Automatic Distribution of Vision-Tasks on Computing Clusters

Thomas Müller and Binh An Tran and Alois Knoll

Robotics and Embedded Systems
Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany

ABSTRACT

In this paper a consistent and efficient but yet convenient system for parallel computer vision, and in fact also realtime actuator control is proposed. The system implements the multi-agent paradigm and a blackboard information storage. This, in combination with a generic interface for hardware abstraction and integration of external software components, is setup on basis of the message passing interface (MPI).

The system allows for data- and task-parallel processing, and supports both synchronous communication, as data exchange can be triggered by events, and asynchronous communication, as data can be polled, strategies. Also, by duplication of processing units (agents) redundant processing is possible to achieve greater robustness. As the system automatically distributes the task units to available resources, and a monitoring concept allows for combination of tasks and their composition to complex processes, it is easy to develop efficient parallel vision / robotics applications quickly. Multiple vision based applications have already been implemented, including academic, research related fields and prototypes for industrial automation. For the scientific community the system has been recently launched open-source.

Keywords: Parallel Computer Vision, Computing Clusters, Open-Source Framework

1. INTRODUCTION

Distribution of computer vision tasks in parallel environments is essential considering the increasing demand for computational resources to accomplish advanced visual processing tasks. While recent industrial approaches primarily focus embedded solutions, namely advanced parallel visual processing on DSP and / or processors with embedded image processing units are developed, this paper shows a different solution on the large scale and an implemented prototype for limp object recognition and robotic handling.

The vision clearly is a dynamic distributed system sharing the image processing load on whatever hardware setup available in the current environment. Think of a mobile device with an embedded camera (e.g., a smart phone) on which the user wants to perform complex visual processing. Operations like monocular 3D reconstruction or visual tracking certainly exceed capacities of such light weight devices. So the goal would be to only trigger computation on a user device and perform the actual computation on powerful remote devices. Another scenario within mobile robotics is obvious: the robots here need to be as light and flexible as possible without power-consuming electronic devices. Again, a visual processing task could be delegated to remote computers with suitable capacities. Thus, a consistent and efficient but yet convenient system for parallel computer vision, and in fact also realtime actuator control is proposed. The system implements the multi-agent paradigm and a blackboard information

Further author information: (Send correspondence to T. Müller)
E-mail: muelleth@cs.tum.edu, Telephone: +49 (0)89 289 25761

storage. This, in combination with a generic interface for hardware abstraction and integration of external software components, is setup on basis of the message passing interface, which is the de facto standard for HPC environments and hence provides support for cluster computation on a large variety of platforms.

The proposed system allows for data- and task-parallel processing, and supports both synchronous, as data exchange can be triggered by events, and asynchronous, as data can be polled, communication strategies. Also, by duplication of processing units (agents) redundant processing is possible to achieve greater robustness. Furthermore, as the system automatically distributes the agents to available resources, and a workflow concept allows for combination of tasks and their composition to complex processes, it is very easy to develop vision / robotics applications quickly. For evaluation, multiple vision based applications have already been implemented, e.g. an evolutionary approach for learning visual saliency features, or a parallel active perception system for robotic recognition and handling of limp objects.

2. RELATED WORK

This section investigates how the proposed system for parallelization of vision tasks fits into findings / results of existing recent research in related fields. Here, relevant and inspiring sophisticated approaches appear primarily within the area of cognitive and blackboard architectures, or multi-agent systems.

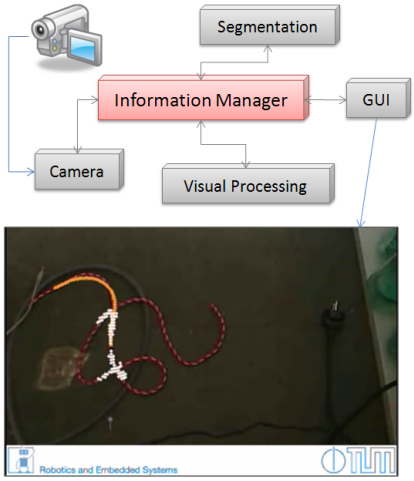
Cognitive architectures originate from psychology and by definition try to integrate *all* findings from cognitive sciences into a general computational framework from which intelligence may arise. Multiple systems have been proposed to fulfill this requirement, including Act-R[12] and Soar.[3] Although these approaches may be biologically plausible and have the potential to form the foundation of some applications in reality, they all face the problem of being a mere scientific approach to cognition. We argue that, in order to develop applications also suitable for industrial scenarios, a framework for intelligent robotics and parallel vision has to be designed keeping the application scope in mind. Thus, additional requirements like efficient distribution on multiple individuals - which is certainly out of scope for a cognitive architecture have to be taken into account here. However, the performance of the biological visual apparatus is outstanding and thus, although we do not propose a cognitive architecture, the proposed system strives to integrate relevant aspects where found useful and appropriate.

The principle theory considering **blackboard architectures** is based on the assumption, that a common database of knowledge, the "blackboard", is filled with such by a group of experts.[4] The goal is to solve a problem using contributions of multiple specialists. We adopt the basic ideas of this concept in the implementation of the information manager of the proposed framework (see Section 3). Nevertheless, one drawback with traditional blackboard systems is the single expert, i.e., a processing thread, that is activated by a control unit.[5] There is no strategy for concurrent data access in parallel scenarios. Futhermore, there is no means for training an expert over time, e.g., applying machine learning techniques, or even exchanging a contributor with another in an evolutionary fashion. We also try to overcome with these shortcomings within the proposed framework.

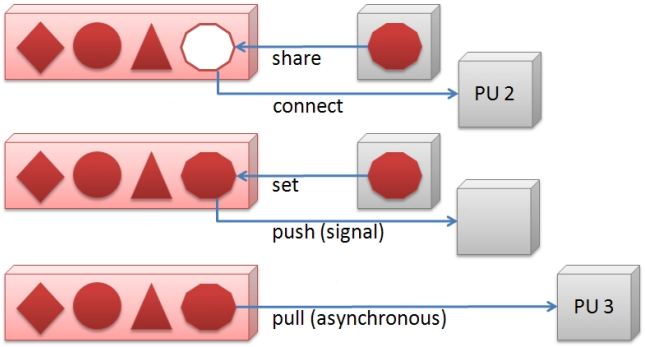
Finally, a **multi-agent system** (MAS) is a system composed of a group of agents. According to a common definition by Wooldridge[6] an agent is a software entity being *autonomous*, acting without intervention from other agents or a human; *social*, interacting and communicating with other agents; *reactive*, perceiving the environment and acting on changes concerning the agent's task; and *proactive*, taking the initiative to reach a goal. Most existing implementations (e.g., JADE[7]) use a communication paradigm based on FIPA's agent communication language,[8] which is designed to exchange text messages, not complex data items. Thus we instead implement the blackboard paradigm which is capable of maintenance of complex items. Nevertheless, we acknowledge the above definition and the fact, that agents may concurrently work on a task and run in parallel. The processing units of our framework are hence implemented according to these MAS paradigms.

3. PARALLEL VISUAL PROCESSING

As we propose a convenient and configuration-less system for parallelizing visual processing applications, this section is dedicated to elaborate on the topic of parallelization on a single computer, with a strong focus on parallel visual processing. Earlier work in this field[9,10] has already been published and is only briefly review here. The next section will then describe how the proposed approach can be generalized for computing clusters.



(a) Multiple processing units (grey) communicate via the information manager (red).



(b) Processing units support both data push and data poll paradigms. Shared data items allow read and write access to any connected unit.

Figure 1. Parallel visual processing and data exchange.

Crucial to any parallel system is the choice on which hierarchical (semantic) layer of abstraction the parallelization takes place. Traditional auto-parallelization systems like OpenMP[11] require the extensive usage `pragma` statements within the source code. The compile interpretes the statements and the result is a parallel application. The approach works well, if one only wants to parallelize a simple sequential application without in depth knowledge of synchronization and parallelization techniques. Other traditional approaches utilizing multithreading approaches work well, but they require expert knowledge on shared data access, function binding, avoidance of dead locks, and the like.

The system presented here encapsulates the actual parallelization from the application developer. The programmer only needs to define three essential structural components:

- *Decoupled Tasks*
  On whatever level of semantic abstraction, the developer decides which tasks shall run in parallel, i.e., which tasks can be seen as a semantic entity.
- *Shared Data*
  The data sharing mechanism enables quick definition of shared data structures. The system automatically provides the synchronization and mutually exclusive access functionality.
- *Application Workflow*
  An application workflow is defined by its data-flow. Essentially, this means that each task may specify events. Events can be used to determine the sequence of operations within a task, or when shared (made available globally), trigger operations within other tasks.

For example, considering a typical application for object recognition and tracking. The application comprises image acquisition, segmentation, recognition and tracking, and result visualization. Regarding the point of view of a parallelization engineer, these operations need not necessarily run in a sequential order, but instead can be decoupled into separate processing tasks. This already defines the first structural component (see Figure 1). Later, the software engineer has to implement the task logic into a *processing unit*(PU).

Next, it has to be defined, which data structures need to be transferred from one task to another. In the example, the camera unit would need to transfer the input image to the segmentation unit, while this would need to transfer the result image to the recognition and tracking unit, and so forth. The system now allows to decouple these tasks automatically, as there is no reason not to load the next image from the camera, while the current image is still being processed. Having defined the shared data structures, thread-safe read and write access as well as synchronization is intrinsically provided by the parallelization framework. The framework realizes this functionality by utilizing a "blackboard" for information management. Once a datum is shared, any `set` or `get` operation automatically results in a post or pull (and required synchronization operations) on the global information storage.

Finally, specifying an application workflow is straight forward. Up to this point, the information manager is only a global storage for data. Now, introducing event management, the storage is extended to be the instance of workflow management within an application. Any shared data item can be defined to emit a signal to any connected processing unit whenever the datum changes. By default, all processing units implement an event listener, so, reacting on incoming events enables the software engineer to conveniently introduce the desired workflow. Considering the above example, one could implement events in the GUI unit (see Appendix) to trigger start and stop of the segmentation.

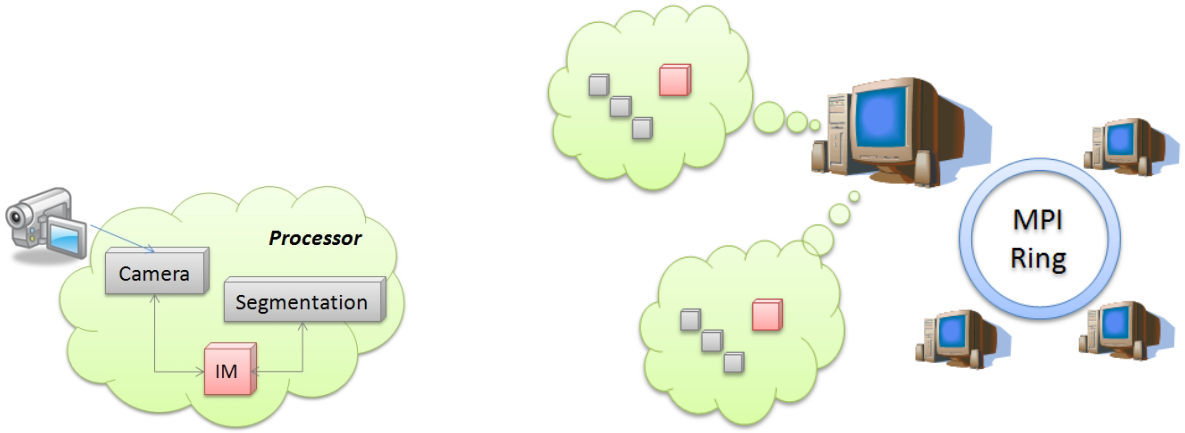
4. TOWARDS DISTRIBUTED SYSTEMS

In the last section parallel vision systems on a single computer using the intrinsic multithreading capabilities of the framework was discussed. The programmer is in this case able to specify and implement parallel applications without actually dealing with distribution of tasks and synchronization issues.

This section elaborates on the automatic process of task distribution whenever an application built with the framework runs on a computing cluster. For this task an implementation of the *Message Passing Interface* (MPI)[12] is used as a middleware. MPI is the de facto standard for high-performance computing (HPC) environments and hence provides support for a large variety of platforms and has a huge user pool. MPI also supports a wide range of cluster participants, from very few or even a single computer to several hundreds or thousands. The standard allows for composition of computing environments by means of joining individuals to the so-called MPI-ring. The MPI implementation even chooses the fastest channel for data exchange within the participants of the ring (be it e.g., shared memory, ethernet, infiniband, ...). Also, extending or setting up a MPI-ring is pretty straight forward, as it only requires to start the MPI system service (daemon) on a computer. When joining a ring, the administrator specifies the number of processors for a resource. Note, that in the MPI-terminology, a *processor* does not specify a CPU as such, but in fact a process started on a computer. The number of processors need not match the number of CPUs available on a computer, but can be set to an arbitrary number. This allows for simulating cluster setups on a single computer - an essential mechanism for debugging. In a productive environment though, it makes sense to specify the number of processors according to the number of CPUs.

When an application built with the proposed framework is run, the user has two options:

- *Direct Execution*
  The application is run by executing the compiled file directly. This could be called local execution also. In this case the application automatically parallelizes on the local, currently used computer according to the structural components described in the last section.



(a) Units are bound to processors at runtime. The user may force binding to a computer by IP-address.

(b) The user specifies the number of processors for each computer. Information managers of each processor are synchronized via MPI.

Figure 2. Towards a distributed multi-parallel application.

- *MPI Execution*
  The application is run using the MPI tools, e.g., `mpiexec`. This results in distributed execution. In this case, at boot time of the application (see Appendix for a code snippet) the application environment automatically checks for presence of processors within the MPI-ring.

In the case of distributed execution using MPI, the application's processing units are automatically distributed to available processors (see Figure 2). Processors in the sense of MPI do only carry a unique identifier, the so-called *rank*, where the processor on which the application is started has rank zero. Within the proposed framework, processors also carry information about their host computer (e.g., the IP-address) and process information (e.g., pid). Furthermore, each processor is initialized with its own information manager for data exchange and synchronization. At runtime, the information managers synchronize using MPI methodology*.

Unfortunately, as the main task of the proposed work is visual processing, the mechanisms described above do not work as well as expected. This has some obvious reasons: MPI and most of its implementations, e.g., MPICH2 are designed to transmit messages, i.e., preferentially text messages or small data chunks. Transmission of for example strings, short integers or even matrices and vectors does work as expected, i.e., reaches realtime performance easily on recent network / hardware infrastructures. But compared to matrices and the like, where serialized messages have the size of a few (hundred) bytes, a serialized image can easily reach a few million bytes (several mega-bytes). Thus, it was necessary to implement a special synchronization mechanism for image transfer† between information managers of different processors. In practice, the sychronization for images is based on on-demand image streaming.

Whenever an information manager of a processor requires an image that is not only connected on the same processor, a bi-directional image stream is connected between the information managers of the

*An implementation detail: the MPI calls are not directly executed, but the powerful MPI wrapper utilities of *Boost.MPI* (`http://www.boost.org/doc/libs/1_44_0/doc/html/mpi.html`) were used. This allows for convenient transmission of arbitrary datatypes given that they are serializable with the *Boost.Serialization*-engine (`http://www.boost.org/doc/libs/1_44_0/libs/serialization/doc/index.html`).

†To be exact: within the implementation a specialization of the general templated `Synchronizer`-class was introduced.

processors hosting the connected processing units. The image streams are encoded / decoded on the fly using the *Xvid*-Codec[‡]. The drawback of this approach is the loss of image quality on a roundtrip. But in practice this is not of much relevance, as most applications do not implement extensive round-trips, but image processing units usually generate new or modified intermediate / result images, or abstract result representations like object classifications or locations. The advantage is obvious: now it is possible to distribute visual processing to a cluster of computers which enables much higher performance and excellent scaling given that the application's structural components are well-defined - and the user does not have to deal with the parallelization across multiple computers, as this is handled automatically by the framework.

Nevertheless, in some cases it might be useful, if not necessary, to have a possibility to specify at least the computer that a processing unit has to run on. For example, a camera device might be connected to a certain computer or the GUI needs to be displayed on some dedicated display. To cope with this requirement, processing units provide a functionality to enforce a physical binding. Whenever the physical binding is specified, the configured unit is only run on the requested machine, forcing the application to terminate if no processor is found on the requested computer. By default though, the physical binding is left empty and the framework decides autonomously, where to run the units.

5. DEMO APPLICATION

This section briefly describes an application developed with the framework. The application setup shown in Figure 3 is used as a prototype for an industrial system in the SFB 453 (see Acknowledgments). The goal of the overall task is to grasp both ends of a linear deformable object (cable, thread, etc.) in whatever initial position it is on a platform, and agglutinate them[§].

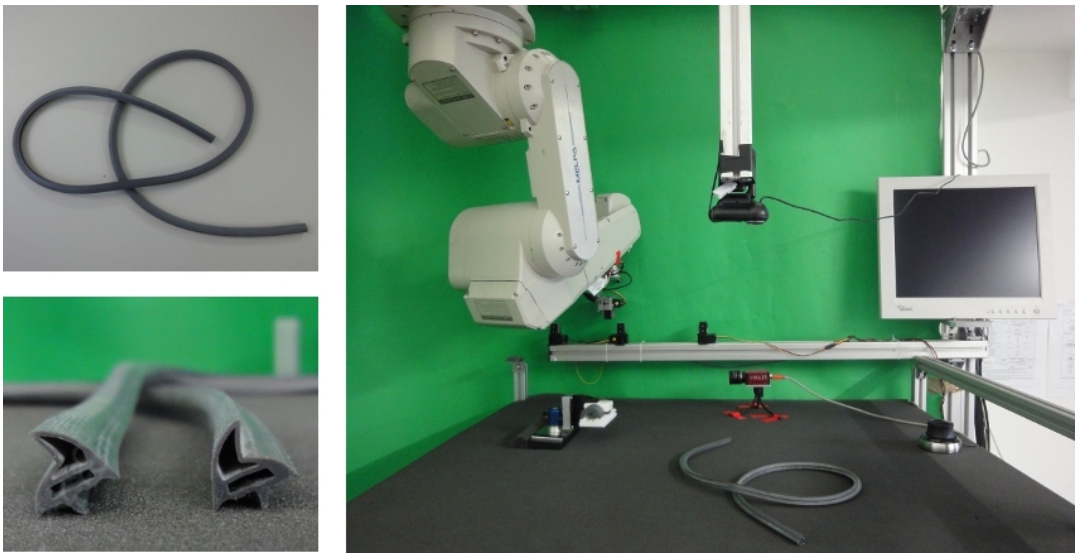


Figure 3. The demo application scenario setup.

The application (see Figure 4) comprises an image acquisition unit, connected to the camera device; visual processing units, i.e., segmentation, object recognition and obstacle detection units; action planning; actuator control units, i.e., the robot and the gripper units; a GUI unit for visualization of results and interactive event generation (buttons and other control items like sliders); and a space mouse unit for interactive robot motion control.

[‡] `http://www.xvid.org/`

[§] A video showing the whole task can be reviewed from `http://www.youtube.com/watch?v=z2sS52YnVrk`.

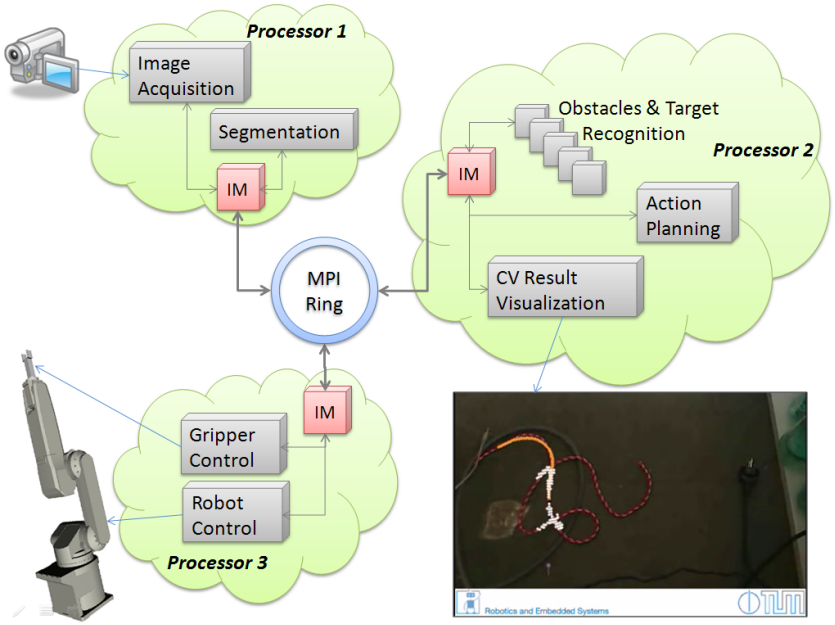


Figure 4. A possible unit setup of the application for limp object recognition and robotic handling. The actual mapping of units to processors and of processors to computers depends on the MPI environment.

In this application, the action planning unit receives, sends and coordinates events from several units to form the application workflow, while segmentation input is connected to the camera image, segmentation results are connected to the structure analysis units, and so on. To clarify on the introduced workflow, part of the it is shown in Figure 5.

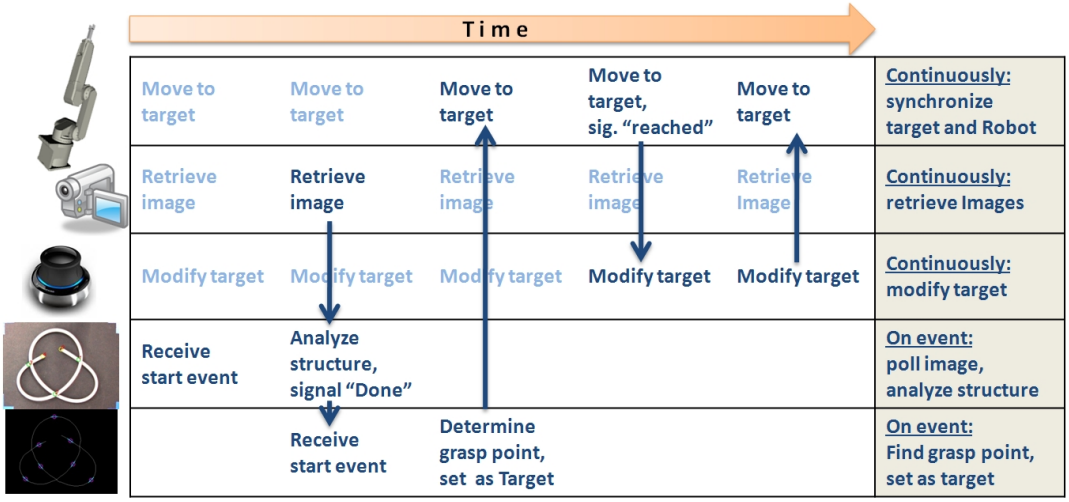


Figure 5. Schema of a subsequence of the introduced demo application workflow[13]

6. CONCLUSION AND FUTURE WORK

In this paper, a system for parallel visual processing of arbitrary kind is introduced. The system parallelizes tasks using multi-threading techniques on a single computer. Shared data access, synchro-

nization and avoidance of deadlocks is accomplished by means of the information manager provided by the framework. Considering distributed visual processing on computing clusters, the systems provides the necessary infrastructure on the basis of the message passing interface and on-demand image streaming. Conveniently, the parallelization is completely transparent to the user, so at compile-time, the developer need not make any choice about the underlying hardware setup. Any application can be run on a single computer or a MPI ring without changing the configuration.

Still, at the time of writing this paper, some further ideas are on the way of being integrated. Most prominently, as the system distributes tasks (units) at startup using a simple algorithm, no measure on the runtime load produced by the units is taken into account for parallelization. A solution, being implemented at the moment, extends both the processors and processing units to hold information about load currently caused on a computer. This information will then be used to re-distribute units whenever the load exceeds some limit - the goal here is to establish a dynamic load balancing mechanism.

Nevertheless, the framework performs and scales well, whenever the structural components are designed with care. It is also already used within many applications and proved to be easy to use and install, as for example for Linux, only resources from standard repositories are used. Furthermore, it was launched open source to the community recently (see `http://www.flexiblerobotics.com` for details, implementation examples and demo applications).

ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG) within the Collaborative Research Center SFB 453 on *High-Fidelity Telepresence and Teleaction*[¶].

REFERENCES

[1] Newell, A., [*Unified Theories of Cognition*], Harvard University Press (1994).

[2] Anderson, J. R., [*How Can the Human Mind Occur in the Physical Universe?*], Oxford University Press (2007).

[3] Lehman, J. F., Laird, J., and Rosenbloom, P., "A Gentle Introduction to Soar, an Architecture for Human Cognition." 14-Sept-2009, http://ai.eecs.umich.edu/soar/sitemaker/docs/misc/GentleIntroduction-2006.pdf.

[4] Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, D. R., "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty," *Computing Surveys* **12**(2), 213–253 (1980).

[5] Corkill, D. D., "Collaborating Software: Blackboard and Multi-Agent Systems & the Future," in [*Proc. of the International Lisp Conference*], (2003).

[6] Wooldridge, M. J., [*An Introduction to MultiAgent Systems*], John Wiley & Sons, 2nd ed. (2009).

[7] Bellifemine, F., Caire, G., Poggi, A., and Rimassa, G., "JADE: A White Paper," tech. rep., Telecom Italia Lab and Universita degli Studi di Parma (2003).

[8] "FIPA ACL Message Structure Specification," tech. rep., Foundation for Intelligent Physical Agents (2002).

[9] Müller, T. and Knoll, A., "A generic approach to realtime robot control and parallel processing for industrial scenarios," in [*Proceedings of the IEEE International Conference on Industrial Technology*], (2010).

[10] Müller, T., Ziaie, P., and Knoll, A., "A Wait-Free Realtime System for Optimal Distribution of Vision Tasks on Multicore Architectures," in [*Proceedings of the 5th International Conference on Informatics in Control, Automation and Robotics*], (2008).

[11] Chapman, B., Jost, G., and van der Pas, R., [*Using OpenMP*], MIT Press (2007).

[¶] `http://www.sfb453.de`

[12] "MPI, A Message-Passing Interface Standard," tech. rep., University of Tennessee, Knoxville, Tennessee (June 1995).

[13] Müller, T., Tran, B. A., and Knoll, A., "A FLEXIBLE ROBOTICS AND AUTOMATION SYSTEM: Parallel Visual Processing, Realtime Actuator Control, and Task Automation for Limp Object Handling," in [*Proceedings of the 7th International Conference on Informatics in Control, Automation and Robotics*], (2010).

APPENDIX

Creating a distributed visual processing application using the proposed system can be achieved conveniently with only very few lines of code[‖].

Listing 1. Implementing a distributed application

```
#include <frf.h>

using namespace frf;

int main(int argc, char **argv)
{
  /// Boot distributed environment
  boot(argc, argv);

  /// Setup processing units
  vision::camera::OpenCVCamera* cam = new vision::camera::OpenCVCamera();
  vision::processing::Segmentation* seg = new vision::processing::Segmentation();
  gui::GuiUnit* gui = new gui::GuiUnit();

  /// Initialize the processing units
  init();

  /// Connect the shared parameters
  unsigned int id;
  id = cam->share<data::Image>(vision::camera::OpenCVCamera::LIVE_IMAGE);
  seg->connectTo(vision::processing::Segmentation::INPUT_IMAGE, id);
  gui->connectTo(id, "Livestream", new gui::feedback::Image("USB Image"));

  id = seg->share<data::Image>(vision::processing::Segmentation::RESULT_IMAGE);
  gui->connectTo(id, "Livestream", new gui::feedback::Image("Result Image"));

  id = seg->shareEvent(vision::processing::Segmentation::START);
  gui->connectTo(id, "Livestream", new gui::control::Button("Start Segmentation"));

  id = seg->shareEvent(vision::processing::Segmentation::STOP);
  gui->connectTo(id, "Livestream", new gui::control::Button("Stop Segmentation"));

  /// Run distributed application
  return run();
}
```

In the listing, the `boot(argc, argv)` statement loads the application configuration. If the application was started within a valid MPI[12] environment, the call checks for presence of processors within the ring and initializes them - if not, the system simply assumes a single processor on the local computer. The call also loads an instance of the information manager for each processor.

[‖]The snippet is taken and slightly modified from the framework's website `http://www.flexiblerobotics.com`.

Lines 11–13 specify the units acutally performing processing tasks, e.g., for visual processing, loading images from the camera(s), displaying results or controlling additional sensor or actuator devices. Optionally, the application developer may specify the *physical binding* for a certain unit at this point. By calling `init()`, the application distributes the unit instances to available processors. As mentioned above, a processor is bound to ("lives" on) a computer, while a computer can specify multiple processors. For instance if a computer is a multicore machine, it makes sense to specify the number of processors according to the number of cores of that machine, when joining the MPI-ring. The next lines specify the connections for data exchange of units in the system. *Sharing* data allows access to a datum for other units via the information manager. The "globalized" datum is also automatically synchronized with information managers of other processors in the ring. Finally, `run()` starts the main event loop of the application and the processing units, and, on occurrence of a shutdown event, stops the application and finalizes the MPI environment.