A Generic Software-Framework For Distributed, High-Performance Processing of Multi-View Video

Dirk Farin[a] and Peter H.N. de With[a,b]

[a]Eindhoven University of Technology, PO Box 513, 5600 MB, The Netherlands
[b]LogicaCMG, PO Box 7089, 5605 JB Eindhoven, The Netherlands
d.s.farin@tue.nl

ABSTRACT

This paper presents a software framework providing a platform for parallel and distributed processing of video data on a cluster of SMP computers. Existing video-processing algorithms can be easily integrated into the framework by considering them as atomic processing tiles (PTs). PTs can be connected to form processing graphs that model the data flow of a specific application. This graph also defines the data dependencies that determine which tasks can be computed in parallel. Scheduling of the tasks in this graph is carried out automatically using a pool-of-tasks scheme. The data format that can be processed by the framework is not restricted to image data, such that also intermediate data, like detected feature points or object positions, can be transferred between PTs. Furthermore, the processing can optionally be carried out efficiently on special-purpose processors with separate memory, since the framework minimizes the transfer of data. Finally, we describe an example application for a multi-camera view-interpolation system that we successfully implemented on the proposed framework.

Keywords: software frameworks, middleware, parallel algorithms, 3D-video.

1. INTRODUCTION

Video processing and analysis systems pose high demands on the computation platform in terms of processing speed, memory bandwidth, and storage capacity. Requirements grow even further if real-time processing speed is needed for analysis and visualization applications, and if multi-view video has to be processed. On current commodity hardware, this processing power is not available. For example, simultaneous capturing of two 640×480@25fps videos from IEEE-1394 cameras already fills the full bus bandwidth. The straightforward solution for this problem is to use computation clusters instead of expensive specialized hardware. However, the design of distributed systems is a troublesome task, prone to design flaws.

Most previous approaches to parallel computation have concentrated on fine-granular data-parallelism. In this approach, the algorithms themselves are parallelized, which requires a reimplementation of the algorithms. This is difficult, especially because computer-vision engineers are rarely experts in distributed processing.[1] Since complex systems are composed of many processing steps, parallelization can also be carried out by keeping the (sequential) algorithms as atomic processing steps. This has two advantages: algorithms are easier to implement and algorithm development stays independent of the parallelization. Note that it is at the same time still possible to use the distributed-processing framework to also parallelize the algorithms itself by splitting the task into smaller sub-tasks that can be computed independently.

In this paper, we propose a generic framework for distributed real-time processing, into which existing software components (algorithms) can be integrated effortlessly. Algorithm parallelization is achieved by splitting the processing into a set of processing tiles (PT). Each processing tile performs a specific operation on a set of inputs and generates one or several outputs. The inputs and outputs of the PTs can be connected to build arbitrary processing graphs, describing the order and dependencies of processing. The framework software takes care about the appropriate distribution of the algorithms over the processing resources and the control of execution.

The proposed framework provides the following features.

1. **The processing graph is not limited to a certain topology** (like processing pipelines). In particular, PT outputs can be efficiently connected to the inputs of several PTs. As an example, consider a multi-view system, in which depth estimation has to be carried out between pairs of cameras, feeding the same input data (captured images) into several algorithms (depth estimators for different pairs of cameras).

2. **Automatic parallelization.** While the processing within one PT is considered an atomic operation, parallelism is obtained by running PTs in parallel. The framework also allows to process data from different time instances in parallel, such that not only horizontal parallelism (concurrency of independent tasks), but also vertical parallelism (pipelining) is exploited.

3. **The framework is network transparent.** The processing graph can be distributed over a cluster of computers and still be accessed in a uniform way. If data has to be sent across the network, this is done transparently by the framework.

4. **The framework supports operations on arbitrary data-types.** Hence, not only image data can be processed, but also structured data-types like mesh-based geometry. An important alternative view onto this is that data can be processed in different representations. For example, low-level processing tasks (lens-distortion correction, image-rectification, depth estimation) can be implemented more efficiently on the graphics processor (GPU). In this case, the image data is loaded into the texture memory of the graphics card. Since the overhead of transferring the image data between main memory and the graphics card would annihilate the performance gain of processing on the GPU, it should be avoided whenever possible. This is achieved by passing only handles to the texture data between PTs and doing the conversion to image data in main memory only when necessary.

In this paper, we will first describe the main considerations taken into account when designing the framework software (see Section 2), and we give an overview of the framework architecture. In Section 3, we describe the implementation in more detail. An example multi-view application and its implementation using the proposed framework is presented in Section 4. Finally, Section 5 briefly describes some advanced features of the framework that further increase its flexibility.

2. DESIGN OF THE DISTRIBUTED PROCESSING FRAMEWORK

2.1. Design Considerations

The design of our *Distributed Processing Framework* (DPF) was driven by the following requirements.

2.1.1. Fine-grained vs. coarse-grained parallelization

Previous work on parallel algorithms has mostly considered parallelization using a fine-granular data-parallelism within a single algorithm. The difficulty with this approach is that it complicates the implementation of the algorithm, because it requires knowledge of the image-processing algorithms as well as knowledge about parallel and distributed programming. On the other hand, large applications consist of a multitude of processing steps. This allows to apply parallel processing on a coarser level, at which every algorithm is (conceptually) implemented in a sequential program (in practice, this does not have to be followed strictly, as we will describe in Section 5). Since we are targeting complex video-processing systems comprising several algorithms, we have chosen the coarse-grained parallelization because it simplifies the implementation of each algorithm. Furthermore, the coarse-grained parallelization has a lower communication overhead, which is usually one of the most restricting bottlenecks.

2.1.2. SMP and cluster parallelization

Currently, multi-core processors begin to replace single-core processors because they allow to increase the computation speed more economically than by increasing the processor frequency. While current processors have two or four cores, the number of cores is expected to increase further in the future. But even with multi-core processors, the processing speed is limited because of the limited memory-bandwidth, limited I/O bandwidth, or simply because the computation speed of these processors is still not sufficient for the application. To overcome these limitations, processing in a cluster of computers is a viable approach.[2]

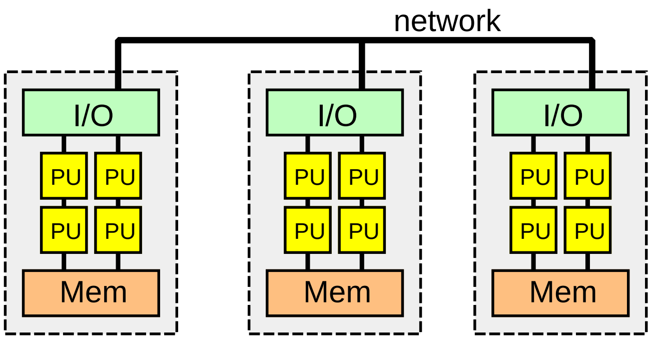


Figure 1. Three multi-core computers (with several processing units 'PU') that are connected over a network.

For these reasons, the DPF should enable parallel processing in both ways, by exploiting multiple processing cores within one shared-memory system (SMP), and by distributing the work over a cluster of computers (Fig. 1).

2.1.3. Automatic parallelization vs. manual splitting of tasks

In designing a system using our DPF, it must be decided which processing tiles should be processed on which computer. The optimal distribution of the processing network over the computers could be determined automatically, but we still prefer to specify the assignment of the processing tiles by hand for the following reasons. First, the number of processing tiles is rather limited for most systems, and the placement of some of these tiles is dictated by the hardware (e.g., the camera capturing must take place at the computer to which the cameras are connected). Second, for an automatic optimal distribution of the tasks, exact knowledge about the processing times, the required bandwidths, or the available computation resources is required. These are often difficult to specify formally, particularly in a heterogeneous architecture with special-purpose processors.

2.1.4. Flexibility

Our intention in designing the DPF was to provide a general framework for a wide area of applications. Hence, the design should not impose a specific processing architecture, like a fixed processing pipeline, which might be unsuitable for many applications. Furthermore, the framework should strengthen the reuse of processing tiles for different applications. This is supported by allowing processing tiles to have a flexible number of inputs and outputs, which do not have to be connected all at the same time. Unconnected outputs can indicate to the processing tile to disable part of its processing, and optional inputs can be used for optional hints that may help the algorithm.

2.1.5. Special-purpose processor utilization

Many low-level image-processing tasks are well suited for parallel processing, but this parallelism cannot be achieved with standard general-purpose processors. Even though these processors nowadays have support for SIMD instructions, especially designed for multimedia applications, (like MMX on x86, or Altivec on PowerPC architectures), the degree of parallelism is limited and restricted to simple processing.

Additional to the CPU, specific media-processors do exist which offer higher parallelization factors than regular CPUs. Our framework supports the use of these specialized processors if they are integrated in the system as co-processors. As an example, the *graphics processing unit* (GPU) has evolved into a programmable processor that can also be used independently of the CPU. Even though the GPU was originally designed for computer-generated 3D graphics, its programmability has increased the area of application to a wide range of general processing tasks. Other than the CPU, the GPU is especially efficient at heavy parallel processing on simple data-structures, like the pixels in 2-D images. Hence, for pixel-based low-level processing, the GPU is generally more efficient than the CPU. These tasks include operations like geometric correction of radial lens distortion, geometric rectification of stereo image-pairs for depth estimation, color-space transformations, or feature detectors like corner or edge detectors. Also more complex algorithms like depth estimation can be

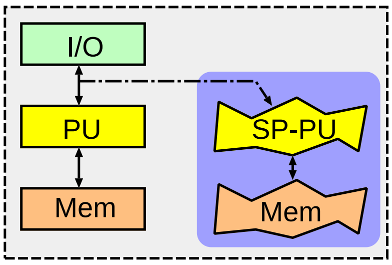


Figure 2. A computer with a special-purpose processing unit (SP-PU) that has its own local memory.

carried out more efficiently using an algorithm combining the fast pixel-based processing on the GPU with the processing of complex data-structures on the CPU.[3]

When utilizing the GPU, it should be considered that the image data to be processed has to be stored in the graphics-card memory (Fig. 2). The time to transfer the image between different memories is not negligible, and can even exceed the actual computation time. For this reason, it is important to avoid unnecessary memory transfers wherever possible. This has to be considered in the design of the DPF by providing several options how the image data is transferred between successive processing tiles.

2.2. Overview of the Distributed Processing Framework

The core of our distributed processing framework (DPF) is a set of user-definable processing tiles (PT). A processing tile conducts an operation on its input data and generates new output. The number of inputs and outputs is flexible. In order to define the data-flow through the PTs, they can be connected to form a processing graph of an arbitrary topology (without cycles).

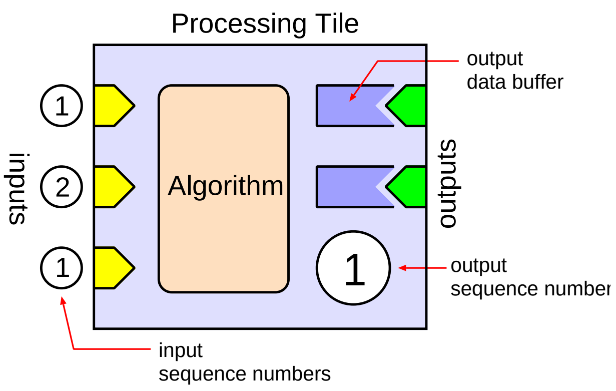
A *Processing Graph Control* (PGC) distributes the tasks, that should to carry out in the graph of PTs, over a set of worker threads, hereby exploiting multi-processor parallelism in an SMP system. The scheduling is determined by splitting the processing into a set of "(PT, sequence-number)" pairs. Each of these pairs represents a processing task that can be issued to a thread. The scheduler maintains the set of tasks in three queues: the tasks that are currently processed, tasks that are ready to run (all input dependencies are met), and tasks that cannot be started yet because of unmet dependencies. These queues are updated whenever a PT has finished its computation.

For specifying processing graphs that are distributed over several computers in a cluster, the PGC can be wrapped into a *Distributed Processing Graph* (DPG). The DPG provides a uniform access to all PTs even though they might be distributed over different computers. Whenever data is to be transferred between computers, a network connection is established transparently to the user. From a user point-of-view, a DPG looks like a local processing graph, because most of the network distribution is hidden from the user.

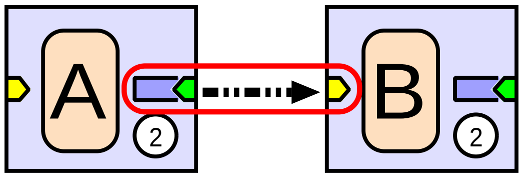
It should be emphasized here that the DPF is organized in three separate layers that represent subsets of the features made available by the DPF. The idea is to provide simpler APIs to the programmer when the full-featured framework is not required. The typical usage of these three layers is briefly explained below.

- A system building upon the first layer includes only the PT objects, which are connected to a graph. There is no central scheduler organizing the data processing. Data can be processed directly by pushing new data into some PT input. This will trigger the processing of this tile and also the successive tiles, where additional input is requested as required. On the other hand, the graph of PTs can also be used in a pull mode, where new output data is requested at some PT, which again triggers the processing of the tile. If there is missing input, the PT first acquires the required inputs from the preceding tiles.

- The second layer adds a scheduler (PGC) to the graph of connected tiles. This scheduler manages multiple worker threads to carry out the computations in parallel.



(a) Processing tile with three inputs and two outputs.



(b) The input is taken directly from the output buffer of the connected PT in order to avoid redundant memory copies.

Figure 3. A Processing tile (PT) has connectors for input data and output data. The data for each output is stored in a output buffer. The successive PT is accessing that data directly without copying. Processing of the input is only allowed if the sequence number of all inputs are equal. After processing, the *sequence number* of the tile's outputs is updated.

- The third layer adds a network-transparent distribution of the processing graph (DPG) over a cluster of computers. To this end, a server application is run on every computer in the network, and the servers are registered at a central control computer. PTs can be instantiated at any arbitrary computer through a uniform API at the control computer. Connecting two PTs across the network is possible and handled transparently to the user.

3. IMPLEMENTATION DETAILS

3.1. Processing Tiles

All algorithms are wrapped into *Processing Tile* (PT) objects. Each PT can accept a number of inputs and can also create several outputs. For the DPF, a PT appears like an atomic operation (however, the algorithm in the PT may itself be implemented as a parallel algorithm, independent to the parallelization performed in the DPF). Each PT provides the memory for storing the computed results, but it does not include buffers for holding its own input data. The input data is accessed directly from the output buffers of the connected tiles. This prevents that data is unnecessarily copied between PTs, which could constitute a considerable part of the computation time. On the other hand, this blocks the PT that provides the input from already starting the work on the next Data-Unit. If this is a problem, an additional buffering PT can be inserted in-between the two PTs to decouple the data-dependency between these two PTs.

Every output can be connected to several inputs, without any extra cost. Moreover, inputs and outputs can also be left unconnected. While the algorithm within the PT might simply proceed as usual even when there are unconnected outputs, it can be more efficient by disabling generating the data for this output. This feature can be used, for example, to create outputs that provide a visualization of the algorithm. When the visualization is not required, the output can be left unconnected. Connecting only some of the inputs can be used, for example, for operations that work with a varying number of inputs (like composition of images, depth estimation from multiple cameras), or to support additional *hints* for the algorithm (segmentation masks that can help to increase the quality of the result, but which are not required).

Each PT also saves a *sequence-number*, indicating which frame was processed in the last step, now being available at the outputs. Using this sequence-number, a PT can check if all its inputs are available such that it can start processing.

The interface to a PT comprises the following main methods.

- **Process()**. Run the algorithm encapsulated in this PT and generate the output data. Unconnected outputs may be skipped.
- **IsBlocked()**. Indicates if the PT cannot process more data at this moment for internal reasons. Such internal reasons could be that required processing resources are currently not available, or that buffers are completely filled.
- **Connect(int output, PT& pt, int pt_input)**. Connect an output to the input '`pt_input`' of PT '`pt`'.
- **SetParameter(string name, string value)**. Set some algorithm parameters. These are coded as strings to provide a uniform interface even when parameters have to be sent to PTs located on other computers in the network.
- **StartCommand(string name)**. Perform some PT-class specific action like starting or stopping capturing from cameras.

3.2. Data-Units

Data that should be passed between PTs is encapsulated in *Data-Unit* objects. Each Data-Unit provides a uniform interface for communicating with the DPF, but it can nevertheless hold arbitrary types of data. The Data-Unit must provide a function to serialize the data and to reconstruct the Data-Unit again from the serialized data. This is used by the DPF in order to send data across the network, transparently for the user.

As an example for non-streamable data, let us consider the example of processing image data in the GPU again. As stated above, transfering data between main memory and graphics-card memory is relatively time-consuming. Hence, this memory transfer should be avoided between successive steps that are processed on the GPU. In this case, the Data-Unit does not hold the actual data, but it merely saves the handle to the memory area in the GPU, in which the image data is stored. Since this data cannot be serialized without being transferred to main memory, this data is considered to be non-streamable. Because non-streamable data is a special representation of a corresponding ordinary, streamable data representation, we also require that special PTs are available for converting between the streamable and the non-streamable representations. In our example of GPU processing, these would be PTs that transfer the data to the graphics-card memory, and *vice-versa*.

There is another special Data-Unit type, which is generated to indicate the *end-of-data* condition. Whenever a PT receives an end-of-data Data-Unit, it also puts end-of-data Data-Units into its outputs.

3.3. Using a Graph of PTs Without Processing Graph Control

As noted above, it is possible to use a graph of connected PTs without any further central control. For simple pipelined processing, this comes close to the *Decorator* design-pattern[4] used in software engineering. However, processing in our graph of PTs is more flexible than a straight processing pipeline. We do not only allow arbitrary (acyclic) graphs of PTs, but also allow a push-data semantic as well as a pull-data semantic. Using a graph of PTs without a central Processing Graph Control (see next section) is simple to use, but note that parallel processing is not available.

Processing of a new unit of data is triggered at an arbitrary PT in the graph. If new input is fed into the graph, then triggering happens at the moment when the new data is passed into one PT. Whenever a PT is triggered, it checks if the data at its inputs are already available. This is visible from the sequence-numbers in the predecessor PTs. If some input is missing, this predecessor PT is triggered. When all input is available, the tile performs its operation and triggers all output PTs if they are not yet at the same sequence-number (this can happen if triggering one output PT has propagated to another output).

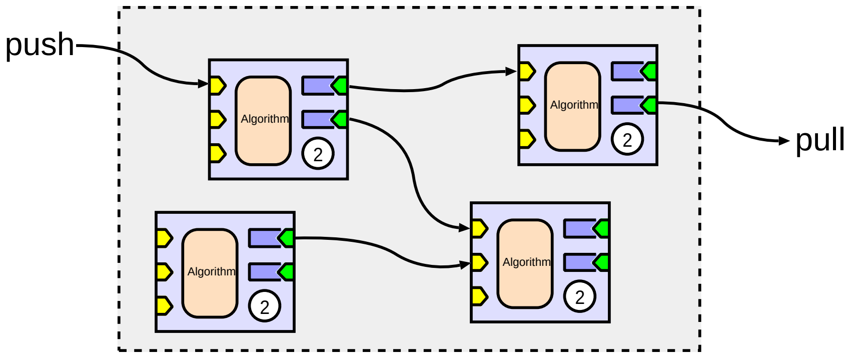
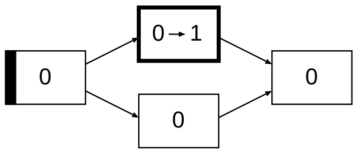
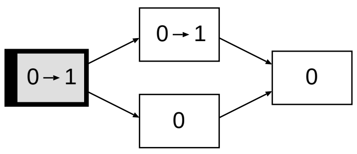


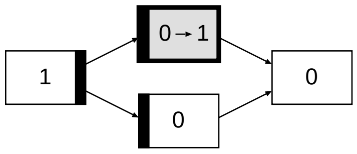
Figure 4. Connecting several Processing Tiles together as a processing graph. Processing can be carried out by either *push*ing new data into the graph, or by *pull*ing the next data unit out of the graph.



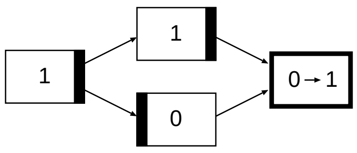
(a) Step 1.



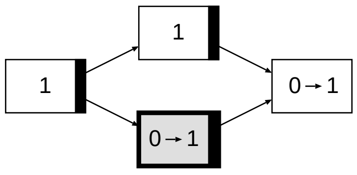
(b) Step 2.



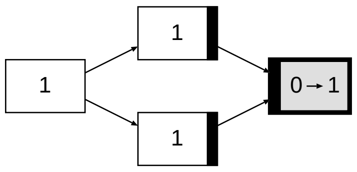
(c) Step 3.



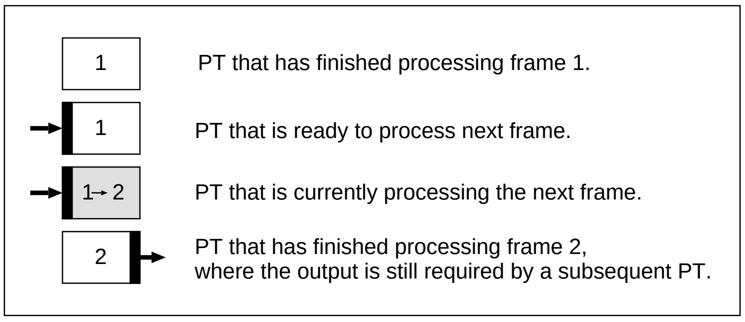
(d) Step 4.



(e) Step 5.



(f) Step 6.


1
PT that has finished processing frame 1.
1
PT that is ready to process next frame.
1→2
PT that is currently processing the next frame.
2
PT that has finished processing frame 2,
where the output is still required by a subsequent PT.

(g) Symbols used in the diagrams.

Figure 5. Example processing graph that is being processed without a central scheduler.

3.4. Processing Graph Control

The *Processing Graph Contol* (PGC) comprises a scheduler that distributes the processing tasks in a graph of PTs over several processors running in parallel. To this end, a *pool-of-tasks* scheme is applied. The PGC maintains a set of PTs that are ready to process the next Data-Unit. A PT is ready if all the input data is available at the connected inputs, if the PT is not internally blocked, and if the outputs of the PT are not required by any other PT anymore. The PGC maintains three sets of "(PT, sequence-number)" pairs to schedule the tasks to the threads. The **to-be-processed** set is initially filled with all tasks for the first sequence-number. Whenever the first task with the latest sequence-number has started, all tasks for the next sequence-number are added to this set. After a PT has finished its processing, the successor and predecessor PTs are examined if they are now ready to be processed. If they are, the corresponding task is moved from the to-be-processed set into the **ready-to-run** set. Whenever a working thread has finished a task, it gets a new task from the ready-to-run set, moves this task into the **in-progress** set and starts processing.

In order to efficiently test if a task can be processed, an **active-edges** set is maintained. An active edge is an edge in the PT graph, connecting an already-processed output of a tile PT_o with a not-yet processed tile PT_i. This active edge represents a data-dependency which prevents that PT_o can process another unit before PT_i has finished using this output data.

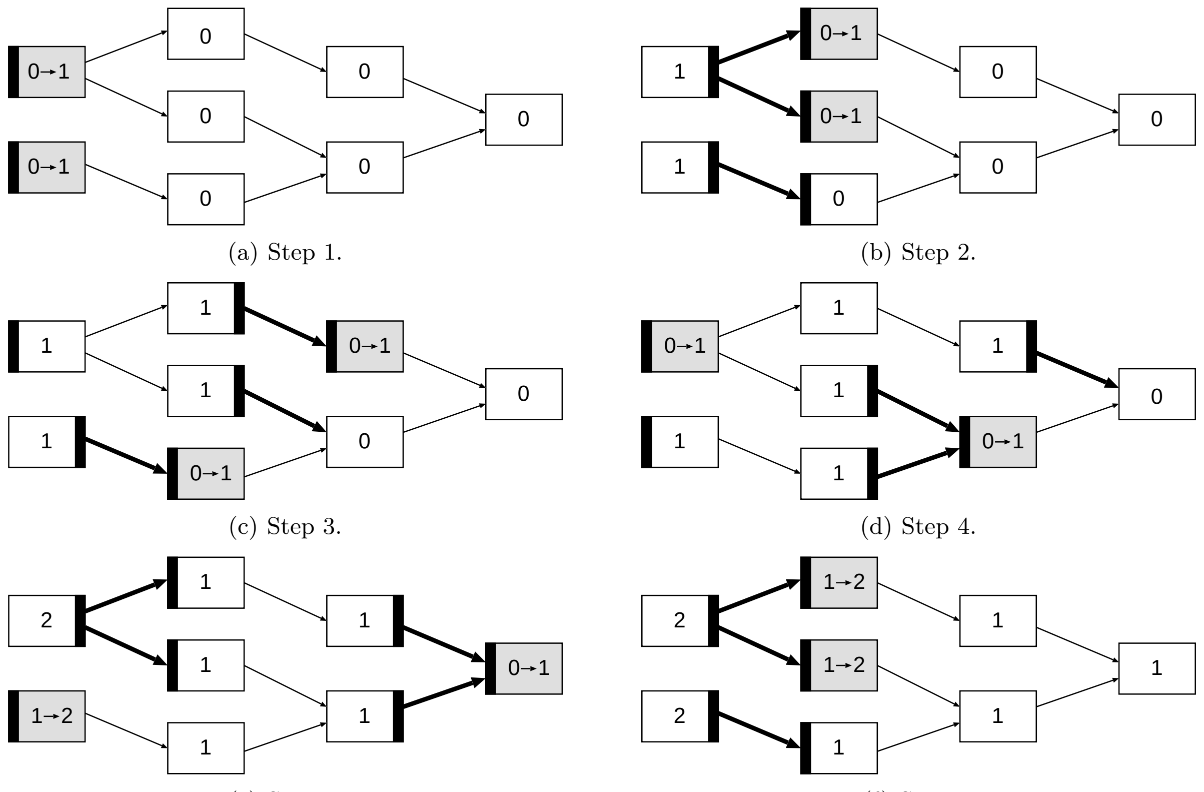
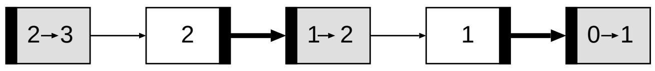
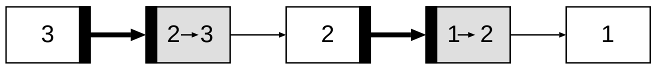


Figure 6. Example processing graph running on two CPUs. It is assumed here that processing of each tile takes the same time.

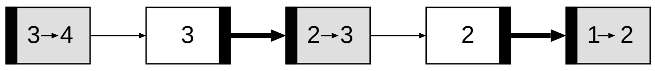
An example of this scheduling algorithm is presented in Fig. 6. In this figure, PTs that are *ready-to-run* are indicated with a black bar at the left (input) side. PTs that are currently being processed are depicted in grey color. When they have their output data available, this is indicated with a black bar at the right (output) side. Active edges between PTs are shown with bold lines. The number in the PT denotes the current sequence-number of the outputs. In this example, we have assumed that processing takes equal time-periods for every tile. Note that this is not required by the scheduling algorithm, in which the processing time is not defined. Also



(a) Step 1.



(b) Step 2.



(c) Step 3.

Figure 7. Parallel processing in a pipeline. Only every second tile can run in parallel, because the output memory of the previous tile is shared with its successor.

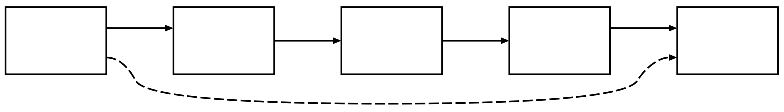


Figure 8. A pipeline with a short-cut connection (dashed line). The short-cut is preventing any parallelism in the pipeline.

note that the shown schedule is not the only possible one. In Fig. 6(b), instead of processing the two top tiles in the second column, any two tiles from the second column would be a valid next step. The step in Fig. 6(f) is similar to (b), just with the sequence-numbers increased to the next frame.

A second example is depicted in Fig. 7. This example considers a simple processing pipeline. Note that only every second tile can be processed independently, because the output of the previous tile is still locked by the successive tile. This situation can be resolved by adding buffering tiles between pairs of PTs. These buffering tiles just provide a copy of their input, thus releasing the data-dependency relation between two successive tiles, which can then be processed in parallel.

Another useful case for buffering tiles is a situation like shown in Fig. 8. The processing is basically a pipeline, but the output of the first PT is also required at the last PT. Because of this dependency, no new data can be processed in the first tile until the last tile has been processed. This results in no parallelism in the pipeline. The solution to this problem is to insert an equal number of buffering tiles in the short-cut path, such that also the second output of the first PT can be delayed.

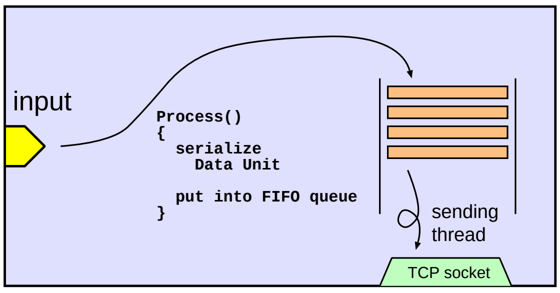
3.5. Network Transfer

In order to connect two processing graphs on two different computers, the data processed on one computer must be sent to the input at the second computer. This is realized with a network-sender PT and a network-receiver PT. The network-sender PT uses the `Serialize` method of the Data-Unit interface to generate a bit-stream representation of the data. This bit-stream is then stored in a FIFO buffer from which it is sent over the network. The sending is carried out in a separate thread which is managed by the PT itself instead of a global scheduler. This has the twofold advantages that (1) streaming can run with maximum throughput because the thread is always active, and (2) the network-connection PTs can also be used without any PGC. Since the sending thread is almost always blocked in the system-function for network transmission, its computation time is negligible.

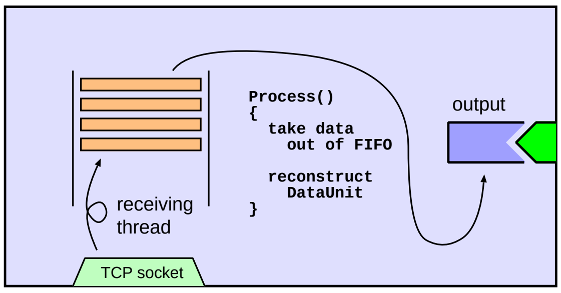
The network-receiver PT at the other side works similarly to the network-sender. A separate thread is responsible for receiving new data bit-streams from the network. Whenever the PT is triggered, one data packet is removed from the FIFO and used to reconstruct a Data-Unit.

3.6. Distributed Processing Graph

A *Distributed Processing Graph* (DPG) wraps a PGC into a network interface. When DPGs are created on different computers, they can be joined together with a control connection. From that moment onwards, both



(a) Network-sender PT.



(b) Network-receiver PT.

Figure 9. Internals of the PTs for network transmission of streamable Data-Units.

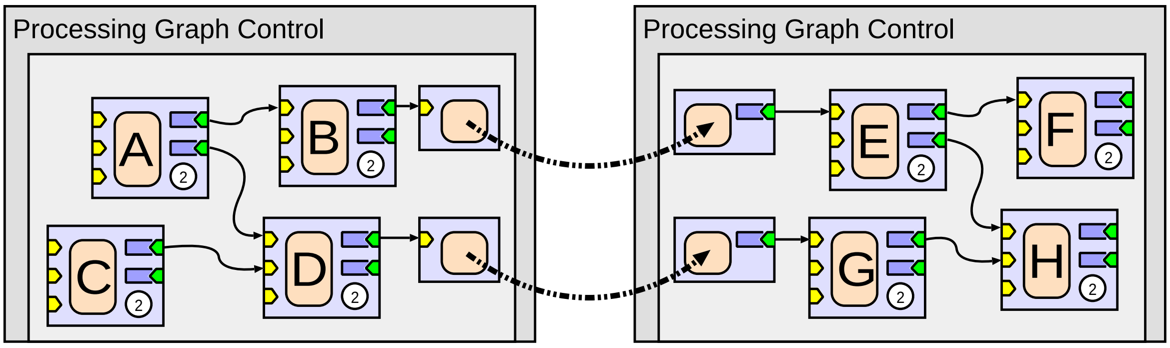


Figure 10. The *Distributed Processing Graph* object manages the PTs and uses the PGC to schedule the tasks on the local computer. Connections to PTs on other computers are made via pairs of network-sender / network-receiver PTs.

graphs in the DPGs appear as a joint graph and can also be controlled in a unified way, even though the PTs might be on different computers. New tiles can be created on any computer via the DPG interface by specifying the name of the PT and the system on which it should be instantiated. *Universally Unique Identifier*s (UUIDs) are used to access the PTs in the network.

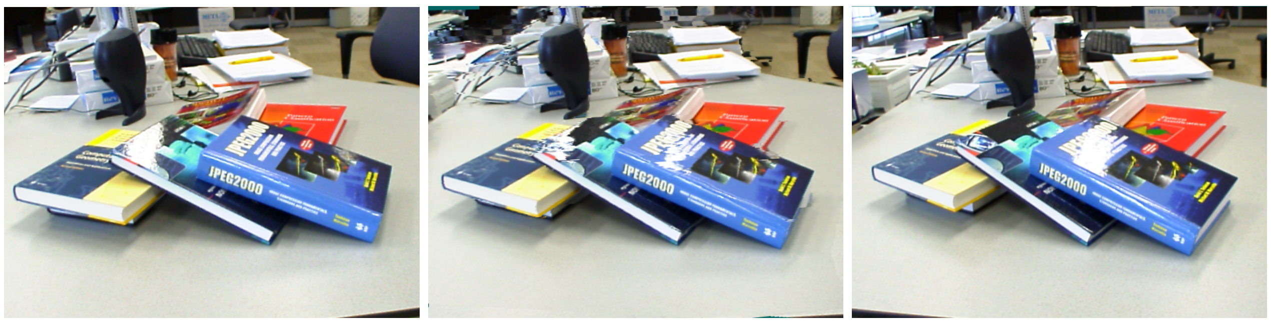
In the general case, many computers with DPG daemons running on them can be connected into a control tree. Each DPG stores which PTs can be reached via each of its neighboring computers. Control commands for a non-local PT are forwarded to the closest neighbor DPG, which then further handles the request.

The DPG interface can also be used to connect pairs of PTs. If both PTs are on the same computer, a simple direct connection is established like before. If the PTs are on different computers, special PTs are added that serialize the data into a network stream on one computer, and then reconstruct the Data-Unit on the receiving computer (like described in Section 3.5). These network-transmission PTs are connected to the pair of PTs that originally should be connected. This process is transparent to the user of the DPF. Note that the data-transfer connections are independent from the control connections between DPGs. Hence, while the control connections always have a tree topology, data is transferred directly between the involved pair of computers.

In order to run our distributed processing framework on a cluster of computers, a DPG is started as a daemon process on each computer. One computer acts as the control computer, running the actual application program. Note that the DPG daemons are independent of the application program as long as all PTs required by the application are compiled in the DPG daemons. Future work will provide a method to also transmit the PT code over the network and link it dynamically to the DPG.

4. EXAMPLE APPLICATION: MULTI-VIEW VIDEO

As an example application, we implemented a multi-camera view-interpolation system which allows to synthesize images along a set of cameras.[5] An example result of our view-interpolation system is depicted in Fig. 11. For the view-interpolation system, the following tasks have to be carried out.



(a) left view (b) synthesized middle view (c) right view

Figure 11. An example result of the view-interpolation system. The left and the right image are the captured images, while the middle view has been synthesized using these two images and the depth-map estimated from them.

- **Capturing the video streams from the digital cameras.** Uncompressed video streams are captured by DCAM cameras, connected via the FireWire bus. The high data-rate limits the number of cameras to only two per bus. Hence, one capturing computer is required for each pair of cameras.
- **Correction for the radial lens distortion.** Wide-angle lenses introduce lens distortion in the image, which has to be compensated for. This is a geometric transformation between points $\mathbf{p} = (x, y)^\top$ and $\mathbf{p}' = (x', y')^\top$ that can be described as

$$\mathbf{p}' = \mathbf{c} + (\mathbf{p} - \mathbf{c}) \cdot (1 + \alpha_1 r + \alpha_2 r^2 + \alpha_3 r^3 + \cdots), \qquad (1)$$

  with $r = |\mathbf{p} - \mathbf{c}|$, and $\mathbf{c}$ being the image center. A direct implementation of this transformation on the CPU is complex, but it can be implemented very efficiently in the GPU by "texture-mapping" the input image onto a deformed 2D-mesh (Fig. 12).
- **Stereo image rectification.** In order to enable a fast computation of depth images, most algorithms require that the input images are rectified (corresponding scanlines are horizontal). This can be achieved with a geometric homography transformation that can also be implemented efficiently on the GPU. In fact, this transformation and the previous correction for the radial lens distortion can be combined into a single geometric transformation, but we still consider them here as two independent operations.
- **Depth estimation.** Estimating the depth from two rectified images is the most computation-intensive process. Because of the complexity, algorithms have been developed to perform depth-estimation on the GPU. However, current algorithms providing high-quality result work on complicated data-structures. Hence, they are best implemented in the CPU. A combined approach are algorithms that carry out the first steps on the GPU, but which compute the final result on the CPU. This approach is efficient if the first step consists of computing a correlation volume (correlation between the left and right view for every possible disparity), which is then used in a second step to extract globally optimal disparity surfaces.
- **View interpolation.** The synthetization of arbitrary views from a set of images and corresponding depth images consists of a per-pixel warping according to the depth image (which is best implemented on the CPU) and a global transformation to cancel the image rectification (which is best implemented on the GPU). Since the final result is going to be displayed, the final result from view interpolation does not have to be transferred back from the graphics-card memory to main memory.

These sub-tasks can be implemented in separate PTs. Note that some of the tasks (lens undistortion, rectification, and part of the view interpolation) are implemented on the GPU, while the depth estimation is implemented on the CPU. At the connection between GPU processing and CPU processing, there are conversion PTs to transfer the data between graphics-card memory and main memory.

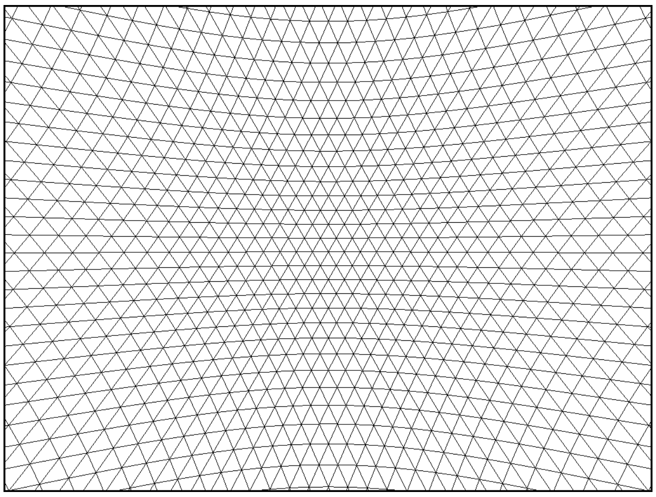


Figure 12. The image with corrected radial distortion is obtained by approximating the undistortion transformation with small triangular patches. The texture-mapping for each triangle can be carried out efficiently on OpenGL hardware.

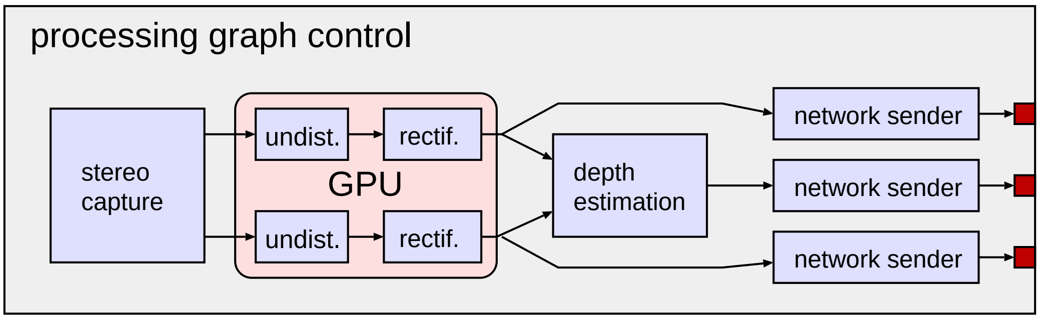


Figure 13. The processing graph on one computer for capturing two camera streams, correcting for the lens distortion and rectifying the images on the graphics card, and estimating the depth between the two images.

5. ADDITIONAL FRAMEWORK ENHANCEMENTS

5.1. Parallel Algorithms

Because the depth-estimation process is computationally expensive, it is attractive to parallelize this algorithm itself. Even though parallelization *within* one PT is not supported directly by our framework, it can be easily achieved by splitting the PT into separate PTs which compute a partial solution each. The results are then combined into a final solution in a multiplexer PT (see Fig. 15).

5.2. Cyclic Processing Graphs

In some applications, circular data-dependencies between processing tiles exist. One example are tracking applications, in which the previous state is used to predict the state in a successive frame. If the tracking algorithm is not concentrated in a single PT, this results in a circular dependency in the processing graph. The scheduler described above does not support this circular dependency. A cycle in the processing graph would restrict the processing such that the PTs in the cycle have to be processed sequentially, since the result of the previous frame has to be ready when the next frame is processed. We resolve this dilemma by introducing *weak* inputs at processing tiles. Data is also transferred to these inputs, but it is not required that the data at weak inputs has the same sequence number as the other inputs. This means that the data at weak inputs may be several frames delayed (Fig. 16). The exact number of frames is unknown and may even change during the program run because of variations in the scheduling. However, this is sufficient for inputs that are used, e.g., for a state prediction. Assume, for example, that a PT is computing the position of an object in the current frame. It does

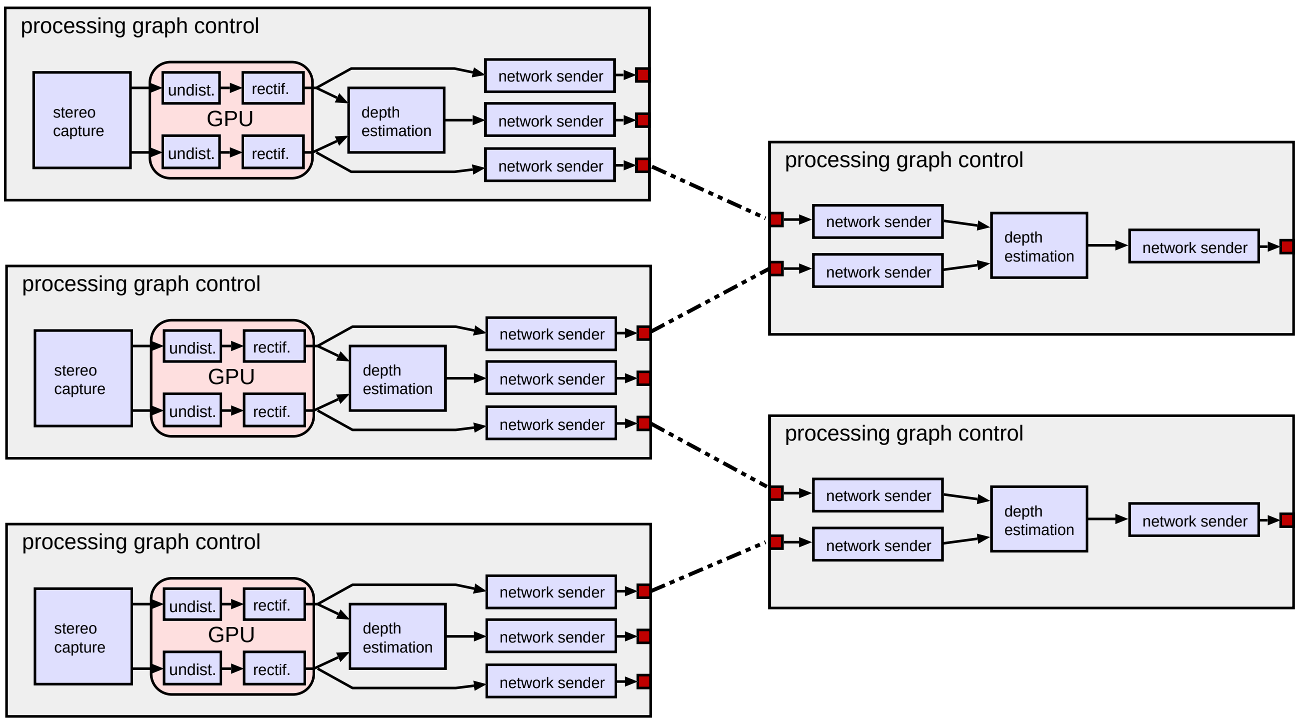


Figure 14. Multi-camera view interpolation as an example application for our distributed processing framework. For six cameras, this system can be implemented using five computers, where three computers record two camera streams each. Furthermore, they carry out the geometric transformations and the depth estimation between these two cameras. Two additional computers carry out the depth estimation between the two remaining pairs of cameras connected to different computers.

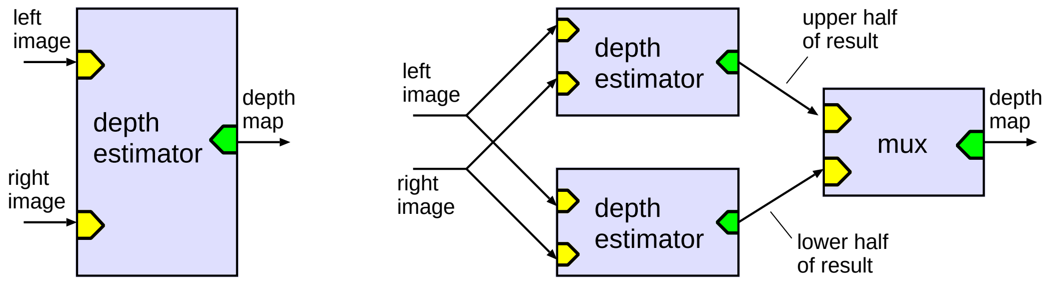


Figure 15. A PT with a non-parallelized algorithm (left side) can be easily replaced by a parallelized algorithm in which each PT computes only part of the result (right side).

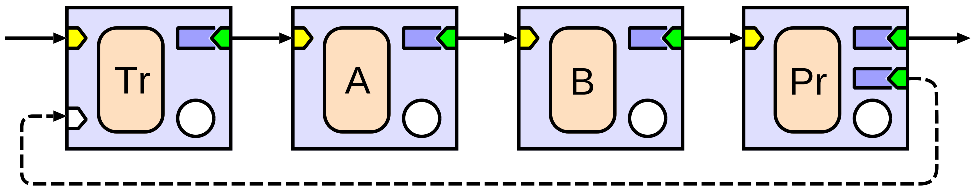


Figure 16. Processing graph with a cycle (dashed edge). The PT 'Tr' uses the state prediction generated in PT 'Pr'. If this graph is processed in parallel as a pipeline, the delay of the prediction (along the dashed edge) can be up to two frames, instead of only one.

so by including the information about where the object has been located in the previous frames. To this end, it is sufficient to know the positions at *some* moments in the past. The current position can be extrapolated using the object speed and the difference in the sequence numbers between the current frame and the data at the prediction input.

6. CONCLUSIONS

This paper has presented a software framework to provide an easy-to-use platform for parallel and distributed processing of video data on a cluster of SMP computers. Existing algorithms can be easily integrated into the framework without reimplementation of the algorithms. The framework organizes the processing order of the algorithms in a graph of atomic processing tiles with unrestricted topology. Parallelization is carried out automatically using a pool-of-tasks scheme.

A specific feature of our framework is that it can be used at three different levels, each comprising comprise a subset of the framework. During the development, PTs can be directly connected without parallelization, to simplify development and debugging. At a second level, a scheduler is added which automatically parallelizes the execution on SMP machines. Finally, in a third level, the processing graph can be distributed over a cluster of computers. As such, the framework provides a scalable approach for distributed processing, without introducing much burden on the programmer.

The framework has been successfully applied on a cluster of multi-processor computers to implement various video-processing tasks, including the described view-interpolation system for multiple input cameras. Because of its flexibility, the framework can be applied to a wide range of applications, probably even in fields other than video processing.

REFERENCES

1. F. Seinstra, D. Koelma, and A. Bagdanov, "Towards user transparent data and task parallel image and video processing: An overview of the parallel-horus project," in *Proceedings of the 10th International Euro-Par Conference (Euro-Par 2004)*, *Lecture Notes in Computer Science* **3149**, pp. 752–759, Aug. 2004.
2. D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer, "Beowulf: A parallel workstation for scientific computation," in *Proceedings of the International Conference on Parallel Processing*, 1995.
3. I. Geys, T. P. Koninckx, and L. van Gool, "Fast interpolated cameras by combining a GPU based plane sweep with a max-flow regularisation algorithm," in *Second International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT'04)*, pp. 534–541, 2004.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Professional Computing Series, Addison-Wesley, 1995.
5. D. Farin, Y. Morvan, and P. H. N. de With, "View interpolation along a chain of weakly calibrated cameras," in *IEEE Workshop on Content Generation and Coding for 3D-Television*, June 2006.
6. M. Korch and T. Rauber, "Evaluation of task pools for the implementation of parallel irregular algorithms," in *International Conference on Parallel Processing Workshops (ICPPW'02)*, pp. 597–604, 2002.
7. J. Hippold and G. Runger, "Task pool teams for implementing irregular algorithms on clusters of smps," in *International Parallel and Distributed Processing Symposium (IPDPS'03)*, pp. 54–61, 2003.