embeddedworld 2015
Exhibition&Conference
... it´s a smarter world

Embedded Multicore Building Blocks

Parallel Programming Made Easy

Tobias Schüle
Corporate Technology
Siemens AG
Munich, Germany
tobias.schuele@siemens.com

***Abstract*—In this paper, we present an open source C/C++ library called Embedded Multicore Building Blocks (EMB²) which aims to simplify the development of parallel applications for embedded systems. EMB² relieves software developers from the burden of thread management and synchronization which significantly improves reliability, performance, and productivity. The library is based on MTAPI, an international standard for task management in embedded systems.**

Keywords—multicore processors; embedded systems; parallel programming; task management; MTAPI

I. INTRODUCTION

Multicore processors, which are prevalent in various classes of devices, from smart phones to servers, increasingly find their way into embedded systems. However, to leverage the computational power of such processors, the applications have to be parallelized. Unfortunately, this raises several challenges for software developers. For example, concurrency bugs such as deadlocks and race conditions compromise product quality and require special tool support to ensure correctness of the applications. Moreover, conventional approaches to parallel programming are tedious and usually limited to a small number of threads (cores).

To solve these problems, a number of frameworks have been developed in recent years. These include library-based approaches like Intel's Threading Building Blocks (TBB)[1] for C++ or Microsoft's Task Parallel Library (TPL)[2] for C#, as well as language extensions such as OpenMP[3]. As a major advantage, these approaches abstract from the target hardware, particularly the number of processor cores, and provide support for fine-grained tasks. The latter is essential to achieve scalability. However, most of these approaches have been designed for desktop and server applications making them hardly usable in embedded systems. The reasons for this are manifold: Firstly, they are optimized for high throughput and neglect real-time requirements frequently encountered in embedded systems such as minimal latencies. Secondly, they rely on dynamic memory allocation which is usually prohibited in embedded systems after startup. Thirdly, they target classical symmetric (homogeneous) multicore processors while in the embedded domain, heterogeneous systems-on-a-chip are becoming mainstream. This challenges software developers even further: According to a report recently published by the IEEE Computer Society, low-power scalable homogeneous and heterogeneous systems as well as hard real-time architectures are among the top challenges in computer science until 2022 [1].

In this paper, we present the Embedded Multicore Building Blocks (EMB²), an open source[4] C/C++ library for the development of parallel applications. EMB² has been specifically designed for embedded systems and the typical requirements that accompany them, such as real-time capability and constraints on memory consumption. As a major advantage, low-level operations are hidden in the library which relieves software developers from the burden of thread management and synchronization. This not only improves productivity of parallel software development, but also results in increased reliability and performance of the applications.

EMB² is independent of the hardware architecture (x86, ARM, ...) and runs on various platforms, from small devices to large systems containing numerous processor cores. It builds on the Multicore Task Management API (MTAPI)[5], a standardized programming interface for leveraging task parallelism in embedded systems containing symmetric or asymmetric multicore processors [2, 3]. One of the main features of MTAPI is low-overhead scheduling of fine-grained tasks among the available cores during runtime. Unlike existing libraries, EMB²

[1] https://www.threadingbuildingblocks.org/
[2] http://msdn.microsoft.com/en-us/library/vstudio/dd460717
[3] http://openmp.org/

[4] EMB² is publicly available under BSD (2-clause) license on https://github.com/siemens/embb
[5] http://www.multicore-association.org/workgroup/mtapi.php

supports task priorities and affinities, which allows the creation of soft real-time systems. Additionally, the scheduling strategy can be optimized for non-functional requirements such as minimal latency and fairness.

Besides the task scheduler, EMB² provides basic parallel algorithms, concurrent data structures, and skeletons for implementing dataflow-based applications (see Fig. 1). These building blocks are largely implemented in a non-blocking fashion [4], thus preventing frequently encountered pitfalls like lock contention, deadlocks, and priority inversion. As another advantage in real-time systems, the algorithms and data structures give certain progress guarantees. For example, wait-free data structures guarantee system-wide progress which means that every operation completes within a finite number of steps independently of any other concurrent operations on the same data structure.

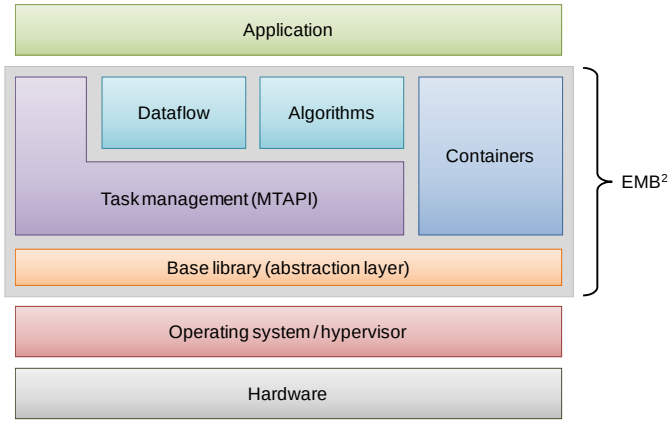

Fig. 1. Overview of the Embedded Multicore Building Blocks

In the following, we briefly describe the main components of EMB² and explain their usage by means of examples.

II. Components

A. Base Library (Abstraction Layer)

In order to achieve portability, the main components build on an abstraction layer that hides platform-specific details. This abstraction layer, also referred to as the base library, provides fundamental functionalities, mainly for creating and synchronizing threads as well as for memory management. Most of the functions are wrappers for features specific to the underlying operating system. Additionally, the base library provides atomic operations which are directly mapped to the instruction set of the target processor.[6] Atomic operations are essential for implementing efficient data structures and algorithms.

The base library is implemented in C so that it can be used in non-object-oriented applications or system-level code such as device drivers. However, EMB² also provides C++ interfaces for the base library which are easier to use and more safe. As an example, atomic data types are encapsulated in classes similar to C++11:

```
embb::base::Atomic<int> x; // atomic integer
```

[6] The current version of EMB² supports x86 and ARM processors.

This way, it is guaranteed that all operations such as `x++` are performed atomically. Additionally, implicit memory fences prevent subtle errors due to instruction reordering by the compiler or the processor.

B. Task Management (MTAPI)

Leveraging the power of multicore processors requires splitting computations into fine-grained tasks that can be executed in parallel. Threads are usually too heavy-weight for that purpose, since context switches consume a significant amount of time. Moreover, programming with threads is complex and error-prone due to concurrency issues such as race conditions and deadlocks. To solve these problems, efficient task scheduling techniques have been developed that distribute the available tasks among a fixed number of worker threads [5]. To reduce the overhead, there is usually exactly one worker thread for each processor core.

While task schedulers are nowadays widely used, especially in desktop and server applications, they are typically limited to a single operating system running on a homogeneous multicore processor. System-wide task management in heterogeneous embedded systems must be realized explicitly with low-level communication mechanisms. MTAPI addresses these issues by providing an API which allows parallel embedded software to be designed in a straightforward way, covering homogeneous and heterogeneous multicore architectures. It abstracts from the hardware details and lets software developers focus on the application.

As shown in Fig. 2, MTAPI provides two basic programming models, tasks and queues, which can be implemented by software or hardware [2, 3]. Tasks are not limited to a single multicore processor but may also be executed on remote nodes (the developer decides where to deploy a task implementation). For that purpose, MTAPI provides a basic mechanism to pass data to the remote node and back to the calling node. Queues can be used to control the scheduling policy for related tasks. For example, order-preserving queues ensure that subsequent tasks are executed sequentially. Other scheduling policies are possible and may even be implemented in hardware.

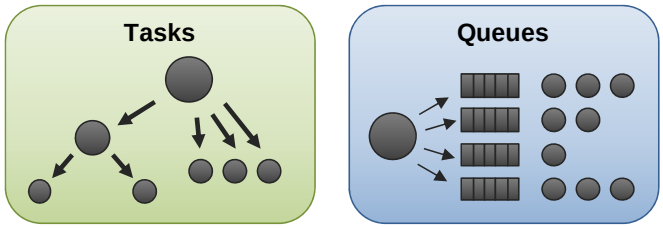

Fig. 2. MTAPI programming models

MTAPI defines C interfaces to be usable in a variety of systems. Similar to the base library, EMB² provides C++ wrappers for a subset of the C interfaces. Fig. 3 shows an example for task creation and synchronization. The `main` function first gets a reference to the current `node`. By calling `Spawn`, it then creates a task that executes the function `work` in parallel to `main`. Finally, it waits for the task to finish (the argument passed to `Wait` specifies the timeout). Note that the function `work` takes as parameter a reference to an object of type `TaskContext`,

which provides information about the status of the task and can be used to set an error code to be returned by `Wait`.

```
using namespace embb::mtapi;

void work(TaskContext& task_context) {
  // do work ...
}

int main() {
  Node& node = Node::GetInstance();
  Task task = node.Spawn(&work);
  // do something...
  mtapi_status_t status =
    task.Wait(MTAPI_INFINITE);
  // check status ...
}
```

Fig. 3. Example for task creation and synchronization

C. Algorithms

The *algorithms* building block contains skeletons for typical parallelization tasks such as parallel loops. They split computations to be performed in parallel into tasks which are executed by the MTAPI task scheduler. Suppose, for example, we are given a range of integer values and want to double each of them. In principle, one could apply the operation to all elements in parallel as there are no data dependencies. However, this results in unnecessary overhead if the number of elements is greater than the number of available processor cores. A better solution is to partition the range into blocks and to process the elements of a block sequentially. With the `ForEach` construct provided by EMB², users do not have to care about the partitioning into chunks of reasonable size, since this is done automatically. Similar to the Standard Library's `for_each` function, it is sufficient to pass the operation in form of a function object or a lambda function (see Fig. 4).

```
std::vector<int> v;
// initialize v ...
embb::algorithms::ForEach(v.begin(), v.end(),
  [] (int& x) {x *= 2;});
```

Fig. 4. Example for parallel for-each loop

In the previous subsection, we already saw how to run and wait for tasks. A more convenient way to run multiple tasks in parallel is to use the `Invoke` function that implicitly waits until all tasks have finished. As an example, assume we want to parallelize the quick sort algorithm which partitions an array into subarrays and sorts them recursively. This can be accomplished using `Invoke` as shown in Fig. 5 (for the sake of brevity, the implementation of `Partition` is omitted). However, it should be mentioned that this is not yet optimal. In particular, the creation of new tasks should be stopped when the subarrays are small enough in order to reduce the overhead for task creation and management.

All functions that directly or indirectly create tasks have optional parameters for specifying the tasks' priorities and affinities. Priorities are useful for (soft) real-time applications, where

```
using embb::algorithms::Invoke;
void ParallelQuickSort(int* first, int* last) {
  if (last - first <= 1) return;
  int* mid = Partition(first, last);
  Invoke([=](){ParallelQuickSort(first, mid);},
         [=](){ParallelQuickSort(mid, last);});
}
```

Fig. 5. Example for parallel function invocation

certain computations must be executed within a given amount of time. Affinities specify on which processor cores the tasks shall be executed. This way, the available cores can be partitioned into groups, each dedicated to certain kinds of computations. In industrial automation, for example, two cores of a quad-core processor may be reserved for the control software and the other two cores for the human machine interface.

D. Dataflow

Embedded systems often process continuous streams of data, e.g., audio signals, network packets, or sequences of images. EMB² provides generic skeletons for the development of parallel stream-based applications. In the simplest case, they can be modeled as ordinary pipelines, but more complex structures are also possible including conditional execution [6]. Fig. 6 depicts some examples for dataflow networks supported by EMB².

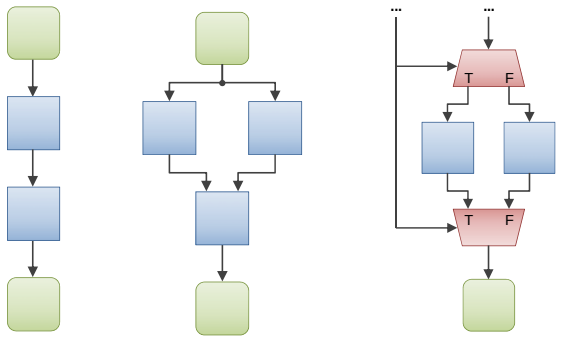

Fig. 6. Different kinds of dataflow networks supported by EMB²

Consider, as a simple example, an application that reads a sequence of strings from a file and writes them to the console. Fig. 7 shows how to parallelize this application as a two-stage pipeline. As the first step, we create a network, where the tem-

```
// Create network with at most 2 elements being
// processed in parallel
typedef embb::dataflow::Network<2> Network;
Network nw;
// Create pipeline stages
Network::Source<std::string> read(...);
Network::Sink<std::string> write(...);
// Connect pipeline stages
read.GetOutput<0>() >> write.GetInput<0>();
// Add pipeline stages to the network
nw.Add(read);
nw.Add(write);
// Run the network
nw();
```

Fig. 7. Example for a two-stage pipeline

plate parameter of class `Network` specifies the maximum number of elements that are processed in parallel. By limiting the number of elements it is avoided that the network is flooded with new elements before the previous ones have been processed. Then, we create a source (`read`) and a sink (`write`), where the actual functionality is specified in the constructor calls of the classes `Network::Source` and `Network::Sink`, respectively. Finally, we connect the stages, register them by calling `Add`, and run the network.

E. Containers

Containers are essential for storing objects in an organized way. Since the containers which are part of the C++ Standard Library are not thread-safe, they cannot be used in a multi-threaded environment without manual synchronization. EMB² provides containers that enable a high degree of parallelism by design. They are implemented in a lock- or wait-free fashion, thus avoiding any blocking operations [4]. This is particularly important for real-time systems which require certain progress guarantees. For instance, a non-real-time task passing data to some real-time task must not prevent the latter from making progress. Lock-free algorithms guarantee that the system as a whole always makes progress, whereas wait-free algorithms even guarantee progress for each thread [7]. Additionally, these algorithms exhibit certain properties essential in the embedded domain:

- Absence of deadlocks: Lock- and wait-free algorithms are immune to deadlock conditions.
- Avoidance of priority inversion: Threads cannot delay other threads with higher priority.
- Signal safety: Coherency in the context of asynchronous interrupts is guaranteed.
- Termination safety: An operation may be aborted without sacrificing overall system availability.

Another feature of the EMB² containers is that all memory is allocated during construction. Especially in safety-critical areas, dynamic memory allocation is usually forbidden during operation to avoid unpredictable behavior. Of course, this requires that the maximum number of elements stored in a container is known in advance. In the example shown in Fig. 8, we create a multiple-producer multiple-consumer (MPMC) queue that can hold up to ten elements. The methods `TryEnqueue` and `TryDequeue` return a Boolean value, where `false` indicates the queue is full or empty, respectively.

```
// create queue with capacity 10
embb::containers::LockFreeMPMCQueue<double>
  queue(10);
// enqueue element
assert(queue.TryEnqueue(1.0));
// dequeue element
double x;
assert(queue.TryDequeue(x));
```

Fig. 8. Example for container usage

III. Summary and Conclusion

With the current semiconductor technologies, a significant performance increase can only be achieved by parallelism. This not only holds for high-end processors used in servers but also for small embedded systems. In fact, even mobile devices like smart phones and tablets already contain multicore processors. To leverage their performance, libraries and runtime systems are necessary that split complex computations into fine-grained tasks and execute them in parallel. Task-based programming models have proven to be efficient in practice and are widely used in various domains.

However, embedded systems pose additional challenges to the developers. These include real-time capability and resource awareness especially concerning memory consumption. EMB² addresses these issues while providing easy and safe to use interfaces that enable parallelization of existing code. Key features are fine-grained control over core usage (task priorities, affinities) and predictable resource allocation.

Currently, we are working on support for distributed and heterogeneous systems. Using MTAPI's features to schedule tasks on different compute nodes, expensive computations can be carried out by specialized hardware such as GPUs, DSPs, or FPGAs in a power-efficient way. This is not only crucial for battery-powered devices but also for embedded systems with limited capabilities to dissipate heat, e.g., controllers. Also, we plan to extend EMB² with additional lock-/wait-free data structures such as trees.

IV. Acknowledgments

The author would like to thank everyone who has contributed to the development of EMB².

References

[1] H. Alkhatib, P. Faraboschi, E. Frachtenberg, H. Kasahara, D. Lange, P. Laplante, A. Merchant, D. Milojicic, and K. Schwan. "IEEE CS 2022 Report". IEEE Computer Society, 2014.

[2] U. Gleim and M. Levy. "MTAPI: Parallel Programming for Embedded Multicore Systems". The Multicore Association, 2013.

[3] "Multicore Task Management API (MTAPI) Specification V1.0". The Multicore Association, 2013.

[4] M. Herlihy and N. Shavit. "The Art of Multiprocessor Programming". Morgan Kaufmann, 2012.

[5] R. D. Blumofe and C. E. Leiserson. "Scheduling multithreaded computations by work stealing." J. ACM, vol. 46, nr. 5, ACM, 1999.

[6] Tobias Schuele. "Efficient Parallel Execution of Streaming Applications on Multi-core Processors". International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2011.

[7] M. Herlihy and N. Shavit. "On the Nature of Progress". International Conference on Principles of Distributed Systems (OPODIS), Springer, 2011.