A FRAMEWORK FOR MODULAR SIGNAL PROCESSING SYSTEMS WITH HIGH-PERFORMANCE REQUIREMENTS

Lukas Diduch

Information Access Division, NIST
Gaithersburg MD, USA
ldiduch@nist.gov

Ronald Müller, Gerhard Rigoll

Institute for Human-Machine Communication
Technische Universität München, Germany
rm@tum.de

ABSTRACT

This paper introduces the software framework *MMER_Lab* which allows an effective assembly of modular signal processing systems optimized for memory efficiency and performance. Our C/C++ framework is designed to constitute the basis of a well organized and simplified development process in industrial and academic research teams. It supports the structuring of modular systems by provision of basic data-, parameter-, and command-interfaces, ensuring the re-usability of the system components. Due to the underlying multi-threading capabilities, the applications built in *MMER_Lab* are enabled to fully exploit the increasing computational power of multi-core CPU architectures. This feature is carried out by a buffering concept which controls the data flow between the connected modules and allows for the parallel processing of consecutive signal segments (e.g. video frames). We introduce the concept of the multi-threading environment and the data flow architecture with its comfortable programming interface. We illustrate the proposed module concept for the generic assembly of processing chains and show applications from the area of video analysis and pattern recognition.

1. INTRODUCTION

'If you have an application that can benefit from parallel processing, make threading a priority in your company. The benefits can be enormous.[...]Plan for multi-core environments - not just dual-core. Assume there will be 4, 8, and even 16 cores in the future.'

Inspired by this citation from *Intel(R) Software Insight* magazine, issue on *Multi-core capability* [1](p.7), the team of the MMER (Multi-Modal Emotion Recognition) project at the Institute of Human-Machine Communication started to conceptualize and implement the software framework now published as *MMER_Lab* (lab.mmer-system.eu) in order to benefit from the new computational performance of the upcoming multi-core CPU and multi-CPU architectures in modern and future computer systems. In general this benefit can only be realized via internal parallelization of algorithms predominantly by separating logically independent functions such as large matrix operations. This often comes only with an expensive and difficult reworking of the algorithms. For some systems with certain properties there is an opportunity to profit from parallel computational power without re-implementation. In particular, Signal Processing Systems with several, mostly serial-connected components organized in pipelines. These show a great potential for acceleration via porting into the *MMER_Lab* framework. (Please note: We distinguish between *framework* (Fig. 1) and the *system* (Fig. 2) which runs in the framework.)

Frame-based video processing systems constitute an ideal example: Imagine a simple video analysis system consisting of four modules: a video source, a head localization and tracking module, a face-based person recognition module and a video display for demonstration. Such system can be parallelized by executing each module in its own thread. Thus, the person recognition module can perform its recognition task for frame number n on CPU-core 1, while the head tracking module already localizes the head(s) in frame m with $m > n$ using CPU-core 2.

However, parallel computing requires considerable skills in programming. *MMER_Lab*'s design is addressed to all levels of software developers including pure users. Module developers are offered a convenient way to get around implementation problems of multi-threaded systems, since the embedding of algorithms is performed using *MMER_Lab*'s API.

Thanks to the Graphical User Interface (GUI), casual users can instantly load existing modules, connect them to a complete system, and execute it via mouse clicks for ad-hoc demonstration or education purposes. Advanced users benefit from the scripting facilities and can perform numerous iterations and evaluations, e.g. of system parameter settings, in a batch processing style.

Our framework is developed cross platform for Linux and Windows operating systems and independent of the bit- architecture (32 or 64). This is achieved by purely gcc-compliant C/C++ programming code and usage of the Tcl/Tk [2] script interpreter. Furthermore, every kind of software library written in C and C++ can be applied for the implementation of modules for any signal processing system. Thus, our frame-

work provides an efficient opportunity to quickly transfer existing signal processing systems into multi-threaded applications with a full exploitation of the parallel clock speed of current and future multi-core architectures.

In the subsequent sections we explain and discuss the concepts of *MMER_Lab* for multi-threading, module structures and data flow, followed by application scenarios. Moreover we discuss our design decisions for *MMER_Lab* and give a conclusion.

2. STRUCTURE

The framework structure is based on the following three main concepts: A multi-threaded environment, a Tcl script interpreter with a Tk GUI, plus the underlying flexible and modular software architecture.

In the **Multi-Threaded Environment** (Fig. 1) modules are executed quasi parallel, depending on the number of available processors. Herein, the system modules (C++ objects) embed algorithms and work like stand-alone processors with a predefined interface. Each module core runs as a thread with an additional thread for the modules communication interface. Modules are connected by cables (C++ queue objects) for the transport of content (data, memory pointers, parameters and commands).

The synchronization between these components is implemented inside the cable and module classes using the *monitor pattern* [3]. This behaves as follows: Sources push data into a cable, while filters and sinks pull data from a cable. A source module *automatically* stops producing content, if the outgoing cable buffer is full, and continues when the cable buffer is depleted. A sink module *automatically* stops pulling content from the incoming cable buffer if it is empty and continues as long as the cable buffer is filled with content. A filter module is a combination of both behaviors. Multiplexers inside the modules interface are used for data dispatching with *round robin* or *priority based* methods.

The second important concept is the embedded **Tcl interpreter** [2] with a **Tk GUI** on top of the architecture (Fig. 1). The interpreter can create unique instances of modules and cables with a set of procedures to manipulate them (e.g. to connect them together). This provides an user interface for communication with the multi-threaded environment and its objects. By using the scripting-language capabilities of Tcl, we define a *high level framework language* which includes a large set of control command wrappers, helper functions and Tk procedures for GUI control. This approach has several advantages compared to using an API: The Tcl/Tk language itself can be used to perform initializations, interface I/O, algorithmic tasks *besides* the computations running in the the multi-threaded environment. This helps the user to focus on the main problem while he utilizes the scripting language do high- and low-level algorithmic tasks. Examples are sorting, parsing, file handling, computing numeric parameters for system/module setups, managing socket ports for distributed cluster processing or creating pipes to processes. The multi-platform capabilities of the language are also used to provide a convenient interface for the user on multiple operating systems (e.g. different file handling on Windows and UNIX platforms). The script interpreter is extremely helpful to configure module/cable setups for simplified loading and starting of frequently used systems. No recompilation or restart is needed for setup modifications or new different setups. During demonstrations, practical courses, and lectures, this capability turned out to be very valuable. Additionally, a great variety of freely available Tcl/Tk packages can be used by the framework (e.g. OpenGL Widgets, Audio Processing Toolkit, Secure Socket Layer). One important feature is the efficiency of plugging module packages into the system at *runtime*. Memory and performance can be saved, because the module code (C/C++) is stored in a set of shared libraries which are loaded into memory on demand. *MMER_Lab* is per se started with no functionality except for instantiating and controlling cables and modules. As mentioned above the front-end to the framework is implemented as a Tk GUI. However a GUI is *not required* to run the framework. It can be run in a batch mode only. This feature is useful to gain utmost performance or hide the framework interface if desired.

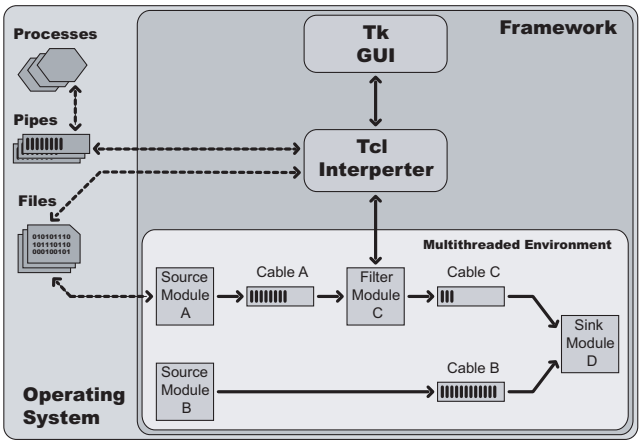

Fig. 1: Schematic view of the framework concept

The last core concept is a flexible **Software Architecture** which is explained in detail within the following section. Based on this architecture, the software developer is able to embed almost every existing library written in C/C++ into the modules computation core. Thus, useless and laborious re-implementation of standard functionality becomes dispensable.

3. SOFTWARE ARCHITECTURE

The underlying software architecture of *MMER_Lab* consequently follows the concepts of procedural and data abstrac-

tion [4]. Four fixed layers of abstraction have been established to provide access from low level functions (libraries) up to high level procedures (modules).

The lowest system layer (**libraries**) contains all basic *external* code libraries used in the framework and in most of our systems. This includes standard libraries like the C++ stdlib, Xlib, OpenCV, OpenGL and Boost or ATLAS, C-BLAS and GSL, high performance and high precision mathematical libraries compiled optimized for the dedicated processor architecture in Fortran, assembler and C.

The mid level system layer (**core functions**) is used to implement high level procedures as well as core framework functions by wrapping the basic libraries low level functionality. In this layer where abstraction techniques have been applied utmost. Here it is possible to adapt the function calls of the layer below to a unified notation of the architecture. Due to this concise abstraction interface it *is possible to easily exchange an external library below without adapting the layers above* the core functions.

The high- to midlevel (**module functions**) layer provides abstraction for base classes and functions to be called inside modules as well as the *base class hierarchy* of modules, cables, and the framework itself. These are implemented in this layer using C++ with the advantages of object orientation. Wrappers of framework procedures to Tcl/Tk are also located here. Due to the object oriented structures in C++ and the high abstraction level, other developers can easily re-use the basic behavior of their modules as well as basic algorithmic patterns. Experienced developers can change the multithreading and data dispatching behavior on this level.

The top layer (**modules**) consists of high level module implementations. The modules are derived from the base classes of the layer below. This is the primary layer for the researcher and developer who wants to embed his algorithms into *MMER_Lab*. Here we define the data and parameter interface to the module as well as an interface to interpreter and GUI, which is part of a module package. Finally the C/C++ code is stored in shared libraries (.dll/.so), organized as a package which can be loaded on demand at runtime.

4. DESIGN DECISIONS, APPLICATIONS, EVALUATION

Modularity, cross platform usability, and performance issues led to several crucial decisions, which have been made in the design phase of *MMER_Lab*. Some of the few existing software systems have been examined with respect to the three mentioned requirements.

Fermus [5] and Smartflow [6] for example are process-based systems for network distributed computing utilizing Inter Process Communication (IPC) techniques like socket communication and shared memory. This systems, designed to work on networks, support modularity and the usage of multiple processors very well. However, we will show that context switching on standalone computers using multi-tasking decreases the performance compared to multi-threaded architectures. This is because multi-tasking systems operate within their own virtual address space. Processes are protected by the operating system from interference by other processes. A user process can not communicate with another process unless it makes use of IPC mechanisms provided by the operating system. Passing data between computers comes with natural network latencies as well, which can affect real time high bandwidth computations such as video processing. The main advantage of [5] and [6] is the enormous computing power of a cluster system versus the network latency and intercommunication time.

Fig. 2: Training a SVM with AAM Parameter Data

A multi-threaded implementation on a multi-core computer addresses this flaw. Multiple processors are used to distribute the computing tasks and the hardware architecture provides a high bandwidth for exchanging data between the processors. The context switch of a threaded system is lightweight, since the threads all run in a single process and share the same address space. *MMER_Lab* was chosen to be implemented in this way providing performance and using an API for modularity and a carefree application of multi-threading. If desired, the framework can be extended to work in a distributed network environment by using a middleware as Smartflow. The multi-threaded approach also supports the tight integration of graphic processors into the framework. With *MMER_GPU* and *MMER_Lab* GPUs and CPUs can perform different computations in parallel. (gpu.mmer-systems.eu)

To demonstrate the capabilities of the framework, we instantiate a video processing and pattern recognition application. It performs the training of a Support Vector Machine (SVM) classifier with feature vectors provided by an Active Appearance Model (AAM) for the automatic recognition of four facial expressions (neutral, smile, frown, scream) based on a training set of annotated example images (see Fig. 2). An 'image source' module is directly connected to a 'head and eye localization' module. The output of the localizer provides a set of four signals (image frame, head-,left-,right-eye-region). These dataflows are connected to the AAM module for initialization of its face analysis. The output of the AAM module contains the extracted facial feature vector. This vector data is connected with the first port of the 'SVM' module. The second port is used for the class identifier of the corresponding facial expression. To begin the SVM training process the system requires two data items: filenames of the training images for the image source (at the head of the chain) and their corresponding class assignment for the SVM module (at the tail of the chain). A script parses the image-

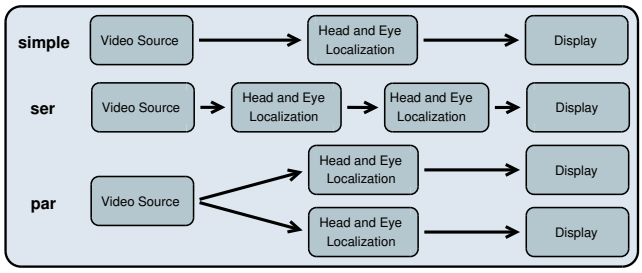

Fig. 3: Three Performance Evaluation Setups

filenames including the class information and feeds both modules with data at the same time. Here the synchronization abilities become apparent: The SVM cable containing the class information buffers the data until the module gets a facial feature vector from the AAM. Due to the fact that the SVM-module is built to pull exactly one instance from each of the input cables, it is ensured that the class identifier matches with the AAM feature vector of the correct image file.

For evaluation purposes we use a simpler application setup. Three different types of modules are combined to form three different video processing chains as seen in Fig. 3. We want to measure the performance using two different machine setups and three different framework types ('MMER', 'SF', 'direct') as explained below Table 1. To estimate the performance we measure the average framerate at the end of the chains. A higher framerate (more images are processed per second) indicates a higher performance.

The results of the evaluation can be seen in Table 1. The performance of both frameworks compared to the 'direct' implementation drops on a single core CPU system. This is a reasonable result because of the framework overheads (Threading, IPC) in both cases. *However one can see that the performance on MMER_Lab increases for the 'simple' and 'ser' video processing systems for dual core CPUs*. For the 'ser' case we have an increase of one third compared to a direct C implementation. In comparison with Smartflow, *MMER_Lab* achieves a slightly higher performance for every system in the evaluation. This result was expected due to the usage of IPC mechanisms for data transport in Smartflow. Smartflow is designed to work in a distributed network environment, so as default data is copied completely instead of passing pointers. However it is one of the few frameworks available at the moment which could be used for an evaluation with *MMER_Lab*.

5. CONCLUSION

When developers strive to exploit the parallel computational power of modern hardware, it is mandatory to separate systems and its components into multiple threads. Since threaded implementation constitutes an complex field in software development and requires considerable experience in programming, we built up *MMER_Lab* as framework to provide a multi-threading environment with the multi-threading and all its difficulties encapsulated behind interfaces. Due to the module concept of *MMER_Lab* with the necessity to define interfaces, the re-usability of modules is ensured and the system development process within research teams is constructively supported by our framework. *MMER_Lab* is especially appropriate for systems with high data flow between components, since it allows the handover of memory pointers on whatever kind of data structure. *MMER_Lab* is in our daily application for development, evaluation and demonstration of systems from the areas of image/video processing and pattern recognition. This validates, that the actualized concepts of *MMER_Lab* constitute a practical exemplar for generic and flexible environments to identify new design paradigms and to be prepared for the highly parallel hardware architectures of the near future.

System	Single Core			Dual Core		
	simple	ser	par	simple	ser	par
Direct	100%	100%	100%	100%	100%	100%
MMER	94.9%	93.9%	98.7%	105.3%	133.9%	96.7%
SF	92.0%	88.1%	93.6%	94.6%	130.9%	93.3%

Table 1: Performance Evaluation Results

MMER_Lab framework ('MMER') (one process, 3-5 threads) compared with the NIST Smartflow ('SF') system (3-5 processes) and a 'direct' implementation in OpenCV and C (only one process). The 'direct' implementation serves as a reference and is supposed to have always 100% of performance. Two different computer types have been used: a single core AMD 3700+, 2GB RAM computer with Linux Fedora Core 5 (FC5) and a dual core Intel Pentium4 3.0 GHz, 1GB RAM and FC5.

6. REFERENCES

[1] Intel Corporation, "Intel(R) software insight - multi-core capability," www.intel.com/cd/software/main/asmona/eng/285893.htm, 2005.

[2] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addision-Wesley Professional Computing Series, 1997.

[3] D. Schmidt, M. Stal, H. Rohnert, and F. Bushmann, *Pattern Oriented Software Architecture*, vol. 2, pp. 343–355, John Wiley and Sons, 2000.

[4] H. Abelson, G. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs (Second edition)*, The MIT Press, 1996.

[5] G. McGlaun, M. Lang, and G. Rigoll, "Development of a generic multimodal framework for handling error patterns during HMI," in *Proc. of SCI*, 2004, vol. I.

[6] M. Michel, V. Stanford, and O. Galibert, "Network transfer of control data: An application of the NIST smart data flow," *J. of Systemics, Cybernetics and Informatics*, vol. 2(6), 2005.