Algorithmic Skeletons for Parallelization of Embedded Real-time Systems

Alexander Stegmeier, Martin Frieb, Ralf Jahr, and Theo Ungerer

Institute for Computer Science,
University of Augsburg, 86135 Germany
{alexander.stegmeier,martin.frieb,ungerer}@informatik.uni-augsburg.de}
info@ralf-jahr.de
http://www.informatik.uni-augsburg.de/en/chairs/sik/

Abstract. Since parallel code is difficult to understand, it is a challenging job to develop it. It is even harder for code applied in embedded real-time systems. To facilitate parallelization of code running on these systems, we introduce a new algorithmic skeleton library. It is characterized by a deterministic execution behavior and is built in a timing-analyzable way. Our library offers algorithmic skeletons for parallel execution of tasks, parallel data processing and pipelined processing of data. When parallelizing a legacy single-core application, e. g. by applying a pattern-supported parallelization approach, it may be employed to implement parallelization on code level. Furthermore, it can be utilized for enabling structured parallelism at development of a new real-time application for an embedded system.

Keywords: Algorithmic Skeletons, Parallelization, Parallel Design Patterns, Embedded Systems, Real-Time, Static WCET Analysis

1 Introduction

The multicore revolution [23] does not stop at embedded real-time systems. As in high-performance computing, parallelizing sequential legacy code is desired for embedded real-time systems, too. This can be achieved e. g. by applying the parallelization approach developed by Jahr et al. [15,14]. However, developing parallel code remains challenging since it is difficult to understand [21]. It is even harder when the resulting code has to be timing predictable for analysing it with a static WCET analysis tool like e. g. OTAWA [2]. Hence, parallelism should only be implemented in a structured way [16]. A comfortable way for this is to encapsulate all issues of parallelism in an algorithmic skeleton library. The concept of algorithmic skeletons was introduced by Murray Cole [7,6]. If a skeleton library is built in an analysable way, its parallelism introduced to an originally sequential program will not break its timing analysability.

This paper introduces a predictable skeleton library which can be utilized to parallelize legacy embedded real-time applications. Furthermore, it is applicable to introduce structured parallelism during development of applications from

scratch. In both scopes, our library leads to a timing-analysable parallel program. It has been developed and applied in the parMERASA project [24,25]. Our algorithmic skeleton library is open source and can be downloaded on GitHub[1].

An overview on existing algorithmic skeleton libraries is given in Section 2. Our algorithmic skeleton library is described in Section 3 and evaluated in Section 4. Finally, the paper is concluded accompanied by an outlook on future work in Section 5.

2 Related Work

There are already a lot of existing skeleton libraries. A detailed overview can be found in [12]. Many skeleton libraries have been developed for high-level programming languages, e. g. Muesli [5] for C++ or Skandium [18] for Java.

For embedded systems, we only consider algorithmic skeleton libraries for the execution language C. To our knowledge, there are around eight popular algorithmic skeleton frameworks to use with C: Eden [19] and HDC (Higher-order Divide and Conquer) [13] use variations of Haskell as coordination language to generate C code. Both allow nested skeletons. P^3L [1] utilizes a custom coordination language and nesting is limited to two levels. A subset of C with functional features to implement skeletons is applied by Skil [4]. The resulting code is transformed into regular C code; nesting is not possible.

Since the resulting code has to be checked each time after generation, the concept to generate code is difficult for timing-analysis. With a high effort it might be possible to do a generic analysis over all code which can be generated. However, it seems to be less challenging to build a new skeleton library which is timing predictable. Our skeleton library is built in a static way with no dynamically generated code. Hence, it is possible to do a single analysis to obtain its timing behavior. If our skeleton library is applied to parallelize a previously analysable sequential program, it will stay analysable afterwards.

SKELib [8] is provided as a library in C without dynamic generation of code. It is built for workstation cluster architectures. However, communication takes place over plain TCP/IP sockets, which is clearly not timing-analysable.

Another promising approach is eSkel [3], which is used in a static way. But, timing-analysis is complicated since there is a lot of dynamic memory allocation.

There are two algorithmic skeleton libraries which try to address real-time requirements: QUAFF [9] and SkiPPER [22]. However, they only focus on soft real-time requirements just like image processing, not on hard real-time requirements for embedded systems. Furthermore, SkiPPER is domain-specific and only provides limited nesting capabilities.

Almost all presented algorithmic skeleton libraries use MPI as distribution library. To our knowledge, there is no timing-analysable implementation of MPI. For our skeleton library, we assume communication over shared memory. It has been developed for the timing-analyzable parMERASA multi-core [24], where

[1] Website: `https://www.github.com/parmerasa-uau/tas/` Licence: GNU LGPL v3

several cores belong to a cluster and share the clusters' memory. The cores and clusters are connected over a network on chip (NoC) which works in the background.

3 Architectural Design

We designed our skeleton library for embedded real-time systems. Therefore, it does not use any code generation, but is coded as static library in C. The utilization of the skeletons is done by calling functions provided by the library.

In this section, we first (Section 3.1) give an overview about the types of parallelism which are supported by our skeleton library. Afterwards, we describe design decisions and implementation details in Section 3.2.

3.1 Supported types of parallelism

It is possible to describe structured parallelism with Parallel Design Patterns (PDPs), which are a textual description of best practice solutions [20]. We see PDPs as an abstract concept for skeletons. They describe in natural language how parallelism could work. Algorithmic skeletons are an implementation fulfilling the behavior described in the PDP.

Our goal is to build a time-predictable skeleton library for parallelization issues. Therefore, it is based on the parMERASA pattern catalogue [11] which contains timing-predictable PDPs. It contains patterns which have been found by analysing applications from automotive, avionics and construction machinery domains. The types of parallelism realized in our algorithmic skeleton library are described in the following:

1. **Task Parallelism**, which is also known as *farm* skeleton: A number of tasks is executed concurrently and the further execution of the program is suspended until they are all completed. The WCET is mainly defined by the longest WCET of each subtask. To be timing-analysable, the tasks have to be scheduled and mapped statically.
2. **Data Parallelism** is mostly referred as *map* skeleton or *SPMD (Single Program Multiple Data)*: This skeleton performs computations on a data structure, which can be decomposed into concurrently computable chunks. Therefore, the same algorithm can be applied simultaneously to several parts of the data structure. Since the same computations take place for different data, the WCET should be similar for all computed chunks.
3. **Pipeline**, or shortly named *pipe* skeleton: The executed computations on input data can be divided into several stages. After data has been processed in one stage, it is handed over to the next one. Afterwards, the finished stage can process the next set of data. Hence, the data is processed in a chain of producers and consumers. Ideally, the stages are load-balanced, i. e. their workloads obtain similar WCETs. To achieve this, stages might be joined or split or further skeletons may be applied within single stages. For the

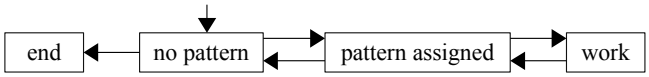


Fig. 1. State machine of worker threads

WCET analysis, each stage and the data exchange between stages have to be analyzed.

In combination with a systematic parallelization approach, the parMERASA pattern catalogue and our skeleton library may be used to parallelize a sequential program [15], which also works for hard real-time applications [14].

3.2 Design Decisions and Implementation Details

In the following, we assume a processor with several cores and shared memory as the platform utilized for running the skeletons. We further number the cores in an ascending order to facilitate the distinction between them. Thereby, we denote core0 as main core which invokes the skeletons for executing code in parallel.

Underlying implementation layer and state machine Our Skeletons base on an underlying implementation layer which is characterized by three different phases of execution. At first, the initialization of the skeletons takes place, followed by the phase responsible for executing user applications applying the skeletons. At last, finalization is needed to correctly shutdown the skeleton library. While initialization and finalization have to be done only by core0, in the execution phase all applied cores are involved.

During the *first phase* the synchronization idioms (e. g. several barriers) and threads for the skeleton instance are initialized and created. Therefore, the number of applied worker cores has to be determined and one thread is created and moved to a thread pool for each of them. All utilized synchronization idioms have to be implemented in a timing predictable way. How this can be done can be seen in [10].

In the *second phase* of the underlying implementation core0 executes the user application with integrated invocations of skeletons, while the threads of the workers run a state machine which determines their behavior. This state machine is displayed in figure 1, the transitions between the states are realized with barriers. The states are listed below:

no pattern After initialization, all threads are in this state. It represents a stand-by mode where the threads are waiting to get a specific skeleton assigned. As described in section 3.1, each kind of skeleton implements one particular pattern. By assigning a skeleton to a thread, it changes its state to *pattern assigned*[2]. There is also a transition to state *end* for finalization purposes.

[2] The assignment of a skeleton's workload takes place at runtime, but it has been statically defined at design time which worker threads get which workload assigned.

pattern assigned A thread in this state is assigned to a particular skeleton. It obtains its work to execute and waits for all other threads involved in this skeleton to get their work, too. Once all threads are ready, they move to state *work*. If the thread is released from the skeleton, it changes to state *no pattern*.

work Here, a thread executes its assigned work and afterwards waits for all other participating threads to finish their work. After the last thread has finished, all threads move back to state *pattern assigned*.

end In this state the thread shuts down the execution of the state machine. Thus, the execution of the thread is finished and it is ready for its termination.

The distinction of the states *pattern assigned* and *work* separates the assignment of workload (done by main thread) from the execution of this workload (done by workers). This design decision facilitates the timing analysis, because execution of the workload starts for all workers at the same time.

In the *phase of finalization* of the skeleton instance all threads are forced to move to the state *end* immediately after joining state *no pattern* the next time. Once all threads of the thread pool are in this state, they are terminated. Afterwards, the execution of the skeleton instance is finished.

Invocation of a skeleton The invocation of a skeleton is done by the main core (core0) of the implementation. It executes the user application and invokes the skeletons for the parallel parts of the execution. As the whole implementation, the invocation of a single skeleton is divided into three segments, where each is executed by a single function call of the skeleton library. These steps are initialization, execution and finalization and are illustrated in code example 1:

Code example 1 Initialization, execution and finalization call for a task parallelism skeleton with two workers.

```
tas_taskparallel_init(&task_parallelism, 2);
tas_taskparallel_execute(&task_parallelism);
tas_taskparallel_finalize(&task_parallelism);
```

During initialization, the main core notifies the threads it needs for execution of the particular skeleton. Thus, these threads move to state *pattern assigned*. Afterwards, the execution is started by handing over the work to the worker threads. Then, the execution of the workload takes place. Therefore, all participating threads (excluding the main core) move from *pattern assigned* to *work* and back to *pattern assigned*. If the overall work of the skeleton is split into more parallel parts than threads are available, the workers execute the workload in multiple rounds. Therefore, they change between *pattern assigned* and *work* several times. At the end of execution all workers are in state *pattern assigned*. The finalization step is responsible for releasing the applied threads. Therefore, it forces the workers to move to state *no pattern*. A manual for using the skeletons containing lots of examples is [17].

Implementation details Synchronization idioms are needed for the implementation of the skeletons. Namely, these are barriers and ticket locks. They have to be provided by the execution platform to enable the application of our skeleton library. While the barriers are mainly utilized for transitions between the states of the state machine, the locks are used to keep the metadata for the thread pool consistent. In each state (excluding *end*) a thread has to wait for a kind of synchronization event before changing to another state.

Code example 2 Declaration of data structures in global shared memory for a task parallelism skeleton.

```
//Functions to be called
SHARED_VARIABLE(memory0) tas_runnable_t tp_runnables[] = {
     (tas_runnable_t) runnable0,
     (tas_runnable_t) runnable1}

//Pointer to arguments for functions (NULL because not needed)
SHARED_VARIABLE(memory0) void * tp_args[4];

//Task Parallelism: list of ...
SHARED_VARIABLE(memory0) tas_taskparallel_t task_parallel = {
    tp_runnables, //... functions to be called
    tp_args,      //... arguments of functions
    2};           //... and the number of functions
```

At an invocation of a particular skeleton the work executed by the related threads must be handed over. This is done by providing a list of function names, each representing the work for one thread (runnable0 and runnable1 in code example 2). It is statically determined by the programmer which thread is assigned to a skeleton. This is done by defining a list including the ids of the participating threads. Thereby, the particular assigned work for each of these threads can also be figured out. The elements of the function list including the work are assigned to the specified threads by mapping the element order of the function list to an ascending order of the thread ids. This guarantees a deterministic behavior of the parallel execution and preserves its timing analysability.

The execution of assigned work is realized by invoking a function pointer which executes the function associated to the according thread. The concrete executed function for each thread can be obtained by observing the defined function and thread lists. Hence, the estimated WCET of the correct function can be inserted at the corresponding places of the analysis.

To enable the execution of a particular skeleton, the data structures describing the functions to be executed and their parameters must be located in shared memory. This includes the data which has to be handed over to the applied threads and the code to be executed in parallel. Therefore, specific data structures are provided which have to be initialized in shared memory and referenced by pointers. These pointers are timing-analyzable as they always point to the same

memory location of the data structures. The data structures always have valid values (function names and parameters).

4 Evaluation

In this section we show that employing the skeletons for parallelization is applicable to gain a speedup. Therefore, we focus on performance tests based on observed execution times (ETs). The results of a WCET analysis and the obtained overestimation for the skeletons will be presented in future work. The evaluation here mainly concentrates on analyzing the achieved parallelization overhead. We aim to investigate if the obtained overhead provides moderate length in time compared to the executed workload. A further objective is to show the scalability for large input sets.

4.1 Experimental setup

In the experiments we run a signal processing application from the scope of avionics. Algorithms like this can often be found in embedded real-time systems for processing data which has been captured with sensors. The structure of the application is displayed in figure 2. It takes two sets of matrices as input (subsequently named a and B). They are characterized by the same size and number of included matrices and used for calculation of one output matrix. The program is executed iteratively, processing ongoing input sets. The first step of the computation is a Fast Fourier Transformation of each matrix of a. The result (named A) is multiplied with B element by element. Afterwards, the calculated set of matrices C is summarized to one matrix D by computing the sum of element by element. At last the output matrix d is generated by calculating the inverse Fast Fourier Transformation.

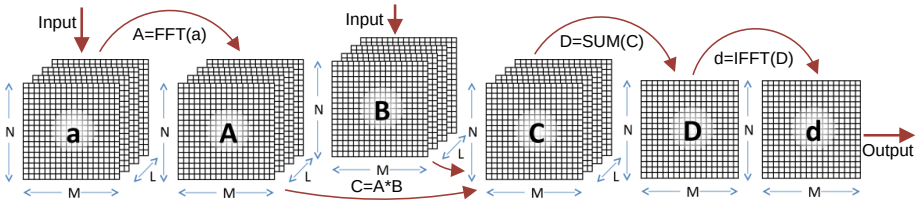


Fig. 2. Implemented application for the evaluation

Since the complete calculation is divided into several computation steps, the pipeline skeleton fits best. The pipeline consists of five stages. The first one is utilized for generating input values. Thus, it does not belong to the application itself, but is needed for simulation purposes. The other four stages implement the calculation of the output matrix and are listed below.

1. Fast Fourier Transformation (FFT) of a (result: A)
2. Multiplication of A and B (result: C)
3. Summarize all matrices of C (result: D)
4. Inverse Fast Fourier Transformation (IFFT) of D (result: d)

However, the single stages of the pipeline are not balanced. Calculating the FFT for a set of matrices takes a lot more time than the other calculation steps. Thus, we further parallelize this particular step by utilizing the data parallelism skeleton. The set of matrices a is split into a number of smaller sets. Their sizes are dependent on the number of cores applied for calculating this stage. After all cores calculated their subset of A, the next pipeline stage can be executed using this particular matrix set A.

Three different implementations of the application are run for evaluation:

- a sequential version without skeletons
- a parallel version only employing the pipeline skeleton
- a version which utilizes pipeline parallelism and data parallelism

The tests are run on the parMERASA simulator, which was developed in the parMERASA project [24,25]. As mentioned in section 2, this simulator provides a predictable many-core platform. The cores are divided into clusters, which are connected by a NoC. In addition to the cores, each cluster comprises a memory which is shared by the cores of the cluster[3]. This memory contains a private area for each core and an area utilized as global memory space for all cores of the cluster. The emulated cores support the PowerPC 750 ISA with little variations. In this paper, we have configured the simulator to provide eight cores in one cluster. Hence, all cores are connected to the same shared memory with a latency of 40 cycles. Memory accesses are organized hierarchically by utilizing a private first level cache for each core. Thereby, we assume a perfect instruction cache and a data cache with LRU replacement policy.

We apply two input sets of different size to investigate the influence of differing workloads, where each one consists of two sets of matrices. Each matrix set of the small input set consists of four matrices. Each of them has 16 rows and 16 columns. The large input set consists of 8 matrices of 32 elements in rows and columns per set. A complete input set is generated for each new execution step of the pipeline, which is done for 25 iterations. Hence, in the experiments the pipeline runs for 29 iterations and for 21 iterations all stages execute workload. This behavior is caused by the starting and finalization delays of the pipeline, where only some of the five stages have work.

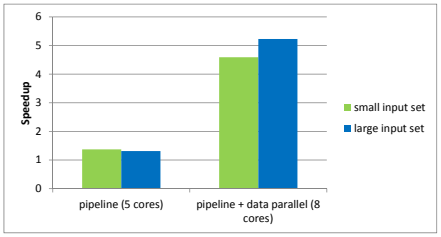
4.2 Performance Evaluation

Results of the execution on the parMERASA simulator can be seen in table 1. The columns show the three different versions and the numbers for the small (16x16x4) and the large (32x32x8) input sets.

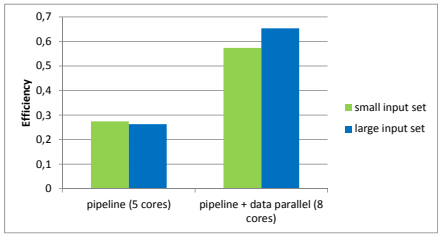
[3] For details see parMERASA Deliverable 5.3 on `www.parmerasa.eu`.

Table 1. Evaluation results

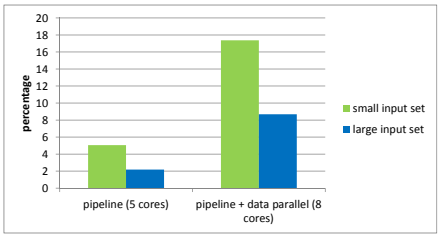
	sequential version		only pipeline		pipeline & data parallel	
	small	large	small	large	small	large
ET	66 328 306	578 788 292	48 326 199	440 411 595	14 448 736	110 708 495
overhead	-	-	2 445 611	9 615 521	2 510 436	9 616 293



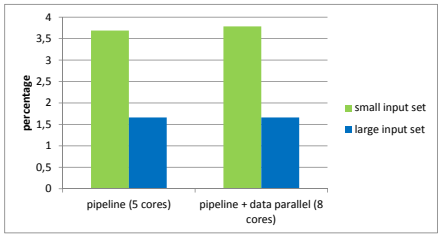
(a) The gained speedups of the parallel versions



(b) The gained efficiency of the parallel versions



(c) The overhead ET divided through the overall ET of the parallel version (ROP)



(d) The overhead ET divided through the overall ET of the sequential version (ROS)

Fig. 3. The results of the obtained from measured ETs of the experiments utilizing the parMERASA simulator

Independently from the applied input sets the investigation of the ETs shows a small speedup for the implementation only utilizing the pipeline and a significant higher one for additionally applying the data parallelism (see figure 3(a)). The small speedup of the first implementation is (in addition to a lower number of employed threads) mainly caused by an unbalanced pipeline. With the large (small) input set, the stage executing the FFT needs 72.52% (67.34%) of the overall time to execute its workload for each iteration. At the same run, the percentages of the other stages reach from 8.45% (7.16%) to 11.46% (17.58%). This disparate distribution of workload can be overcome by further parallelizing the unbalanced stage, which is done in the implementation additionally utilizing the data parallelism. In this implementation the pipeline obtains an improved balance with proportions of 38.25% (34.67%) to 16.94% (14.32%) for the large (small) input set. As can be seen in figures 3(a) and 3(b), a considerably higher speedup and efficiency (gained speedup per utilized core) is reached in this implementation.

In table 1 the obtained parallelization overheads are illustrated. Although one parallel implementation employs an additional data parallelism skeleton, both parallel executed versions exhibit nearly no difference in measured parallelization overheads. The reason for producing only low additional overhead for the data parallelization is, among other things, the starting and finalization delay of the pipeline.

The *relative ET overhead based on parallel* (ROP) is the overhead ET of a particular parallel version divided through the overall ET of the same version (compare table 1). In figure 3(c), it can be seen that the implementation with the additional data parallelization exhibits a higher ROP than the application only utilizing the pipeline. As the workload stays the same, this is caused by a smaller overall ET of the application. The *relative ET overhead based on sequential* (ROS) is the overhead ET divided through the overall ET of the sequential version (compare table 1). Since the ROS is similar for both parallel implementations (compare figure 3(d)), it confirms that the workload stays the same. As the ROS does not exceed 4%, the parallelization overhead for applying the skeletons does not influence the execution time of the parallel execution drastically.

A closer look at the executions with different sized input sets exhibits an enormous difference in the overall ET of the implementations. The ETs are about ten times higher for the large input set than for the small one. Furthermore, it seems that there is an inconsistency in the calculated speedups. While the speedup of the pure pipeline implementation decreases for a larger input set, it increases for the pipeline and data parallel implementation. This behavior can be explained by the different workloads of the pipeline stages for the various input sets. The execution time of the large pipeline stage in the pure pipeline implementation increases faster than the ETs of the other stages for a larger input set. Thus, the pipeline obtains a worse balance for larger input sets. In contrary, the ETs for the pipeline applying data parallelism increase similar for all stages. Hence, the pipeline stays balanced (respectively gets even more balanced) and the speedup stays the same or increases.

A further reason for increased speedup when applying a larger input set is the reduction of ROS and ROP. Though, the absolute execution time for parallelization overhead increases with an enlarged input set, the ROS and ROP decrease. This loss in proportion in execution time is caused by a faster increase of ETs for calculation than for ETs of raised overhead.

5 Conclusion and Outlook

We presented the design and implementation principles of a new algorithmic skeleton library which can be utilized for the parallelization of legacy software running on embedded real-time systems. Moreover, it is applicable to enable structured parallelism at the development of new applications for an embedded real-time system. Our focus was to keep the skeleton library simple and timing-analyzable. The skeletons are designed for multi-core platforms providing shared

memory for all cores. The supported types of parallelism are *Task Parallelism*, *Data Parallelism* and *Pipelining*.

The evaluation shows that our skeleton library is applicable for parallelization of sequential code to gain speedup. The analysis of time needed for execution of parallelization overhead obtains a moderate proportion of the overall execution time of the application. Furthermore, the percentage of overhead execution remains stable for increased input sets.

As future work we propose to make a detailed timing analysis of our algorithmic skeleton library. We want to determine static WCETs with OTAWA [2]. With the results of the analysis, it should be possible to optimize the implementation of the skeletons. Thereby, we hope to reduce the obtained overestimation of the WCETs. Therefore, our algorithmic skeleton library could also be utilized to meet tight timing constraints.

Acknowledgments. The research leading to these results has received funding from the EU 7th Framework Programme under grant agreement no. 287519.

References

1. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: A structured high-level parallel language, and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.
2. C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *LNCS*, pages 35–46. Springer Berlin Heidelberg, 2011.
3. A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In J. Cunha and P. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *LNCS*, pages 761–770. Springer Berlin Heidelberg, 2005.
4. G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *5th International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society Press, 1996.
5. P. Ciechanowicz, M. Poldner, and H. Kuchen. The Münster Skeleton Library Muesli: A comprehensive overview. ERCIS Working Papers 7, Westfälsche Wilhelms-Universität Münster (WWU) - ERCIS, 2009.
6. M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389 – 406, 2004.
7. M. I. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. PhD thesis, University of Edinburgh, 1988. AAID-85022.
8. M. Danelutto and M. Stigliani. SKElib: Parallel Programming with Skeletons in C. In A. Bode, T. Ludwig, W. Karl, and R. Wismller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *LNCS*, pages 1175–1184. Springer Berlin Heidelberg, 2000.
9. J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. QUAFF: Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32(7):604–615, Sept. 2006.
10. M. Gerdes. *Timing Analysable Synchronisation Techniques for Parallel Programs on Embedded Multi-Cores*. PhD thesis, University of Augsburg, 2013.

11. M. Gerdes, R. Jahr, and T. Ungerer. parMERASA Pattern Catalogue. Timing Predictable Parallel Design Patterns. Technical Report 2013-11, Department of Computer Science, University of Augsburg, Augsburg, Germany, 2013.
12. H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software – Practice and Experience*, 40(12):1135–1160, Nov. 2010.
13. C. A. Herrmann and C. Lengauer. $\mathcal{HDC}$: A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(02n03):239–250, 2000.
14. R. Jahr, M. Gerdes, and T. Ungerer. On efficient and effective model-based parallelization of hard real-time applications. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IX*, pages 50–59, Schloss Dagstuhl, April 2013. fortiss GmbH, Munich.
15. R. Jahr, M. Gerdes, and T. Ungerer. A pattern-supported parallelization approach. In *International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 53–62, New York, NY, USA, 2013. ACM.
16. R. Jahr, M. Gerdes, T. Ungerer, H. Ozaktas, C. Rochange, and P. G. Zaykov. Effects of Structured Parallelism by Parallel Design Patterns on Embedded Hard Real-time Systems. In *20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Chongqing, China, August 2014.
17. R. Jahr, A. Stegmeier, R. Kiefhaber, M. Frieb, and T. Ungerer. User Manual for the Optimization and WCET Analysis of Software with Timing Analyzable Algorithmic Skeletons. Technical Report 2014-05, University of Augsburg, 2014.
18. M. Leyton and J. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 289–296, Feb 2010.
19. R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.
20. T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.
21. A. Meade, J. Buckley, and J. J. Collins. Challenges of evolving sequential to parallel code: An exploratory review. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 1–5, New York, NY, USA, 2011. ACM.
22. J. Sérot and D. Ginhac. Skeletons for parallel image processing: an overview of the SKIPPER project. *Parallel Computing*, 28(12):1685 – 1708, 2002.
23. H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.
24. T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. Zaykov, Z. Petrov, B. Boddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Casse, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quinones, M. Panic, J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability. In *2013 Euromicro Conference on Digital System Design (DSD)*, pages 363–370, Sept 2013.
25. T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, A. Stegmeier, M. Frieb, J. Fernandes, P. Zaykov, Z. Petrov, B. Boddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Casse, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quinones, M. Panic, J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. Experiences and Results of Parallelisation of Industrial Hard Real-time Applications for the parMERASA Multi-core. In *Proceedings of the 3rd Workshop on High-performance and Real-time Embedded Systems (HiRES)*, 2015.