A WAIT-FREE REALTIME SYSTEM FOR OPTIMAL DISTRIBUTION OF VISION TASKS ON MULTICORE ARCHITECTURES

Thomas Müller, Pujan Ziaie and Alois Knoll
Robotics and Embedded Systems Group, Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany

Keywords: Robot Vision, Multithreaded Realtime System, Asynchronous Data Management, Interpretation-Based Preselection, Optimal-Backoff Scheduling.

Abstract: As multicore PCs begin to get the standard, it becomes increasingly important to utilize these resources. Thus we present a multithreaded realtime vision system, which distributes tasks to given resources on a single off-the-shelf multicore PC, applying an optimal-backoff scheduling strategy. Making use of an asynchronous data management mechanism, the system also shows non-blocking and wait-free behaviour, while data access itself is randomized, but weighted. Furthermore, we introduce the top-down concept of *Interpretation-Based Preselection* in order to enhance data retrieval and a tracking based data storage optimization.
On the performance side we prove that functional decomposition and discrete data partitioning result in an almost linear speed-up due to excellent load balancing with concurrent function- and data-domain parallelization.

1 INTRODUCTION

The multicore integration of off-the-shelf PCs is clearly observable with recent hardware development. Correlated to this, algorithms have to be developed that exploit parallel resources and generate the expected proportional speed-up with the number of cores. A computer vision (CV) system is a perfect prove of the algorithmic concept we present in this paper, because it requires high computational effort and realtime performance. The vision system is part of the JAST ("*J*oint *A*ction *S*cience and *T*echnology") human-robot dialog system. The overall goal of the JAST project is to investigate the cognitive and communicative aspects of jointly-acting agents, both human and artificial (Rickert et al., 2007).

Vision processing in the JAST system (Figure 1) is performed on the output of a single camera, which is installed directly above the table looking downward to take images of the scene. The camera provides an image stream of 7 frames per second at a resolution of 1024 × 768 pixels. The output of the vision process (recognized objects, gestures, and parts of the robot) has to be sent to a multimodal fusion component, where it is combined with spoken input from the user to produce combined hypotheses representing the user's requests.

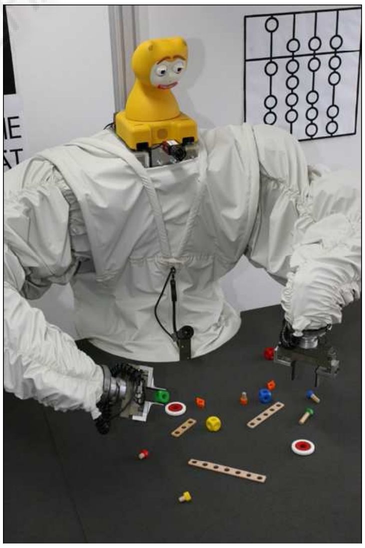

Figure 1: The JAST human-robot interaction system.

According to our research field of interest, the vision system is required to publish object, gesture, and robot recognition results simultaneously and in realtime, although continuous realtime result computation is not feasible. Therefore the JAST vision

Müller T., Ziaie P. and Knoll A. (2008).
A WAIT-FREE REALTIME SYSTEM FOR OPTIMAL DISTRIBUTION OF VISION TASKS ON MULTICORE ARCHITECTURES.
In *Proceedings of the Fifth International Conference on Informatics in Control, Automation and Robotics - RA*, pages 301-306
DOI: 10.5220/0001505403010306

setup is well suited for investigations on parallelization techniques and data flow coordination.We propose a multithreaded vision system based on a high level of abstraction from hardware, operating system, and even lower level vision tasks like morphological operations.This minimizes the overhead for communicational tasks, as the amount of data transferred decreases in an abstract representation. Furthermore, the scalability of the system with integration of multiple cores can be examined soundly by connecting different machines to the JAST system, each running a copy of the vision system (details in Section 4).

2 PARALLEL COMPUTATION

On an abstract level two major parallelization scenarios may be identified: distribution of processing tasks on multiple machines on one side and distribution of tasks on a single machine with multiple processors and / or cores on the other.

Many approaches employing the distributed scenario have been proposed, see (Choudhary and Patel, 1990) for an overview regarding CV or (Wallace et al., 1998) for a concrete implementation. However, with recent development in integration of multiple cores the latter scenario also becomes more relevant. Thus there is increasing demand for algorithms fully exploiting parallel resources on a single PC. This is especially the case, where computational power easily reaches the limits – e.g. in computer vision.

2.1 Communication

In parallel environments one can generally apply either synchronous or asynchronous communication strategies for data exchange between processes or threads. Though being robust, due to its blocking nature a synchronous approach can cause problems especially for realtime systems where immediate responses have to be guaranteed. For this case asynchronous *non-blocking* communication mechanisms (ACM) have been proposed. With ACMs information is dropped when capacities exceed – which is acceptable as long as the system does not block. Non-blocking algorithms can be distinguished into being *lock-free* and *wait-free* (Sundell and Tsigas, 2003). Lock-free implementations guarantee at least one process to continue at any time (with the risk of starvation). Wait-free implementations avoid starvation as they guarantee completion of a task in a limited number of steps (Herlihy, 1991).

According to (Simpson, 2003), ACMs can be classified based on the destructiveness of data access. The classification of ACM protocols by (Yakovlev et al., 2001) distinguishes data access with respect to their overwriting and re-reading permission. One can find manifold implementations of ACMs regarding each of these classification schemes. Some common implementations, e.g. from (Sundell and Tsigas, 2003) use lock-free priority queues or employ FIFO-buffers (Matsuda et al., 2004).

2.2 Parallelization Techniques

According to (Culler et al., 1999) we have to distinguish parallelization techniques by means of *data-domain* or *function-domain*. With function-domain parallelization the overall computation process is divided into stages and each thread works on a separate stage. In contrast to this, with data-domain parallelization data is partitioned and each partition requires the same computation performed by equally designed threads (Chen et al., 2007). This distinction may be correct and worthy for low level vision tasks like edge detection, but this paper will show, that on a higher level a carefully modeled CV system does not require this distinction. Moreover a combined approach can be derived and, on the basis of an asynchronous data management, a system implementing both aspects can perform very well in practice.

Aiming this goal, we first have to deliberately design anchor points for distributed computation. Also, the level of abstraction considering computational tasks matters in terms of parallelization. In order to avoid unnecessary overhead regarding communication and take full advantage of the multicore environment, we decided to model concurrent computation on a high level of abstraction. Therefore, we do not intend to parallelize primitive control-structures – like *for*-loops – specific to a programming language. Instead we try to identify major and subsequently minor tasks of computation (see Fiture 2).

For function-domain parallelization we assume, that the division into well-defined functional submodules is feasible. In the processing layer of the proposed CV system this is obviously the case, as we can identify three major functional stages: *Preprocessing*, *Analysis and Interpretation* and *Postprocessing*. Further refinement divides these stages into subtasks. Modules implementing a task independently pick a data partition (also called data item below), analyze it and write it back. In case new items are created within the analysis, these are also stored in the corresponding data management queue (see Section 3).

As the recognition process is decomposable in the function-domain, we now have to achieve data-domain parallelization in order to prove our claim.

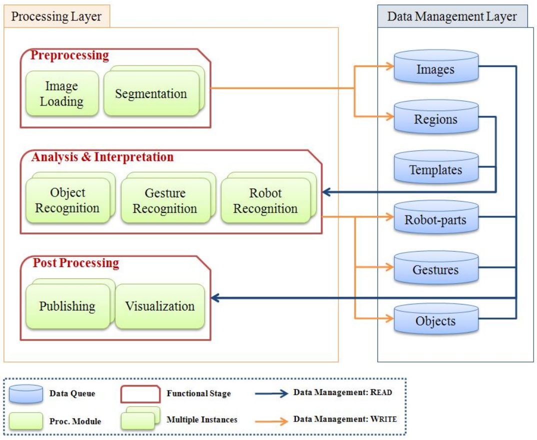

Figure 2: Architectural overview.

Hence we have to specify the functional tasks according to the need of multiple instantiation of the processing modules. We therefore derive the following approach from the non-blocking paradigm of ACMs: as we want to publish cyclicly in realtime, we rather publish incomplete analysis results of a scene than waiting for a complete analysis that would block the system meanwhile. This allows multiple concurrent module instances for the analysis of data items as long as the data management is implemented threadsafe (see Section 3). According to (Chen et al., 2007) we are thus able to implement data-domain parallelization, which is the second part of our claim.

2.3 Scheduling

There is one catch in such an implementation of the ACM: we risk that a module requests certain data from the data management, which is not available at the moment. In this case the data management delivers a NULL-data item, so modules have to deal with these items as well.

Therefore we propose an algorithm which, whenever a NULL-item is received, tries to suspend module instances for an optimal amount of time, until a correct data item is expected to be delivered again. An incremental back-off time $b(c)$ may be calculated as follows:

$$b(c) = \min\left(c \cdot i, \left(\frac{a \cdot j}{n}\right)\right) \quad (1)$$

In (1) the parameter c denotes the counter for the number of tries since the last correct data item has been received by the module, i denotes the predefined back-off increment in milliseconds, a is the maximum age of a data item until it is deleted, j the number of module instances operating on the same task and n the current number of items matching the request. If a NULL-data item is retrieved, c is incremented and the module is immediately suspended for a time $b(c)$ again. In case a correct item could be delivered, c is reset to 0 and the item is processed.

The back-off strategy tries to optimally calculate suspension periods for instances not needed at the moment, but at the same time to provide an instance whenever needed. The first argument of *min* calculates an incremental amount of time for the module instance to sleep and the second argument represents the expected mean time until the next correct data item can be delivered. This value is then used as the maximum amount of time to suspend a module instance.

3 DATA MANAGEMENT

Implementing an adequate data access strategy for concurrent requests is crucial for the proposed system. The strategy has to ensure integrity and consistency of data and as well provide error management policies. One also has to consider priorization whenever a module requests to write while another simultaneously wants to read data from or write data to the storage. Another important point is the deletion of data items when they expire.

Considering modularity, we organize data access in a data management layer (right part of Figure 2). A natural approach for the implementation is based on the *Singleton* design pattern (Gamma et al., 1998). Singleton implementations only provide a single instance of an object to the overall system, so in our case any request from an analysis module must call the single instance of the data management (DM). Here, derived from common standards (Message Passing Interface Forum, 1995), data items are managed in limited-size priority-queues.

Error handling in the DM layer can be implemented straight forward, as the layer simply delivers NULL-data items whenever an erroneous request was received, a queue was empty or no suitable data item could be found. The error handling approach utilizing NULL-data items is wait-free, because it completes in a limited number of steps.

Organizing the single instance in a threadsafe manner concerning read and write accesses ensures integrity and consistency. In order to achieve this, the DM module is organized as a bundle of queues, each queue for a different type of data item (see Figure 2).

3.1 Data Access

Threadsafe concurrent data access is realized by encapsulating synchronization. Concerning ACMs, the

CV system proposed here implements a *Pool*-ACM in either classification scheme mentioned in Section 2.1. Regarding the Simpson classification, as we do have non-destructive read operations, but write operations include deletion of items, and respectively regarding the Yakovlev classification, as we allow overwriting in a write operation and do not delete items when reading them from the storage.

Concretely, an instance of a processing module sends a request for storage or retrieval of a data item of a certain kind by calling one of the DM operations provided to the processing layer:

```
write<Queue>(Item):void
read<Queue>():Item
```

The retrieval strategy selects a data item to deliver according to the evaluation of a stochastical function. The function is based on the asumption that a data item (re-)detected in the near past must be prioritized to one that last occured many cycles ago – as it may have already disappeared or removed. Since each item in a queue Q has a timestamp, we weigh the items $i \in Q$ according to their age $a_i = now - timestamp(i)$ such that the weight increases, the younger items are:

$$\forall i \in Q : w_i = 1 - \frac{a_i}{maxage} \tag{2}$$

A new queue of pointers to data items from the original queue is built afterwards. The new queue, on which the actual retrieval operation is performed, is filled with at least one pointer to each data item. In fact, according to the weight w_i of an item i, a number of duplicates d_i of each pointer is pushed to the queue:

$$\forall i \in Q : d_i = \frac{1}{\operatorname{argmin}_{j \in Q}(w_j)} \cdot w_i \tag{3}$$

Subsequently the random selection on the pointer queue is performed where more recent items are prioritized automatically as more pointers to the corresponding data-items exist.

3.2 Locking

Before applying the weight to the items of a queue, we have to exclude elements that match the precondition described below. As an item cannot be altered by two processing modules concurrently, we introduce a locking-mechanism for items. Nevertheless the “non-blocking” nature of data access can still be guaranteed due to the error handling approach described earlier. Before a data item is delivered to the processing layer, the state of the item is changed to *locked*. Locked items are not allowed to be delivered to any other instance and so are excluded from the weighting step. Releasing the lock is in responsibility of the module processing the item.

Another important problem to discuss is the behaviour of the system in case of concurrent WRITE or READ operations concering a specific queue. Concurrent READ operations are allowed at any time, but in case a WRITE operation is requested all retrieval requests and concurrent WRITE requests must be blocked meanwhile. Therefore the system has to implement a mechanism utilizing cascaded mutual exclusions.

Again a single operation may be blocked, but the overall system is not. If a mutex can not be aquired at the moment, in case of a READ operation a NULL-item is delivered and in case of a WRITE operation no operation is executed. This behaviour is conform to the definition of an asynchronous non-blocking algorithm, as it is wait-free.

3.3 Enhancements

In order to enhance performance of READ operations, we introduce the concept of *Interpretation-Based Preselection*. We assume that certain data items are not relevant for dedicated tasks. For example a gesture-recognition module could only be interested in a region, that enters the scene from the bottom (Ziaie et al., 2008) or a visualization module might only display objects from within the last 100ms, but skipping gestures totally.

In order to completely leave the relevance decision to the processing modules, we propose a mechanism evaluating a predicate, that is passed within the request. According to the predicate the exlusion step before weighting a queue's items is adapted: now not only locked, but also items that do not match the predicate are removed. Thus the search space for retrieval can be restricted, but the non-deterministic selection algorithm can still be applied. We now extend the trivial retrieval definition from Section 3.1 to the following:

```
read<Queue>(Predicate):Item
```

`Predicate` is a non-empty binary predicate that evaluates to `True` or `False` on each data item of the specified queue. Processing modules are allowed to use item attributes for the implementation of their own predicates. For sophisticated predicate designs some items provide state attributes for tracking or attributes indicating the status of the analysis (`analyzedBy<module>`). We call these attributes *Priority Attributes*.

An enhancement strategy for WRITE operations can also be implemented by our data management module. Considering that data items in a queue are timed, it is possible to *track* them from one cycle to the following. Therefore we define a compare-method that is applied automatically on a storage request. The method evaluates symbolic or meta attributes like classification, color, approximate position, number of points or width and height. Whenever the DM module receives a storage request for a formerly recognized item, only neccessary attributes are updated, all priority attributes (especially the unique id) instead are kept. For example, considering an item fixed and fully analyzed, the existing item just gets all non-priority attributes (such as the timestamp, position, etc.) updated, but the updated item is not marked for analysis again.

4 RESULTS AND CONCLUSIONS

For the evaluation of the system a dual core Intel © ,Pentium IV system and a quad core Intel © ,Xeon system were utilized. For comparison the hardware configurations using a sequential version of the system are also shown. We used a sample video with a resolution of 1024×768 and a duration of 30 seconds at a sampling rate of 7 frames per second. The results refer to the analysis without data maintenance enhancements and postprocessing switched off.

Performance results shown in the tables below are only approxmiate values due to high dependency on the scenes that have to be analyzed. The more objects exist, and the more complex objects get (in the JAST-project also object assemblys are to be analyzed) the longer the analysis takes. In case there are very few objects or the scene remains static, overall analysis is possibly performed in realtime. This would be contradicting on of our preconditions from Section 1, so for our evaluation video we feed the system with dynamic input data, like humans continuously moving objects on the table and the robot picking pieces.

In Table 1 the first column describes the hardware configuration, the second column shows the total system load and the third column, the (mean) LOAD RATIO, weighs the core with highest against the core with the lowest load.

Table 1: Total core utilization.

CONFIGURATION	CPU %	LOAD RATIO
Dual Core (seq.)	52.12	8.20
Dual Core (parallel)	84.13	1.03
Quad Core (seq.)	27.55	36.32
Quad Core (parallel)	51.63	1.07

The values shown in the table were computed from 5-10 averaged samples, each taken with `mpstat` over a period of five seconds. For example a mean ratio of 36.32 on the quad core is caused by an average load of 93.84% on the core with highest load compared to only 2.58% on the core with lowest load. The total utilization in this configuration clearly shows that de facto only one core is used for processing while the others remain idle. In contrast to this, one can see an almost optimal distribution in the parallel scenarios with a load ratio of around 1.0.

Table 2 shows the processing performance of the hardware configurations described above. Now the reason for the quad core parallel configuration only having a total load of 51.63 % becomes clear: there is simply nothing to do for the machine as the input video is only sampled at 7 frames per second.

Table 2: Performance of Processing.

CONFIGURATION	TIME	FREQUENCY
Dual Core (seq.)	228.7s	0.92fps
Dual Core (parallel)	30.0s	3.57fps
Quad Core (seq.)	196.6s	1.07fps
Quad Core (parallel)	30.0s	6.95fps

Processing with the sequential version of the system is slightly faster on the quad core compared to the dual core machine due to internal OpenCV parallelization and scheduling of the operating system. But still it is only capable of processing the video in ≥ 3 minutes. Regarding this, another important property of the asynchronous parallel version becomes clear: processing a 30s-video only takes 30 seconds. This can be achieved because of the non-blocking behaviour. In case computing power exceeds (see second row of Table 2) the asynchronous implementation drops frames, regions and objects from data queues in order to keep the system from blocking. We find that the system still reaches the desired realtime publishing frequency, but the results published are not complete. In fact we see, that it takes a few cycles until each region extracted from a frame is analyzed and results are present.

Normally this does not influence the result, as items can be tracked. But in case of quickly moving objects, it remains as a drawback, because the simple feature-based tracking method applied in the current system often fails to map these objects correctly in a sequence of frames. Consequently, the system assumes items having appeared and begins the analysis: the new items are locked (although they are just duplicates of existing ones) and computing power is wasted. This problem particularly occures for quick hand movements. The worst case would be a moving hand shortly occluding formerly recognized objects, as both the hand and the objects are probably lost and their regions need to be redetected and reanalyzed completely.

Figure 3 shows a performance estimation for the analysis frequency in two input scenes. As we expect, the results show, that parallelizing in the data domain produces almost linear performance gain with the number of processors. This can be achieved, because heavy computing is mainly done within the analysis and interpretation stage, where tasks can be distributed very well. In Figure 3 the measuring points for one core are inferred from performance of sequential version, as we have seen in Table 1 that only one core is used there.

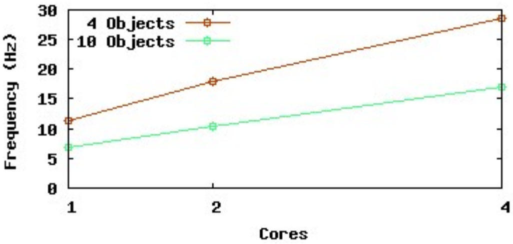

Figure 3: System performance with data domain parallelization.

Still to mention is that with the data management enhancement from Section 3.3 the performance even in a sequential version of the system improves up to 25 Hz as long as the extracted regions can be tracked. When the scene changes, computational effort is needed, so the performance decreases. Here the advantage of the multithreaded system becomes clear: due to function domain parallelization and nonblocking behaviour the system still publishes in realtime, although the results may be incomplete.

A further factor influencing the system performance is the system configuration. The vision system configuration can be customized via an XML file. Here one can specify the number of module instances. This corresponds to a priorization within the data domain: one could for example start a larger number of objectrecognition modules while on the other hand just starting one or two gesturerecognition modules. Due to the scheduling strategy of the operating system, the objectrecognition would be prioritized in this case.

ACKNOWLEDGEMENTS

This research was supported by the EU project JAST (FP6-003747-IP), http://www.euprojects-jast.net/.

REFERENCES

Chen, T. P., Budnikov, D., Hughes, C. J., and Chen, Y.-K. (2007). Computer vision on multi-core processors: Articulated body tracking. pages 1862–1865. Intel Corporation, IEEE ICME.

Choudhary, A. N. and Patel, J. H. (1990). *Parallel Architectures and Algorithms for Integrated Vision Systems*. Kluwer.

Culler, D. E., Singh, J. P., and Gupta, A. (1999). *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1998). *Design Patterns: Elements of Reusable Object-Orientated Software*. Addison-Wesley Professional Computing Series.

Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149.

Matsuda, M., Kudoh, T., Tazuka, H., and Ishikawa, Y. (2004). The design and implementation of an asynchronous communication mechanism for the mpi communication model. pages 13–22. IEEE ICCC.

Message Passing Interface Forum (1995). MPI, A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, Tennessee.

Rickert, M., Foster, M. E., Giuliani, M., By, T., Panin, G., and Knoll, A. (2007). Integrating language, vision, and action for human robot dialog systems. Proc. ICMI.

Simpson, H. R. (2003). Protocols for process interaction. volume 150, pages 157–182. IEE Proceedings on Computers and Digital Techniques.

Sundell, H. and Tsigas, P. (2003). Fast and lock-free concurrent priority queues for multi-thread systems. Int. Parallel and Distributed Proc. Symp.

Wallace, A. M., Michaelson, G. J., Scaife, N., and Austin, W. J. (1998). A dual source, parallel architecture for computer vision. *The Journal of Supercomputing*, 12(1-2):37–56.

Yakovlev, A., Xia, F., and Shang, D. (2001). Synthesis and implementation of a signal-type asynchronous data communication mechanism. pages 127–136. Int. Symp. on Advanced Research in Async. Circuits and Systems.

Ziaie, P., Müller, T., Foster, M. E., and Knoll, A. (2008). A naïve bayes classifier with distance weighting for hand-gesture recognition. CSICC.