Embedded Multicore Building Blocks (EMB²)
Dr. Tobias Schüle
Introduction

Sequential programming is easy (sometimes) ...

Dot product (sequential)

```c
#define SIZE 1000

main() {
    double a[SIZE], b[SIZE];
    // Compute a and b ...
    double sum = 0.0;
    for(int i = 0; i < SIZE; i++)
        sum += a[i] * b[i];
    // Use sum ...
}
```
Introduction

... but multithreaded programming is tedious!

Dot product (POSIX threads)

```c
#include <iostream>
#include <pthread.h>

#define THREADS 4
#define SIZE 1000

using namespace std;

double a[SIZE], b[SIZE], sum;

pthread_mutex_t mutex_sum;

void *dotprod(void *arg) {
  int my_id = (int)arg;
  int my_first = my_id * SIZE/THREADS;
  int my_last = (my_id + 1) * SIZE/THREADS;
  double partial_sum = 0;
  for(int i = my_first; i < my_last && i < SIZE; i++)
    partial_sum += a[i] * b[i];
  pthread_mutex_lock(&mutex_sum);
  sum += partial_sum;
  pthread_mutex_unlock(&mutex_sum);
  pthread_exit((void*)0);
}

int main(int argc, char *argv[]) {
  // Compute a and b ...

  pthread_attr_t attr;
  pthread_t threads[THREADS];

  pthread_mutex_init(&mutex_sum, NULL);
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  sum = 0;
  for(int i = 0; i < THREADS; i++)
    pthread_create(&threads[i], &attr, dotprod, (void*)i);
  pthread_attr_destroy(&attr);

  int status;
  for(int i = 0; i < THREADS; i++)
    pthread_join(threads[i], (void**)&status);
  // Use sum ...

  pthread_mutex_destroy(&mutex_sum);
  pthread_exit(NULL);
}
```

Introduction

Multicore processors are here to stay

Multicore processors have revolutionized the computing landscape by offering higher computing power, higher energy efficiency, and lower material costs. This shift has been driven by advancements in technology and the increasing demand for more powerful and efficient computing solutions.

Capacitance: $C$
Voltage: $V$
Frequency: $f$

Power: $CV^2f$


Single-core processor

Dual-core processor

Core 1

Core 2

Capacitance = 2.2C Voltage = 0.6V Frequency = 0.5f Power $\approx 0.4CV^2f$

Low material costs
High energy efficiency
High computing power

Source: Vishwani D. Agrawal
Introduction

“In 2022, multicore will be everywhere.”

“Multicore has attracted wide attention from the embedded systems community […].

However, to obtain good multicore performance, software is key for decomposing an original sequential program into parallel program parts and assigning them to processor cores.

So far, such parallelization has been performed by application programmers, but it is very difficult, takes a long time, and has a high cost.”

---


Introduction
Frameworks and Libraries for Parallel Programming

Most frameworks for parallel programming are intended for desktop/server applications and are not suitable for embedded systems.

Top challenges for multicore (IEEE CS 2022)
- Low-power scalable **homogeneous and heterogeneous architectures**
- **Hard real-time architectures** with local memory and their programming
- ...

Parallel Patterns Library (PPL)
Threading Building Blocks (TBB)
Apple’s Grand Central Dispatch
Embedded Multicore Building Blocks
Overview

**Embedded Multicore Building Blocks (EMB²)**

Domain-independent C/C++ library and runtime platform for embedded multicore systems.

**Key features:**

- Easy parallelization of existing code
- Resource-awareness (memory consumption)
- Real-time capability
- Fine-grained control over core usage (priorities, affinities)
- Support for distributed / heterogeneous systems
- Independence of hardware architecture (x86, ARM, …)
Embedded Multicore Building Blocks
Multicore Task Management API (MTAPI)

MTAPI in a nut shell

- **Standardized API** for task-parallel programming on a wide range of hardware architectures
- Developed and driven by practitioners of *market-leading companies*
- Part of Multicore-Association’s **ecosystem** (MRAPI, MCAPI, …)

**Heterogeneous Systems**
- Shared memory
- Distributed memory
- Different instruction set architectures

**Tasks**

**Queues**

**Contributing members:**

- **SIEMENS**
  - Working group lead
- **Texas Instruments**
- **Qualcomm**
- **ENEA**
- **PolyCore Software**
- **Freescale**
- **LSI**
- **Plurality**
- **Wind River**
Embedded Multicore Building Blocks

Heterogeneous systems

TI OMAP5430

Xilinx Zynq UltraScale MPSoC

Page 9

Corporate Technology

Unrestricted © Siemens AG 2016. All rights reserved
Embedded Multicore Building Blocks

MTAPI for Heterogeneous Systems (1)
- **Job**: A piece of processing implemented by an action. Each job has a unique identifier.
- **Action**: Implementation of a job, may be hardware or software-defined.
- **Task**: Execution of a job resulting in the invocation of an action implementing the job associated with some data to be processed.

![Diagram](image)
Embedded Multicore Building Blocks

Components

- Application
  - Dataflow
  - Algorithms
  - Containers
- Task management (MTAPI)
- Base library (abstraction layer)
- Operating system / hypervisor
- Hardware

EMB²
Parallel for-each loop

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

No need to care of
- task creation and management
- number of processor cores
- load balancing and scheduling
- ...

Function invocation

```cpp
// Create execution policy
ExecutionPolicy policy(true, 0);
// Remove worker thread 0 from affinity set
policy.RemoveWorker(0);
// Start high priority tasks in parallel on
// specified worker threads (cores)
Invoke([=](){HighPrioFun1();},
    [=](){HighPrioFun2();},
    policy);
```

1st argument: affinity set (true = all)
2nd argument: priority (0 = highest)

Example: worker thread (core) 0 is reserved for special tasks

Pass policy as optional parameter
Dot product (sequential)

```c
#define SIZE 1000

main() {
    double a[SIZE], b[SIZE];
    // Compute a and b ... 
    double sum = 0.0;
    for(int i = 0; i < SIZE; i++)
        sum += a[i] * b[i];
    // Use sum ...
}
```

\[ E = mb^2 \]
Dot product (EMB²)

```c
#define SIZE 1000

main() {
    double a[SIZE], b[SIZE];
    // Compute a and b ...
    double sum = Reduce(
        Zip(&a[0], &b[0]), Zip(&a[SIZE], &b[SIZE]),
        0.0,
        std::plus<double>(),
        [] (const ZipPair<double&, double&>& p) {
            return p.First() * p.Second();
        });
    // Use sum ...
}
```

Recipe (parallel algorithm)

1. Input sequence
2. Neutral element
3. Reduction op.
4. Transformation fn.

No need to care of
- task creation and management
- number of processor cores
- load balancing and scheduling
- …
### Function invocation

```c
// Create execution policy
ExecutionPolicy policy(true, 0);

// Remove worker thread (core) 0 from affinity set
policy.RemoveWorker(0);

// Start high priority tasks in parallel on specified worker threads (cores)
Invoke( [=](){HighPrioFun1();},
       [=](){HighPrioFun2();},
       policy);
```

- **1st argument**: affinity set (true = all)
- **2nd argument**: priority (0 = highest)

Example: worker thread (core) 0 is reserved for special tasks.

Pass policy as optional parameter.
Embedded Multicore Building Blocks

Dataflow Framework

Stream processing

- Embedded systems frequently process **continuous streams of data** such as
  - sensor and actuator data,
  - network packets, …
  - medical images, …
- Such applications can be modeled using **dataflow networks** and executed in parallel
Embedded Multicore Building Blocks

Components

- Application
  - Dataflow
  - Algorithms
  - Task management (MTAPI)
- Containers
- Base library (abstraction layer)
- Operating system
- Hardware

EMB\textsuperscript{2}
Embedded Multicore Building Blocks

Container Requirements

1. **No race conditions** in case of concurrent accesses
   - **Thread safety**

2. **No unpredictable delays** in case of contention
   - **Progress guarantee**

3. **No dynamic memory allocation** after startup
   - **Preallocated memory**

![Diagram](image)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Thread safety</th>
<th>Progress guarantee</th>
<th>Preallocated memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>std::queue QQQueue (Qt)</code></td>
<td>✗</td>
<td>—</td>
<td>✗</td>
</tr>
<tr>
<td><code>std::queue QQQueue (Qt) + Mutex</code></td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><code>boost::lockfree::queue</code></td>
<td>✓</td>
<td>✓ / ?</td>
<td>✗ / ?</td>
</tr>
<tr>
<td><code>tbb::concurrent_queue</code></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><code>embb::LockFreeMPMCQueue</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><code>embb::WaitFreeSPSCQueue</code></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
At least one thread always makes progress

- Lock-free
- Clash-free
- Deadlock-free

Wait-free
- Obstruction-free
- Starvation-free

All threads always make progress

Embedded Multicore Building Blocks

Lock-free / Wait-free Algorithms

Constructor & Destructor Documentation

```
template<
    typename Type,
    class Allocator = embb::base::Allocator<
        embb::containers::WaitFreeSPSCQueue<
            Type,
            Allocator>
    >
>
bool
embb::containers::WaitFreeSPSCQueue<
    Type,
    Allocator>::TryEnqueue
    ( Type const & element )

Tries to enqueue an element into the queue.

Returns
true if the element could be enqueued, false if the queue is full.

Concurrency
Thread-safe and wait-free

Dynamic memory allocation
Allocates capacity elements of type Type.

Concurrence
Not thread-safe

See Also
Queue Concept

Parameters
capacity Capacity of the queue
```
Embedded Multicore Building Blocks
Properties of Lock-free / Wait-free Algorithms

- **Progress guarantees**
  With wait-freedom, the completion of an operation is guaranteed to occur in a finite number of steps. Lock-freedom guarantees the overall progress of a system.

- **Deadlock absence**
  Wait-free and lock-free data structures are immune to deadlock conditions.

- **Signal safety**
  Coherency in the context of asynchronous interruptions is guaranteed.

- **Termination safety**
  Linearizable operations may be aborted at any time without sacrificing the overall availability of a system.

- **Priority inversion avoidance**
  Wait-free algorithms cannot prevent high priority threads from making progress.
Embedded Multicore Building Blocks
Tracing/Profiling MTAPI Applications with Score-P

- Open source community
- Linux & Windows
- Platform independent (x86, ARM, and PPC)
- Heterogeneous system support (e.g., Intel Phi, CUDA)
- Open formats enabling interoperability and custom analysis types
- Extremely scalable

www.score-p.org
Embedded Multicore Building Blocks
Visualization and Interpretation

- Adaptation of well-established and widely used Score-P framework to MTAPI ⇒ Benefit from experience in HPC
- Cooperation with Jülich Supercomputing Centre (JSC)
Embedded Multicore Building Blocks
Hello World!

https://github.com/siemens/embb/

BSD 2-clause license

Feedback and contributions are very welcome!
Embedded Multicore Building Blocks
Code Quality

- Formal verification (partially)
- Static source code analysis
- Linearizability checker
- Rule checker (cppcheck)
- Continuous integration
- Unit tests (> 90% statement coverage)
- Workflow-driven design/code reviews
- Coding guidelines (Google’s cpplint)
- Zero compiler warnings
Measurements from University of Houston show efficiency of EMB² (green bars):

Embedded Multicore Building Blocks

Summary

- Efficient software development
- High performance and scalability
- Improved code quality (prevention of concurrency bugs)
- Suitable for embedded systems (memory and real-time constraints)
Contact

Dr. Tobias Schüle
Siemens AG
Corporate Technology
CT RDA ITP SES-DE
Otto-Hahn-Ring 6
81739 München
Germany
Email: tobias.schuele@siemens.com