actual transmission is based on MPI [11]. ThreadMill's dependence on MPI is minimal though, and can be easily replaced by other existing communication libraries (e.g. PVM [29]), or developed directly on top of a TCP/IP (sockets) mechanism. MPI is convenient because of its ample availability and the availability of freely downloadable libraries. An added bonus is

that MPI-enabled applications can be easily made to run on Globus[15] computational grids. ThreadMill also currently makes use of the deployment facilities offered by MPI (applications are deployed through a call to mpirun).

4 Orchestration mechanism

The functionality described so far (Sections 2 and 3) allows for applications to be assembled from reusable components, and for these applications' concurrency to be defined in detail. Yet, these mechanisms, useful as they are, do not provide support for the synchronization requirements that surface in the context of processing of streamed data. In ThreadMill, these services are incorporated into *orchestration operators* defined on top of the generic model by means of the introspection extensibility mechanisms.

The concurrent and asynchronous processing of time-sequenced data introduces specific synchronization requirements. The objective of the synchronization mechanisms are twofold: on the one hand to maximize the potential concurrency by supporting the independent processing of sub-problems; on the other hand, the mechanisms must provide support for easy reassembly and re-synchronization of the results of these partial computations.

The re-synchronization needs furthermore to take into account the time-sequenced nature of the data. One wants to guarantee that independently of concurrency, results will be obtained in a strictly time-sequenced way, corresponding to the ordering of the input. Consider for instance a video processing application. The results must be produced for each frame in the strict order in which the frames were produced. This is a challenge due to the asynchronous and parallel nature of the applications, that may result in the intermixing of partial computations of multiple frames.

The synchronization requirements that emerge are non-trivial. Embedding of this functionality into application components would therefore make application components more complex and harder to maintain and reuse.

ThreadMill's solution is based on the isolation of recurrent synchronization patterns into specialized reusable components. The approach is therefore open-ended, since additional synchronization needs can similarly be isolated into reusable components.

This section starts with the presentation of the concept of *activity sets*, that identify tuples related to the same time-aligned tasks (Section 4.1). Section 4.2 introduces the synchronization operators provided by ThreadMill.

4.1 Activity sets

ThreadMill promotes a divide-and-conquer pattern of recursive decomposition of problems into subproblems that can be solved concurrently. This approach matches well the nature of applications in the target domain, many of which can be expressed in terms of a pipe-and-filter paradigm for which independent processing of parts is a natural match. Finding hands and face in a specific video frame, for instance, is broken down e.g. into preparing the image for further processing by filtering the skin-colored pixels; predicting the position of the object via kalman filters; generating hypotheses and evaluating them; and choosing the most likely hypothesis as representing the desired positions.

Communication among the different components is achieved indirectly by the production and consumption of tuples that can be seen as representing both the results of a previous stage and as commands that cause dependent stages to be activated. For each single video frame, for example, a large number of tuples is generated and made to flow among components. Take for instance the *Track* component (Figure 2): it generates hundreds (or thousands) of tuples that represent individual hypotheses about the current position of hands and face. These tuples

eventually cause *Evaluate* to be activated repeatedly, to produce likelihood estimates for each of the hypotheses from which the most likely is chosen.

ThreadMill provides a mechanism to identify the multiple tuples that are related to a single activity (e.g. a single frame in which hands and face are to be located) that we call *activity set*. Tuples of an activity set are identified by a unique activity identifier, called the *tick* (as in *clock tick*).

Tick timestamps allow components to distinguish which tuples belong to the same overall activity, e.g. to perform synchronization tasks and to guarantee that processing complies to a temporal order within synchronization operators. All tuples representing hypotheses related to a specific video frame, for instance, are tagged with the same *tick* as the frame itself; similarly, other tuples that refer to this frame, such as the skin-color filtered image and the masks used during evaluation will be tagged with identical ticks for each frame. Tick tagging allows for multiple parts of different activities (e.g the parts related to multiple frames) to flow concurrently through a graph, and at the same time be kept distinctive for synchronization purposes.

After generating all tuples that are related to an activity set, components produce an *end-of-tick* marker, that signals completion of the set associated with a specific *tick*. This marker helps components determine when producers they depend on are done generating tuples of a set, and is particularly useful in situations where producers can generate no tuples for a specific set, or a variable number of tuples per set.

The functionality that deals with ticks for synchronization purposes is isolated into reusable operators. Application components do not in general need to be concerned with the complexity of synchronization in general and tick processing in particular. We examine the synchronization operators of ThreadMill in the following section.

4.2 Synchronization operators

Synchronization operators allow for independently computed results to be recombined according to a variety of different strategies, resulting in indirect synchronization of tasks. These operators represent specialized components, placed within an open-ended hierarchy.The extension mechanism is based on standard inheritance of classes crafted according to ThreadMill guidelines.

Stream-oriented synchronization support is embedded within *channel* and *join* operators:

- **Channels** (represented by ellipsoids) effect time-ordered tuple distribution.
  The objectives of a channel are twofold: 1) to provide a mechanism that merges tuples that are generated by a variable number of producers attached to a channel's input port; 2) to serve each tuple to one (of a variable number of) consumers that is known to be ready to start processing.
  Tuples are served according to the temporal ordering of the *activity sets* they are related to, i.e., in *tick* order. This functionality handles the potential inter-mixing of different *activity sets* that results from concurrent and asynchronous processing, and guarantee that consumers have a temporally aligned view of these sets.
  A channel merges tuples of the same type it receives from potentially multiple producers and implements a *one-out-of-many* delivery policy to consumers that are available to perform a task. A protocol between channels and their consumers allows the latter to announce when they are ready to receive and process a tuple. If no consumer is free, tuples are cached until such time that a consumer becomes available.
- Cartesian product **joins** (represented by triangles marked with and "x") combine messages of potentially different types into single tuples.

Most commonly applications in the target domain comprise phases in which sub problems can be clearly identified. A natural solution in ThreadMill is for a component to generate multiple tuples, each representing a sub-problem, that can then be tackled in parallel by one or more instances of other components. Once the solutions have been computed, a common recombination pattern is to join them together into a single message that then suffers further processing, e.g. a video image filtered for skin-colored pixels and the multiple hypotheses that need to be evaluated against it are combined in preparation for evaluation. Join operators have two incoming ports that accept potentially different tuple types and one output port. All producers and all consumers of a join must be channels.

– A simpler type of join (represented as a triangle with a "v" marking) handles the situation in which tuples are alternatively produced by only one of two incoming connectors at each time. In this case the join operates as a pass-through, propagating to its output port whatever input it receives, i.e. any interleaving of tuples from either connector is accepted.

Figure 4 summarizes the graphical notation assigned to each component type. Notice that from the perspective of the underlying generic component model and meta-machine all these different operators correspond to components that are in principle not distinguished from each other. New operators can thus be similarly developed to enhance the architecture's capabilities without disturbing the underlying mechanisms.

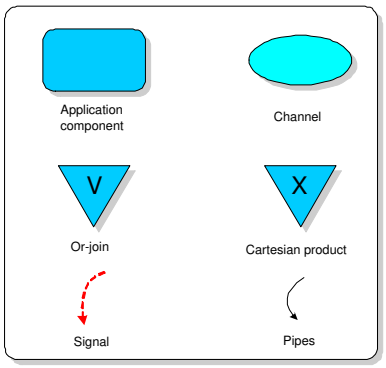

Fig. 4. Elements of ThreadMill's graphical notation.

4.3 Pipes and signals

Pipes and *signals* (represented in Figures 1 and 2 as solid and dotted arrows respectively) are both implemented in terms of *connectors*. Their distinction is based on the expectation with respect to the protocol (or lack thereof) implemented by tuples flowing through these connectors.

Pipes carry tuples that comply to the protocol that is related to the control of activity sets (Section 4.1). That is to say that tuples flowing through pipes are required to be timestamped with a tick, that the end of each activity set has to be marked by an *end-of-tick* marker.

Signals on the other hand are not required to implement the activity set protocol. In fact, signals will in general be used to implement specific inter-component protocols. Signals are useful e.g. to allow for the propagation of user interface related information, both for the purpose of notifying components of user actions and for collecting partial results for presentation to users.

Signals are also used to implement communication among unfolded instances of components, to allow these instances to coordinate their actions. One example of such coordination is the one performed by *channel* instances. Channel instances communicate with each other via signals to migrate tuples between overloaded and free instances so that the overall throughput is maximized. Details of this mechanism are out of the scope of this paper and are described in [6, chapter 3].

5 Experimental results

To experiment with the architecture, a multi-target tracker for hands and face of signers of American Sign Language was developed. This tracker is based on a Joint Likelihood Filter multi-target tracker described by Rasmussen [28]. Details of the implementation can be found in [6].

5.1 Code versions

A sequential version of the tracker was first developed from scratch, based on Rasmussen's [28] description. A ThreadMill version was then developed. The ThreadMill version employs essentially the same algorithm, and shares library code with the sequential version, to guarantee equivalence of processing. The control structure of the sequential and the ThreadMill versions differ considerably.

ThreadMill proved appropriated to describe the flow in this non-trivial application in a compact way. The resulting graph revealed opportunities for parallelism that were hidden in the original sequential code. One example is the potential execution of *Read* (to acquire new frames) and the *Skin filter* concurrently to *Track* and *Mask*. The latter two phases generate random samples and associated image masks based on priors, in preparation for the evaluation of the likelihoods against the evidence provided by the image. Since *Track* and *Mask* themselves do not depend on image data, they can be processed concurrently to the I/O operation performed by *Read* and the skin color filtering.

5.2 Conditions of the experiments

Experiments with multiple versions, ranging from one to ten nodes of a cluster of Compaq AlphaServer DS20 machines were run. In these experiments, a single (unchanged) application was configured to run on multiple distributed nodes. Each version was executed five times for different sample loads, and the execution times were averaged.

The sample loads varied from 1 to 3200 samples or hypothesis explored for each frame. The same movie clip (from Purdue's ASL moive database [23]) was used in all instances. Execution times for each run reflect the average processing time of two hundred frames.

5.3 Cost per phase

Since hypotheses-related processing accounts for the bulk of the computational cost for a non-trivial number of hypotheses, sample set sizes effectively represent growing computational demands that need to be faced by the concurrency mechanisms. Figure 5 shows the percentage of the processing time that is taken by each of the phases in JLF. *Selection*, *Reader*, *Track* and *Logger* took less than one percent of the overall processing time and are not displayed in the graph. As the number of samples grows *Mask* and *Evaluate* strongly dominate the processing requirements. It is thus the latter two phases that present the most promising opportunities for speedup through concurrent execution.

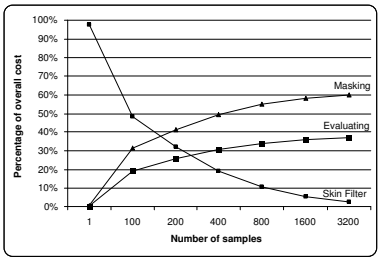

Fig. 5. Task influence per sample set size. Notice that as the sample sets grow, the data parallel tasks dominate the computation.

5.4 Cluster-based experiments

Figures 6-7 and Tables 1and 2 report results of running multiple versions of the JLF application distributed across multiple nodes. These versions are referred to as TM_i, where i indicates the number of address spaces employed by a version. Each address space corresponds to an individual dual-processor node. The experiment shows a close to 7-fold speedup for larger sample sets relative to the baseline sequential version when ten dual nodes (twenty processors) are used. Speedup is calculated as $Seq/TM_i, i = 1 \ldots 10$. A speedup of 2 means 100% faster than (twice as fast as) the sequential baseline. More importantly, the experiments show that given an adequate number of nodes, the time per sample can be kept almost constant even when the sample set size grows exponentially. It is expected that optimizations of the basic algorithm will therefore result in reductions that scale to larger sample set sizes.

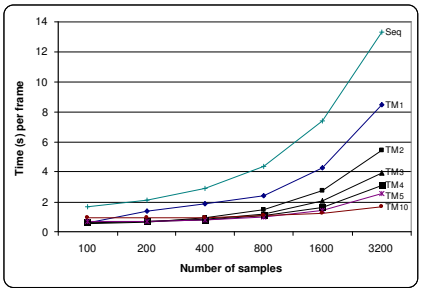

Fig. 6. Time spent per frame for each sample size on one to ten dual AlphaServer DS20 nodes.

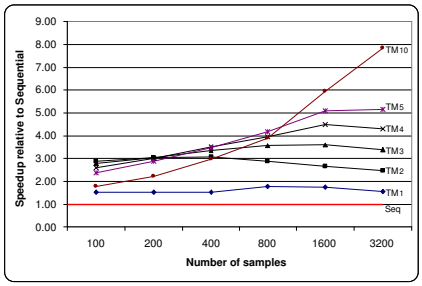

Fig. 7. Relative speedup with respect to sequential baseline.

5.5 Discussion

Since the parallel implementations TM_i execute at least as many steps as the sequential version Seq (the algorithms are identical), speedups are necessarily sub-linear. This follows from Amdahl's law equation [2], that determines maximum speedups for parallelized code as $speedup = \frac{1}{(1-f)+\frac{f}{s}}$, where f is the fraction of the code that is enhanced by parallelization and s is the speedup of the enhanced code. Even assuming an f of 100% and an ideal s equal to the number N of processors employed, the speedup would still be limited to N, i.e. strictly

Version	CPUs	Number of Samples					
		100	200	400	800	1600	3200
Seq	1	1714.40	2139.51	2899.52	4348.32	7424.91	13299.96
TM_1	2	1122.10	1402.65	1886.58	2445.23	4262.72	8500.25
TM_2	4	597.97	702.26	948.70	1510.17	2784.13	5411.85
TM_3	6	619.60	704.07	865.03	1215.73	2061.15	3936.12
TM_4	8	664.38	710.40	829.24	1092.54	1647.90	3078.39
TM_5	10	722.11	745.02	833.88	1038.99	1453.68	2572.87
TM_{10}	20	956.13	965.53	979.89	1110.63	1249.19	1694.56

Table 1. Time per frame (ms) on one to ten nodes of an AlphaServer DS20 cluster. Average of five runs per version/sample.

Version	CPUs	Number of Samples					
		100	200	400	800	1600	3200
TM_1	2	1.53	1.53	1.54	1.78	1.74	1.56
TM_2	4	2.87	3.05	3.06	2.88	2.67	2.46
TM_3	6	2.77	3.04	3.35	3.58	3.60	3.38
TM_4	8	2.58	3.01	3.50	3.98	4.51	4.32
TM_5	10	2.37	2.87	3.48	4.19	5.11	5.17
TM_{10}	20	2.79	2.22	2.96	3.92	5.94	7.85

Table 2. Speedup (Seq/TM_i) with respect to sequential baseline.

linear. Since the portion of the code that can in fact be enhanced by parallelism is less than 100%, results will necessarily be sub-linear. Table 3 presents the results of calculating the fraction f of the code that would correspond to a speedup s equal to the number of processors N. The runs with larger speedups display an enhanced fraction of 87% to 91% across versions, indicating that ThreadMill is taking good advantage of the concurrency that is possible in this application.

Version	CPUs	Number of Samples					
		100	200	400	800	1600	3200
TM_1	2	69.10%	68.88%	69.87%	87.53%	85.18%	72.18%
TM_2	4	86.83%	89.57%	89.71%	87.03%	83.34%	79.08%
TM_3	6	76.63%	80.51%	84.20%	86.45%	86.69%	84.49%
TM_4	8	70.00%	76.34%	81.60%	85.57%	88.92%	87.83%
TM_5	10	64.31%	72.42%	79.16%	84.56%	89.36%	89.62%
TM_{10}	20	46.56%	57.76%	69.69%	78.38%	87.55%	91.85%

Table 3. Fraction f of the program that corresponds to the observed $speedup$ (Table 1) on N processors for each version - $f = \frac{N(1-speedup)}{speedup(1-N)}$.

Super-linear results are possible only in cases where the parallel versions take advantage of concurrency to eliminate steps of computation. In the JLF case, this might come about e.g. if a smarter strategy for evaluation is employed, for instance by using the results of previous evaluations to focus the search for the best joint sample on more promising regions of an image.

As shown in Table 1, the configuration that achieves maximal speedup varies depending on sample set size, e.g. two nodes were enough to achieve the best times per sample for smaller

sample set sizes of 100 and 200 samples, and four nodes when the sample set size is 400 and so on. Runs with smaller sample sets are penalized by the relatively large communication costs, particularly for larger number of processors. This cost is compounded by low speed of the interconnections among the nodes used in the experiments (100Mb ethernet) that makes exchanges among address spaces become expensive. TM_{10} processing 100 samples, for example, distributes over the network the image to be analyzed to ten nodes, each of which processes just ten samples (one hundred samples divided into ten nodes) per frame. That results in a processing time per frame (0.956 s) that is more than fifty percent above TM_1's time (0.632 s) for the same number of samples (100).

Notice as well that the ThreadMill version TM_1 performs significantly better than the sequential version, even on a single (dual) node, i.e. when there is no distribution. This demonstrates the advantage of using intra-node threading, which allows for TM_1 to take advantage of the two processors available in each node to enhance performance. While Seq employs a thread just in connection to reading of new frames, TM_1 uses both processors to execute concurrently all the application's tasks.

6 Related work

6.1 Coarser-grained architectures

Architectures that target the same domain as ThreadMill (e.g. Neem [5], DACS [9], Galaxy Communicator [24], OAA [7]) focus on coarser-grained components. These architectures are typically not concerned with supporting concurrency via threads, or exploiting hardware shared memory to provide fast intra-address space communication. As a result, they might impose too high overheads if used to exploit the kind of fine-grained concurrency that is ThreadMill's focus. In fact, ThreadMill could be used to transparently implement the concurrency mechanisms within individual components of these coarser-grained architectures, complementing their functionality.

6.2 Fine-grained architectures

An exception to the above is Stampede [27]. Stampede shares ThreadMill's goal of facilitating the development of applications that deal with time-sequenced data. It supports distribution and concurrency via threads, and provides a synchronization distributed structure (the Space Time Memory - STM) that is similar in functionality to ThreadMill's channel operator. Both Stampede's STM and ThreadMill channels aim at regulating access to time-sequenced data in the presence of multiple concurrent producers and consumers, and both rely on timestamps embedded into tuples.

Stampede's STM allows random access to temporally indexed data it stores, and provides a rich set of retrieval operations that can specify a particular timestamp, the oldest/newest timestamp, or the newest unread item. ThreadMill, on the other hand, allows for multiple tuples to be associated with a single timestamp, which is an essential feature when support for fine-grained concurrency is desired e.g. to support a variable number of hypothesis to be generated and processed concurrently.

The staging of components is not addressed directly by Stampede (the mechanism is not described in the available literature), while ThreadMill incorporates this functionality seamlessly via *configuration versions*. The detailed control afforded by ThreadMill over concurrency and distribution extends to all components. That means that developers can determine how channels (and other components) should behave in terms of concurrency, providing an extra level of control that might result in enhanced performance in certain situations. Stampede, on the

other hand, defines STM as a privileged, system-level mechanism that obeys its own predefined concurrency and placement strategies, over which developers have little or no control.

A few dataflow architectures share ThreadMill's goals of providing support for fine-grained concurrency. Some of these architectures do not support distribution, only concurrency via threads (e.g. FSF [12], Weaves [16]). Others, such as RPV-II [3], support distribution, but not thread-based concurrency within an address space.

6.3 Timestamping

ThreadMill's timestamping mechanism is related to Jefferson's *timewarp* [17]. Timewarp is based on messages that are timestamped with a virtual time - real values totally ordered by the relation $<$. Processes keep an internal clock that is updated according to the timestamps of received messages. Each process optimistically processes messages in incremental timestamp order. Since messages can arrive in any order, processes might be forced to roll-back some of the processing they performed. This happens whenever a message with a previous ("older") timestamp arrives after one or more messages with subsequent ("newer") timestamps have already been processed. Timewarp offers a mechanism of propagation of "negative messages" that is guaranteed to bring a system to a consistent state from which processing can resume. In contrast to Timewarp, ThreadMill does not require rollbacks. Instead, ThreadMill's channels introduce delays as appropriate to guarantee that consumer operators always have a consistent time-sequenced view of the computation. The essential difference that allows ThreadMill to avoid rollbacks is that the *tick* protocol (discussed in Section 4.1) provides means - basically through the propagation of *end-of-tick* markers - for operators to determine when all tuples of a certain tick have been seen, so that they can move their internal clocks forward without the risk of older tuples ever being received.

6.4 Vision-oriented architectures

A certain number of architectures explores a narrower domain concerned with the support of vision-based applications rather than the more general human communication that is ThreadMill's focus. On the one hand, these architectures may provide better support for the specific tasks that they target; on the other hand, they are restricted in many cases to executing these same specific tasks and cannot in general be employed e.g. to support the development of a broader range of applications. The Argus architecture [21], for instance, targets applications related to gesture-based control of domestic appliances using multiple stereo cameras; the Animate Agent Architecture [10] and Perseus [18] offer vision-based support and planner integration for the purpose of controlling robots; the Blob streaming Framework [26] focus medical imaging applications.

6.5 Actors and message-driven approaches

The asynchronous messaging meta-machine that is the foundation of ThreadMill's communication and concurrency semantics is related to approaches such as *Actors* [1], *message-driven execution* [20] and *processor virtualization* [19]. While these approaches are associated with programming languages and programming language constructs, ThreadMill employs similar mechanisms to define (and implement) its underlying operational semantics. While it is possible to build systems based solely on the asynchronous messaging provided at this level, this solution is less than convenient. This is particularly true considering the elaborate synchronization of stream-oriented applications that require time-sequenced recombinations of partial results, as is the case in the domain targeted by ThreadMill.

6.6 Library-based approaches

ThreadMill's approach can also be contrasted to library-based approaches that are implemented strictly as an API (such as MPI [11] or PVM [29]). The isolation of concerns promoted by ThreadMill's orchestration language makes it easier to reuse application code, given that component interconnection and staging are not embedded in application code, as is required by approaches that require that all services be activated via API calls. Library-based approaches require programmers to face themselves the complexities of structuring their code according to complex usage conventions.

6.7 Pattern-based parallel code generation

ThreadMill's configuration graphs can be seen as user-defined patterns that describe solutions to specific problems (e.g. a tracking problem using a JLF approach). Configuration versions instantiate these patterns by specifying staging details. The ThreadMill compiler generates code that binds these instantiated patterns into the framework provided by the underlying execution mechanism (the ThreadMill meta-machine). A somewhat similar approach is employed by the University of Alberta's CO_2P_3S system [22]. CO_2P_3S is a system that automates the generation of frameworks based on parameterized parallel patterns. Application programmers choose pattern(s) that match the problem they are trying to solve (e.g. a *mesh pattern*), customize the pattern by providing parameters, and code functions that are called by the generated frameworks' *hooks*.

In contrast to CO_2P_3S, in ThreadMill developers design application specific patterns, expressed in terms of a configuration graph built from fine-grained reusable operators. ThreadMill targets stream-oriented rather than parallel applications in general, and is thus able to take advantage of assumptions that are true in this specialized domain to offer more focused support to facilitate the creation of application-specific patterns and performance tuning of these patterns.

Extensibility in CO_2P_3S is coarse-grained - one may develop new patterns, but this requires intensive programming of potentially large amounts of code. In ThreadMill, extensibility is achieved through the development of reusable operators, that may be orders of magnitude simpler to develop than overall patterns.

ThreadMill is furthermore distinguished by the support it provides to staging as a separate pattern instantiation phase. CO_2P_3S apparently supports concurrency within an address space, but not distribution. Performance fine-tuning in CO_2P_3S is achieved by rewriting the generated code that is exposed to programmers at the Intermediate and Native Code Layers respectively.

7 Conclusions and Future Work

This paper presented ThreadMill, an architecture targeting the development of applications in domains where high volumes of streamed data need to be efficiently analyzed, e.g. those that target the analysis of human communicative behavior for instance in speech and gesture recognition. ThreadMill focus on efficient handling of the high volumes of streamed data that characterize applications in this domain. It allows for applications to be optimally deployed on a variety of different execution environments without the need for code changes. The basic notion that affords that is the separation of communication, concurrency and synchronization concerns promoted by ThreadMill.

A few research directions suggest themselves as possible next steps in the development of Threadmill:

– **Dynamic reconfiguration behavior** - The underlying machinery, and consequently the implemented infrastructure that is based on it are reflective. Computation is therefore essentially driven by the contents of tables, that are consulted by the meta-machine, and based on which communication, distribution and concurrency are effected. A natural research direction towards which ThreadMill could be evolved is concerned with adding dynamic reconfiguration behavior.
  Modifying the behavior of a *configuration version* can be in most cases straightforwardly achieved by applying transformations to meta-tables, perhaps in a synchronized fashion when multiple processes are to be affected. One can foresee architectural support being offered e.g. for dynamic attachment and detachment of connectors, for load-sensitive creation and retraction of *units of execution* and for component level migration. Of particular relevance to the implementation of the latter is the decoupling of *units of execution* from components promoted by the meta-machine. Since the *loci* of concurrency are orthogonal to components, component migration is equated to synchronized meta-table patching, and migration of the state s associate to a migrated instance. State migration is equated to message transfer, and is already directly supported by the basic functionality of ThreadMill.
  Interesting related aspects have to do with support for availability, load-balancing, quality of service guarantees, as addressed e.g. by the Neptune Project [25].
– **Identification of a larger set of synchronization patterns** - Of particular interest is how well ThreadMill's paradigm can be extended via this mechanism, and whether this would permit for a larger range of applications to be benefited by the technique. Given component extensibility, discovered patterns could be readily incorporated for reuse. Particularly relevant are patterns concerned with finer-grained data-parallelism, that support concurrent processing of sub-images. While the basic mechanisms to support this kind of concurrency is available in ThreadMill, it is clearly desirable to have reusable components that can be used to handle the distribution and reassembly of sub-images.
– **The automation of performance fine-tuning** - Since performance fine-tuning can be effected largely without introducing changes to application code (other than algorithmic improvements), an iterative style of performance fine-tuning can be pursued. One can envision the optimization of ThreadMill *configurations* as taking place in an environment in which profiling and instrumentation information is iteratively used to reconfigure a ThreadMill application on-the-fly, initiating a new cycle of measurements and further refinements by an optimizer in a cyclic fashion. Such an iterative style that mixes experimentation with the use of an optimizer is used, e.g. by the FFTW project to optimize a Fast Fourier Transform package to different problem characteristics on different execution environments [13].
– **Support for a more flexible state sharing mechanism** - While the single assignment policy associated to tuples seems to match well the nature of the applications in the domain, it is fair to assume that a more flexible mechanism to support sharing of state might be beneficial for some kinds of applications.
  Particularly for unfolded instances, transparent support for sharing of a common state might provide for facilitation of development of components that make use of a single data structure, without the need to require that the component itself implement the potentially complex protocols required to keep instances synchronized.

References

1. Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Cambridge Press, 1986.
2. G.M Amdahl. Validity of single-processor approach to achieving large-scale computing capability. In *Proceedings of AFIPS Conference*, pages 483–485, Reston, VA, 1967.

3. Daisaku Arita and Rin-ichiro Taniguchi. RPV-II: A stream-based real-time parallel vision system and its application to real-time volume reconstruction. In B. Schiele and G. Sagerer, editors, *Proceedings of the Second International Workshop on Computer Vision Systems (ICVS)*, volume 2095 of *Lecture Notes in Computer Science*, pages 174–189, Vancouver, Canada, 2001. Springer-Verlag.
4. P. Barthelmess and C.A. Ellis. A distributed and parallel component architecture for stream-oriented applications. Technical report, University of Colorado at Boulder, 2004.
5. P. Barthelmess and C.A. Ellis. The Neem Platform:An evolvable framework for perceptual collaborative applications. *Journal of Intelligent Information Systems*, 2004. Forthcoming.
6. Paulo Barthelmess. *ThreadMill: A highly configurable architecture for human communication analysis applications*. PhD thesis, Computer Science Department, University of Colorado at Boulder, November 2003.
7. Adam Cheyer and David Martin. The Open Agent Architecture. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(1/2):143–148, March 2001.
8. Greg Eisenhauer, Fabian E. Bustamante, and Karsten Schwan. Native data representation: An efficient wire format for high-performance distributed computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(12):1234–1246, 2002.
9. G. Fink, N. Jungclaus, F. Kummert, H. Ritter, and G. Sagerer. A distributed system for integrated speech and image understanding. In *International Symposium on Artificial Intelligence*, pages 117–126, Cancun, Mexico, 1996.
10. R. James Firby, Roger E. Kahn, Peter N. Prokopopowitz, and Michael J. Swain. An architecture for vision and action. In Chris Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 72–79, San Francisco, 1995. Morgan Kaufmann.
11. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1994.
12. Alexandre R.J. Fran cois and Gérard G. Medioni. A modular software architecture for real-time video processing. In *Proceedings of the International Workshop on Computer Vision Systems*, pages 35–49, Vancouver, B.C., Canada, July 2001.
13. M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the ICASSP*, volume 3, pages 1381–1384, 1998.
14. Erich Gamma, Richard Halm, Ralph E. Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley, 1995.
15. Globus Project Team. Globus project. http://www.globus.org.
16. Michael M. Gorlick and Rami R. Razouk. Using Weaves for software construction and analysis. In Les Belady, David Barstow, and Koji Torii, editors, *Proceedings of the 13th International Conference on Software Engineering*, pages 23–34, Austin, Texas, May 1991. IEEE Computer Society Press.
17. David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.
18. Roger A. Kahn. *Perseus: An Extensible Vision System for Human-Machine Interaction*. PhD thesis, University of Chicago, August 1996.
19. L. V. Kale. The virtualization approach to parallel programming: Runtime optimizations and the state of the art. In *Los Alamos Computer Science Institute Symposium - LACSI 2002*, Albuquerque, 2002. "State of the field" paper.
20. L.V. Kale and A. Gursoy. Performance benefits of message driven executions. In *Intel Supercomputer User's Group*, St. Louis, MO, October 1993.
21. Markus Kohler, Sven Schröter, and Heinrich Müller. The ARGUS-architecture for global computer-vision-based interaction and its application in domestic environments. In *Proc. Human Computer Interaction 1999 (HCI'99)*, pages 296–300, Munich, Germany, August 1999.
22. S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1683, December 2002.
23. Aleix M. Martnez, Ronnie B. Wilbur, Robin Shay, and Avi C. Kak. Purdue RVL-SLLL ASL database for automatic recognition of American Sign Language. In *Fourth IEEE International Conference on Multimodal Interfaces*, page 167, Pittsburgh, Pennsylvania, 2002.
24. Mitre Corporation. *Galaxy Communicator Documentation*, 2002.
25. Neptune Project. http://www.cs.ucsb.edu/projects/neptune/.
26. Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt. Design and performance of an object-oriented framework for high-speed electronic medical imaging. *Computing Systems*, 9(4):331–375, 1996.

27. Umakishore Ramachandran, Rishiyur Nikhil, James Matthew Rehg, Yavor Angelov, Arnab Paul, Sameer Adhikari, Kenneth Mackenzie, Nissim Harel, and Kathleen Knobe. Stampede: A cluster programming middleware for interactive stream-oriented applications. *IEEE Transactions on Parallel and Distributed Systems*, pages 1140–1154, November 2003.
28. C. Rasmussen. *Integrating Multiple Visual Cues for Robust Tracking*. PhD thesis, Yale University, 2000.
29. V. S. Sunderam, A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system. *Parallel Computing*, 20:531–545, March 1994.