
An extension of embedded C for parallel programming

Master-Thesis von Bastian Gorholt
August 2014



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Software Engineering Group

An extension of embedded C for parallel programming

Vorgelegte Master-Thesis von Bastian Gorholt

1. Gutachten: Sebastian Erdweg

2. Gutachten:

Tag der Einreichung:

Inhaltsverzeichnis

1	Introduction	2
2	Background	3
2.1	Processes and Threads	3
2.2	Parallelism and concurrency	3
2.3	Types of parallelism	3
2.4	Data races	3
2.5	Communication model: shared memory	3
2.6	Communication model: message passing	4
2.7	Communication model: transactional memory	4
2.8	Coarse- and fine-grained synchronization	4
2.9	Embedded programming	4
2.10	MPS and mbeddr	5
2.11	C	5
3	Design and Translation	6
3.1	Tasks	6
3.1.1	Design	6
3.1.2	Translation	7
3.1.3	Example code	8
3.2	Futures	10
3.2.1	Design	10
3.2.2	Translation	10
3.2.3	Example code	13
3.3	Shared memory	14
3.3.1	Design	14
3.3.2	Translation of shared types	16
3.3.3	Translation of synchronization statements	21
3.3.4	Example code	23
3.4	Safety measures	27
3.4.1	Avoidance of implicitly shared unprotected data	27
3.4.2	Copying pointers to unshared data into tasks	28
3.4.3	Unsynchronized access to synchronizable data	28
3.4.4	Address leakage of shared resource values	29
3.4.5	Overwriting shared resources	29
4	Evaluation	30
	Literaturverzeichnis	31

1 Introduction

- mbeddr is a language and ide for embedded programming that facilitates programming by providing higher level language mechanisms
- parallel programming is becoming ever more important
- yet, mbeddr is still lacking higher-level support for parallel programming
- compared to library extensions language based support for pp features multiple benefits: individual problem-specific syntax, reduction of erroneous input by type system enhancements, program-wide optimization of generated code
- short examples for benefits
- ParallelMbeddr introduces task-based explicit parallel programming with shared memory that can be synchronized
- Objectives: appropriate syntax, type system support for safer code, optimized output
- outline of the work

Kommentar von Bastian

Question: Introduce example that is later used for the evaluation already here for motivational reasons?

2 Background

2.1 Processes and Threads

Every program in execution which may be delayed or interrupted is represented by a process. A process typically has its own protected data (via virtual memory) and execution state and consists of one or more threads. A thread, also known as lightweight process, shares some of the memory with its process but has its own execution state and thread-local storage as needed[13, p. 20]. In the course of programming language and operating system development several variants of threads have been devised, among others green threads, fibers and coroutines which mainly differ in how they are managed.

2.2 Parallelism and concurrency

If multiple threads are “in the middle of executing code at the same time”[21, p. 124(?)] they are processed concurrently. They can be executed at the same time on different processors or interleaved on a single processor which means that they are executed in an alternating way. The former is also called parallelism. Parallelism generalized to “the quality of occurring at the same time”[9, p. 91] can manifest in different ways

2.3 Types of parallelism

At least four different kinds of parallelism have been conceived. From an application programmer’s view seen at a very low level exist bit-level parallelism and instruction-level parallelism. Bit-level parallelism is concerned with increasing the word size of processors in order to reduce the amount of cycles that are needed to perform an instruction[7, p. 15]. Instruction-level parallelism, also called pipelining, is the simultaneous utilization of multiple stages of the execution pipeline of the processor. Both bit-level parallelism and instruction-level parallelism primarily reside on the hardware level and the operating system level and are, thus, no subject of this work. Data parallelism is present if the same calculation is performed on multiple sets of data and can be regarded as a specialization of task parallelism which denotes the simultaneous execution of different calculations “on either the same or different data”[7, p. 125]. The latter being the more general approach to software-level parallelism is the subject of this work.

2.4 Data races

When a process consists of multiple concurrently running threads which have access to the shared data of their process a second class of errors that is unique to parallel (and distributed) programming arises. These so-called synchronisation errors occur due to data races and are a result of the general non-atomicity of computations and memory references. Data races— also known as race conditions— are defined as at least “two unsynchronised memory references by two processes on one memory location, of which at least one reference is a write access”[10, p. 327]¹. Such data races can result in inconsistent program states and non-deterministic program behavior since the order in which the concerned memory is accessed might change. In order to deal with this issue three main paradigms have been conceived in parallel programming.

2.5 Communication model: shared memory

The memory model that implicitly underlied the former treatment of processes which share some data with their threads is formally known as *shared memory* . In this model communication between entities is realized by shared-

¹ As the definition implies data races are not limited to threads and the shared process data. E.g. file-based race conditions can even occur between two different processes[35]. Since this work is about parallelization of single processes other kinds of race conditions are not further considered.

memory regions which are written to and read from[15, p. 138]². Data races can be avoided with the help of the low-level synchronization primitive *mutex*³. A mutex can be locked by exactly one thread. Any other thread that tries to lock the same mutex is blocked until the locking thread releases the mutex[22]. Thus, code regions can be protected by having threads synchronize over mutexes that protect these regions. One of the disadvantages of mutexes is that they are not tightly coupled to the data or computation that they protect. It is the programmer's duty to take care of the sane utilization of a certain mutex. Therefore various higher-level synchronization measures like monitors in Java(TODO: reference) and synchronized classes in D(TODO: forward reference) were developed. These measures are usually built on top of mutexes[19, p. 25]. Another disadvantage of mutexes is their vulnerability for deadlocks which means that multiple processes are in a state where "each is waiting for release of a resource which is currently held by some other process"[29, p. 119] such that no progress will ever finish executing[11, p. 2-3].

2.6 Communication model: message passing

Whereas communication in the shared memory paradigm happens rather implicitly it is done explicitly in the *message passing* paradigm. Message passing originates from Hoare's paper on Communicating Sequential Processes (CSP)(TODO: reference!). In CSP messages are sent from one entity to another. "The sender waits until the receiver has accepted the message (*synchronous* message passing)"[31, p. 138]. Message passing with asynchronous message sends were deployed by the actor model[18] and pi calculus[25]. Although message passing avoids shared data and realizes communication generally via copies of data⁴ it still suffers from potential race conditions[27].

2.7 Communication model: transactional memory

Transactional memory provides a non-blocking⁵ memory model which enables communication via "lightweight, *in-memory* transactions"[14, p. 3] which are code blocks that from a programmer's perspective are executed atomically. The illusion of atomicity is realized by the underlying transaction system which may execute transactions in parallel and has to take care of conflicting reads and writes in transactions⁶[17]. Transactional memory can either be realized in hardware or in software. While the former promises a better performance it demands specific hardware. Software transactional memory on the other hand seems to suffer from comparatively "poor performance"[3, p. 13].

2.8 Coarse- and fine-grained synchronization

In order to keep structures and computations synchronized the simplest approach to avoid data races is to use the available measures like locks or transactions as broadly as possible. E.g. transactions could be widened to hold every operation a thread has to execute. As every synchronization is basically a serialization of otherwise parallel executed code such coarse-grained synchronization would eliminate the benefits of parallel execution. On the other hand fine-grained synchronization can introduce race conditions if the programmer misses some locking policy. In addition the acquisition of every lock takes time which can become an issue with increasing locking counts. Therefore a trade-off between locking-overhead and scalability problems has to be found[14, pp. 1-2].

2.9 Embedded programming

"An embedded system is a computerized system that is purpose-built for its application."[37, p. 1] Due to its narrow scope and monetary constraints induced by the application domain the hardware of such systems is often

² As for race conditions the model is not limited to intra-process communication via threads or similar approaches to parallelization. Communication between two processes can also be realized via shared memory but is not in the scope of this work.

³ Semaphores as a second synchronization primitive are closely related to mutexes. Since they do not provide further insights for the discussion they are not further investigated in this work.

⁴ Actually shared data is often used in implementations of message passing models in order to enhance the performance. Furthermore it exists on the language level like in terms of monitors that were developed by Hoare to reduce deadlocks in CSP. The main notion of the message passing concept nevertheless goes without shared data.

⁵ TODO: explain

⁶ To this end the corresponding transactions may need to be reexecuted as a whole.

constrained to the point that it just accomplishes the job[37]. Thus, the memory consumption of the resulting program is one main issue to be considered in embedded programming. Additionally, for real-time systems which constitute a subclass of embedded systems not only the correctness of computations but also the consumed time determine their quality and usefulness [6, pp. 1-2]. Therefore the predictability of the program's execution time becomes an issue for real-time systems⁷.

2.10 MPS and mbeddr

“JetBrains MPS⁸ is an open source [...] language workbench developed over the last ten years by JetBrains.”[36] As such it provides a projectional editor which lets the user directly work on the abstract syntax tree (AST) of the program[12]. It supports the development and composition of potentially syntactically ambiguous modular language extensions in combination with the development of integrated development environments (IDEs) or extensions thereof. Mbeddr is an extension of MPS tailored for the embedded software development in C. Every program written in the mbeddr IDE is translated to C99 source code which are then further processed by the gcc tool chain⁹. The implementation of C in mbeddr does not only provide extensions to the core of C but also has a few differences to the basis of C99. They will be introduced as needed.

2.11 C

The semantics of C differ from modern object-oriented languages like Java in a variety of ways. In order to clarify some of the choices that were made for the design and implementation of ParallelMbeddr the most relevant differences shall be outlined. Like Java C leverages pass-by-value semantics for function parameters. Differently though it generally copies the referenced values into the memory that is allocated for function calls whereas Java copies the references themselves. Thus a change to a field of a struct instance that was copied in such a way does not affect the original struct instance¹⁰. On the other hand arrays are treated like pointers which becomes evident when they are passed to functions. Hence a change of an entry of an array argument actually changes the array that is referred to by a variable on the caller site. In order to have an array be copied into a function it can be declared as a struct field which due to the copy semantics for structs ensures that like any other field of the struct instance the array's value is copied into the newly created struct instance. The copy semantics are not restricted to function arguments but also extend to function return values and variable assignments. The “pass-by-pointer-value” semantics for arrays implies that arrays cannot be returned from functions as is done in Java. Instead corresponding pointers are returned. This means that it is not safe to return an array, respectively a pointer thereof, from a function if the array resides on the area of the stack that was allocated for this function¹¹. A peculiarity of C is that global variables may only be initialized with constant expressions such that it is not possible to initialize a global variable with an arbitrary function call of a proper type[4, p. 48].

TODO: Add explanations for MPS aspects, type-system checking rules... as needed

TODO: Maybe add section for explicit vs. implicit parallel programming, look at Programming Distributed Systems by H. E. Bal, pp. 113-114

TODO: Implementation: coarse- vs. fine-grained synchronization, problems => implicit synchronization not exhaustive => optimization for safe lock avoidance helpful

⁷ Mbeddr does not have first-class support for the quantification of related parameters like the worst-case execution time (WCET)[6, p. 8], yet. For this reason predictability is a lesser concern of this thesis and will be reflected primarily in the careful consideration of the CPU consumption and processing time of the implementation.

⁸ <http://jetbrains.com/mps>

⁹ <http://gcc.gnu.org/>

¹⁰ The only way to avoid this behaviour is to copy the memory addresses of values as pointers into functions.

¹¹ Otherwise the pointed-to memory of the returned array pointer would become deallocated after the return of the called function. This again would cause the return of a dangling pointer into the receiver of the returned value, i.e. a pointer that does not point to a valid memory address.

3 Design and Translation

In this chapter the extension of mbeddr for parallel programming, called ParallelMbeddr, is introduced. To this end the new language features for C are explained each in terms of the design and the translation to plain mbeddr C code¹². In order to illustrate the presented features a running example is incrementally built. Further examples are depicted whenever the running example does not provide the right structure to clarify a feature. At the end of this chapter the implemented measures to make the extension sufficiently safe are explained.

3.1 Tasks

The basic parallelization element is a *task*. It denotes a parallel unit of execution and, as the name suggests, aims at task parallelism. As the implementation of the underlying parallelization technique might change in the future it is reasonable to abstract the terminology from it. The most basic task which always exists executes the code of the entry function of the program. A task can also be regarded as a closure of the expression that shall be run in parallel. The reader should distinguish this ‘execution template’ from the actual running instance of a task. The latter will further on be addressed as a *running task*.

3.1.1 Design

The syntax e of expressions in mbeddr is extended by

$e ::= \dots \mid |e|$

When executed a task term yields a handle to a parallel unit of execution. This way the initialization of the task and the actual execution are decoupled and can happen independently. When a task is run the embraced expression is executed and its value is returned. If the type of the expression is *void* no value will be returned. The type of a task reflects its return value:

$t ::= \dots \mid \text{Task}<t>$

Due to implementation reasons (see 3.2.2 for details) the embraced return type of a task must be either *void* or a pointer to the type of the embraced expression:

$$\text{VoidTask} \frac{e \vdash \text{void}}{|e| \vdash \text{Task}<\text{void}>} \qquad \text{NonVoidTask} \frac{e \vdash t \quad t \neq \text{void}}{|e| \vdash \text{Task}<\text{void}*>}$$

When a task is not used anymore to produce running instances of itself it should be cleared in order to free the memory that it implicitly occupies on the heap:

$e ::= \dots \mid e.\text{clear}$

$$\text{VoidTask} \frac{|e| \vdash \text{Task}<\text{void}>}{\text{void}}$$

If a task is copied by the pass-by-value semantics of C the copied task will share the heap-managed data, i.e. the reference environment of its free variables, with the original task. Therefore a task needs to be cleared only once in order to avoid memory leaks. Keep in mind that a running instance of a task will not be affected by the clearance of its task template. The clearance of a task is only necessary if the task is stored somewhere. Thus, a task that is directly run via $|e|.\text{run}$ need not be cleared which makes such expressions memory safe.

¹² For the sake of legibility the syntax of the generated mbeddr code is depicted in a simplified manner where it is deemed necessary.

3.1.2 Translation

The POSIX Threads standard and library (pthreads) was chosen as a means to realize concurrency in the translation. It supports all necessary parallelization features and provides a more direct control of the generated code when compared to frameworks like OpenMP¹³. Every task in ParallelMbeddr is represented by a thread as provided by the POSIX threads standard, a so-called *pthread*¹⁴. As the thread initialization function of pthreads takes a function pointer of type **void*** -> **void*** the computation of the translated task is represented by an according function:

```
1 || void* parFun_X(void* voidArgs) { ... }
```

The **X** in the name symbolizes that for every task a unique adaptee of this function with the prefix **parFun_** and some unique suffix chosen by the framework is generated¹⁵. As the function signature indicates a pthread and therefore a task to be run can be parameterized with values and can return a value which will be explained in the following paragraphs.

If a task contains any references to local variables or function arguments they need to be bound to capture the variable states at the time of the task initialization. Such state is represented by an ‘argument’ struct:

```
1 || struct Args_X {  
2 ||     t_1 v_1;  
3 ||     ...  
4 ||     t_n v_n;  
5 || }
```

where every **v_i** represents an equally named reference in the task expression to a variable of type **t_i**.

The generated function **parFun_X** is then given an instance of **Args_X** which it uses to bind the references of the task expression to. The full function definition of a task **e** of type **Task<t*>** is, thus:

```
1 || void* parFun_X(void* voidArgs) {  
2 ||     t* result = malloc(sizeof(t));  
3 ||     Args_X* args = (Args_X*) voidArgs;  
4 ||     *result = e';  
5 ||     return result;  
6 || }
```

where **e'** is the expression obtained when every local variable reference and function argument reference **r** in **e** is substituted by a reference to an equally named and typed field in args:

$r / args \rightarrow r$

If the embraced expression of a task does not contain any reference of this kind (e.g. only references to global variables) the **args** definition line is omitted as is clearly the—otherwise empty—declaration of **struct Args_X**. In this case **e'** equals **e** except for other reductions of **e** that might occur in the translation process of mbeddr.

The generated function of a task of type **Task<void>** renounces the result-related statements:

```
1 || void* parFun_X(void* voidArgs) {  
2 ||     t* result = malloc(sizeof(t));  
3 ||     Args_X* args = (Args_X*) voidArgs;  
4 ||     e';  
5 || }
```

Again, any argument-related code is generated as needed.

The aforementioned handle that a task yields is represented by an instance of a corresponding struct that captures both the initialization state and the computation of the embraced expression¹⁶. The *void* pointer of the arguments **voidArgs** does not keep their type information and with it their byte size. Therefore an additional field **argsSize** is needed in order to be able to create copies of the arguments later on (see 3.2.2 for details).

```
1 || exported struct Task {  
2 ||     void* args;
```

¹³ <http://openmp.org/>

¹⁴ In the following sections ‘pthreads’ will denote both the library and multiple threads as they are provided by the library. The context should always clarify which one is currently meant.

¹⁵ In the following explanations **X** will always denote some arbitrary suffix. Keep in mind that these suffixes do not necessarily coincide for different kinds of components.

¹⁶ (**void***) => (**void***) **fun** is mbeddr syntax for the not easily edible function pointer **void *(*fun) (void *)** in standard C99.

```

3 || (void*) => (void*) fun;
4 || size_t argsSize;
5 || }

```

As opposed to the unique definitions of other elements that need to be defined for every occurrence of a task (the ones with the **X** suffixes) **struct Task** is generic and is reused for every task. Generic declarations are kept in fixed separately generated modules and are imported into the user-defined modules. With these components in mind the actual translation of a task expression `|.|` that contains references `v_i` to `v_n` which need to be bound becomes an mbeddr block expression¹⁷:

<pre> 1 { 2 Args_X* args_X = malloc(sizeof(Args_X)); 3 args_X->v_1 = v_1; 4 ... 5 args_X->v_n = v_n; 6 yield (Task){ args_X, parFun_X, sizeof(Args_X) }; 7 } </pre>	\Rightarrow	<pre> 1 taskInit(v_1, ..., v_n) with function declaration 2 inline Args_X taskInit(t_1 v_1, ..., t_n v_n) { 3 Args_X* args_X = malloc(sizeof(Args_X)); 4 args_X->v_1 = v_1; 5 ... 6 args_X->v_n = v_n; 7 return (Task){ args_X, parFun_X, sizeof(Args_X) }; 8 } </pre>
--	---------------	--

The expression of the **yield** statement is a compound literal which on evaluation creates an instance of the aforementioned **struct Task**. The block expression is then further reduced by mbeddr to a call of a newly generated inline function¹⁸.

Without any references to bind a task is just reduced to the compound literal:

```

1 || (Task_X){ null, parFun_X, 0 }

```

The reduction of a task is accomplished differently if the task is immediately run via `|e|.run`. Section 3.2.2 will show how it is done.

By above definition of **parFun_X** it becomes clear that the arguments of a task—its environment—are stored on the heap before execution. This approach was chosen mainly in order to simplify the generation of the resulting code. In exchange both the result and the arguments of a task have to be deleted by the programmer by hand. Since for reasons to be explained later the result is returned via a pointer onto the heap, as well, it now becomes obvious that the return type of a task must either be a void type or a pointer type as was mentioned in the design section. Concerning task arguments one advantage of this implementation is that a task may be passed by value, e.g. when using a builder function to create tasks, without the possible need to copy multiple arguments. Instead just the pointer to the heap-managed data is copied. As will be shown in the (FUTURE WORK) chapter a stack-based implementation of task (I/O) is conceivable.

The clearance of a task **e.clear** in the generated code is a call of the **free** function of C parameterized with the arguments of the translated task **e'**:

```

1 || free(e'.args)

```

3.1.3 Example code

The running example is about the calculation of π , based on the definition given in the concurrent-pi example for go¹⁹. π is approximated by the summation of a certain number n of terms where n determines the deviation of the result from the actual value of π : $\pi_{\text{approx}} = \sum_{i=0}^n 4 * \frac{-1^i}{2i+1}$.

In the first scenario the amount of work is distributed under a certain number of tasks each of which calculates the contribution of summands for a range of indices i . The calculation of such a partial sum for a range `[start, end[` of indices is done by the functions:

¹⁷ A block expression contains a list of statements of which the mandatory yield statement returns the result value..

¹⁸ Whereas in C for every struct type **T** a typedef has to be defined in order to reference this type directly with **T** instead of **struct T** in mbeddr this definition is done implicitly.

¹⁹ <https://github.com/foamdino/learning-go/blob/master/concurrent-pi/concurrent-pi.go>

```

1 | long double calcPiRange(uint32 start, uint32 end) {
2 |     long double partialSum = 0;
3 |     for (uint32 i = start; i < end; ++i) {
4 |         partialSum += calcPiItem(i);
5 |     }
6 |     return partialSum;
7 | }
8 |
9 | long double calcPiItem(uint32 index) {
10 |     return 4.0 * (pow(-1.0, index) / (2.0 * index + 1.0));
11 | }

```

The work can be distributed among, e.g. 4 tasks where each task calculates a partial sum for an equally long range which is given by:

```

1 | #constant RANGESIZE = 300000000;
2 | #constant RANGECOUNT = 4;
3 | #constant THRESHOLD = RANGESIZE * RANGECOUNT;

```

These values are then used to initialize an array of tasks:

```

1 | int32 main(int32 argc, string[] argv) {
2 |     ...
3 |     Task<long double*>[RANGECOUNT] calculators;
4 |     for (i ++ in [0..RANGECOUNT]) {
5 |         uint32 start = i * RANGESIZE;
6 |         uint32 end = start + RANGESIZE;
7 |         calculators[i] = |calcPiRange(start, end)|;
8 |     }
9 |     ...
10 | }

```

The final reduction of the calculated values will be shown after the presentation of futures in section 3.2.3. The code is translated²⁰ to the building blocks that were introduced in 3.1.2:

```

1 |
2 | int32 main(int32 argc, string[] argv) {
3 |     ...
4 |     Task[RANGECOUNT] calculators;
5 |     for (int8 __i = 0; __i < RANGECOUNT; __i++) {
6 |         uint32 start = __i * RANGESIZE;
7 |         uint32 end = start + RANGESIZE;
8 |         calculators[__i] = taskInit_0(start, end);
9 |     }
10 |     ...
11 | }
12 |
13 | struct Args_0 {
14 |     uint32 start;
15 |     uint32 end;
16 | };
17 |
18 | inline Task taskInit_0(uint32 start, uint32 end) {
19 |     Args_0* args_0 = malloc(sizeof(Args_0));
20 |     Args_0->start = start;
21 |     Args_0->end = end;
22 |     return (Task){ args_0 , :parFun_0 , sizeof (Args_0)};
23 | }
24 |
25 | void* parFun_0(void* voidArgs) {
26 |     long double* result = malloc(sizeof(long double));

```

²⁰ For legibility reasons the code shown in the following listing and any other mbeddr code is a simplified version of the intermediate code that mbeddr actually generates. The performed changes are restricted to renamings and other minor adjustments.

```

27 | Args_0* args = ((Args_0*) voidArgs);
28 | *result = calcPiRange((args)->start, (args)->end);
29 | free(voidArgs);
30 | return result;
31 | }

```

The type of **calculators** is translated into an array type of the generic **Task** struct type such that the type specification **long double*** which is not needed in the translated code gets lost in this process. The task expression **|calcPiRange(start, end)|** is translated into a function call of the generated inline function **taskInit_0**. This function stores the values of the referenced local variables **start** and **end** in a structure which will later be used as the input for the parallel executed function **parFun_0**. This function, the wrapped arguments and their size are stored in a generic **Task** structure instance which is the handle that will later be used to initiate the task. **parFun_0** takes its arguments generically (as is required by the POSIX threads standard) and also returns its result generically via the heap. As the arguments reside on the heap and are uniquely allocated for this function they must be freed before **parFun_0** returns. The calculation of the result is straightforwardly given by the execution of the expression of the original task sub expression **calcPiRange(start, end)** except that the two variable references are substituted by references to the according fields in the argument struct instance **voidArgs** which is cast to the appropriate type **Args_0**.

3.2 Futures

Whenever a task **t** is run a *future* is generated. Futures in ParallelMbeddr are based on Halstead's definition of a future[16]. A future is a handle to a running task that can be used to retrieve the result of this task from within some other task **u**. As soon as this happens the formerly in parallel running task **u** joins **t** which means that it waits for **t** to finish execution in order to get its result value. The asynchronous execution is, thus, synchronized.

3.2.1 Design

The syntax **e** of expressions in mbeddr is extended by²¹:

e ::= ... | e.run | e.result | e.join

Like with tasks the type of a future is parameterized by its return type:

t ::= ... | Future<t>

e.run denotes the launch of task **e** whereas **e.result** joins a running task that is represented by a future handle **e**, i.e. halts the execution of the calling task until **e**'s execution is finished, and returns its result. The last expression **e.join** can be used to join tasks that return nothing. These properties are reflected in the typing rules:

$$\begin{array}{ccc}
 \text{Future} \frac{e \vdash \text{Task} \langle t \rangle}{e.\text{run} \vdash \text{Future} \langle t \rangle} &
 \text{FutureResult} \frac{e \vdash \text{Task} \langle t^* \rangle}{e.\text{result} \vdash t^*} &
 \text{FutureJoin} \frac{e \vdash \text{Task} \langle \text{void} \rangle}{e.\text{join} \vdash \text{void}}
 \end{array}$$

As was already depicted in the previous section **result** returns a pointer to a heap-managed value. Hence the programmer has to take care of freeing the value eventually.

3.2.2 Translation

A future type **Future<t*>** is translated to a generic **struct** that contains a handle to the thread, a storage for the result value—which is dropped for futures of type **Future<void>**—and a flag that indicates whether the thread is already finished:

```

1 | exported struct Future {
2 |     pthread_t pth;
3 |     boolean finished;
4 |     void* result;
5 | };

```

²¹ If the expression **e** has a pointer type the dots (.) are replaced by arrows (->).

For every task and future expression shown above a generic function reflects the semantics in the translation. The **run** of a task involves taking a task, creating a pthread with the task's function pointer and arguments and generating a future with the initialized thread handle²²:

```

1 Future runTaskAndGetFuture(Task task) {
2     pthread_t pth;
3     if ( task.argsSize == 0 ) {
4         pthread_create(&pth, 0, task.fun, 0);
5     } else {
6         void* args = malloc(task.argsSize);
7         memcpy(args, task.args, task.argsSize);
8         pthread_create(&pth, 0, task.fun, args);
9     }
10    return ( Future ){ .pth = pth };
11 }

```

The code shows that the arguments to be provided to the thread are copied onto a new location on the heap although they already reside on the heap as was shown in section 3.1.2. It is necessary to do so in order to avoid dangling pointers. These could arise when a task is cleared so that its arguments get deleted while one or more running instances (pthreads) of this task are not finished, yet. Furthermore generally every thread needs its own copy of the data in case it modifies it. A corresponding function to the previous function, called **runTaskAndGetVoidFuture**, is generated for futures that return nothing.

The signature of **pthread_create** indicates how the result of a threaded function can be received. As was already previously suggested it expects a function pointer of type **void* -> void*** which is the reason why the function generated for a task is equally typed. The result is, thus, a generic **void** pointer. This implies that the threaded function could generally return the address of a stack-managed value, i.e. a local variable. Since the existence of the value after thread termination could not be guaranteed a dangling pointer[32] could emerge, which resembles the problem for thread arguments. The only safe alternative that fits the task-future structure well is to allocate memory on the heap and return the address of this memory (see section 3.1.2). The translation of the result retrieval is a call the function:

```

1 void* getFutureResult(Future* future) {
2     if (!future->finished) {
3         pthread_join (future->pth, &(future->result));
4         future->finished = true;
5     }
6     return future->result;
7 }

```

First the future is used to join the according thread which blocks the execution until the thread is finished. Additionally the result is copied into the designated slot of the future struct instance. The result is at last returned. In POSIX a thread can only be joined once; every subsequent call causes a runtime error. In order to allow the user to request the result multiple times nevertheless the **finished** flag is used to decide whether a join should happen. The same basic structure can be found in the translation of the **join** function for a future of type **Future<void>**. The main difference is the missing result-related code:

```

1 void joinVoidFuture(VoidFuture* future) {
2     if (!future->finished) {
3         pthread_join (future->pth, null);
4         future->finished = true;
5     }
6 }

```

Both aforementioned generated functions take their future parameters by address. This is necessary to make the setting of the future data work. If futures struct instances would be passed to these functions as is due to C's pass-by-value semantics only copies of the provided future arguments would be filled with data. The result of a task would never arrive in the original future struct instance. Furthermore subsequent calls to these functions would

²² Obviously the thread handle is copied into the **Future**. This is safe as can be seen when looking at the POSIX function **pthread_t pthread_self(void)** which also returns a copy of a thread handle. This useful property is worth mentioning since it does not hold for all POSIX related data structures as is explained in footnote 33.

always work with false **finished** flags and ultimately trigger runtime errors. The necessity for future pointers in turn does not assort well with chained future expressions like:

```
1 Task<int32*> task = |(int32)23|;
2 int32* result23 = task.run.result;
```

In this sample code the result of the future is requested without being stored beforehand and accessed via address. Hence the code conflicts with the previously given definition of the translation of **result**. A first caveat would be to change the line to:

```
1 int32* result23 = (&(task.run))->result;
```

This on the other hand is not allowed because **task.run** is no lvalue[30, pp. 147-148] which disallows the utilization of the address operator on this expression. Instead in order to allow for chainings like **task.run.result** two wrapper functions, one for each **join** and **result**, are provided. These functions each take a future by argument, thus binding it to an adressable location, and call above corresponding functions in turn:

```
1 void* saveFutureAndGetResult(Future future) {
2     return getFutureResult(&future);
3 }
4
5 void saveAndJoinVoidFuture(VoidFuture future) {
6     joinVoidFuture(&future);
7 }
```

By making use of the presented functions the reductions of **e.run**, **e.join** and **e.result** (where **e'** is the reduced value of **e**) straightforwardly become function calls thereof:

```
1 || runTaskAndGetFuture(e')           1 || runTaskAndGetVoidFuture(e')
1 || ((t)getFutureResult(&e'))         1 || joinFuture(&e')
```

The type cast of the result returned by **getFutureResult(&e')** is necessary since it returns a generic pointer **void*** which may not be compatible with the receiver of the value. In consequence the result is cast to the result type of the future for which **e.result** was type checked. For expressions of the kind **|e|.run** the reduction to a call of **runTaskAndGetFuture()** is not applied. Instead a call to a specific function **futureInit_X** that combines both the logic of the task initialization and the future initialization is created. Suppose **e** contains references to local variables or arguments. Similar to the **taskInit_X()** expression block from section 3.1.2 the future initialization function first allocates memory on the heap for the values of the referred variables. To this end it uses an instance of the **Args_X** struct that was created for the task to store the values. Afterwards instead of wrapping the arguments struct inside a **Task** struct instance the function directly declares a pthread and initializes it with the arguments and a pointer to the function **parFun_X** that was created for the task:

```
1 Future futureInit_X(t_1 v_1, ..., t_n v_n) {
2     Args_X* args_X = malloc(sizeof(Args_X));
3     args_X->v_1 = v_1;
4     ...
5     args_X->v_n = v_n;
6     pthread_t pth;
7     pthread_create (&pth, null, :parFun_X, args_X);
8     return (Future){ .pth = pth };
9 }
```

If **e** does not contain any references to local variables or function arguments the arguments-related code is omitted and **null** is given as argument parameter to **pthread_create()**:

```
1 Future futureInit_X() {
2     pthread_t pth;
3     pthread_create(&pth, null, :parFun_X, null);
4     return (Future){ .pth = pth };
5 }
```

In case the type of **e** is **void** the struct type **Future** is replaced by **VoidFuture** in either declaration of **futureInit_X**. The code shows why no clearance of the arguments for a task in the case of a direct run of the task is required as was mentioned in section 3.1.1: Since the arguments struct instance is only used for exactly one running task

no further copies of the arguments are needed. Hence, the freeing of the heap-allocated memory can be left to the function **parFun_X**. **futureInit_X** is called in the reduction of **|e|.run**:

```
1 || futureInit_X(v_1, ..., v_n)
```

Whereas the following section will show the generation of expressions **e.run** to calls of **runTaskAndGetFuture** the reduction to a call of **futureInit_X** is depicted in section 3.3.4.

3.2.3 Example code

The running example concerning the π approximation from section 3.1.3 can now be extended with the result-related code. The tasks that were previously declared are used to initiate a running task instance of each such ‘template’:

```
1 int32 main(int32 argc, string[] argv) {
2     Task<long double*>[RANGECOUNT] calculators;
3     Future<long double*>[RANGECOUNT] partialResults;
4
5     ... // task declarations
6
7     for (i ++ in [0..RANGECOUNT]) {
8         partialResults[i] = calculators[i].run;
9         calculators[i].clear;
10    }
11
12    for (i ++ in [0..RANGECOUNT]) {
13        result += *(partialResults[i].result);
14        free(partialResults[i].result);
15    }
16 }
```

For every task that is run a future of the same type is created. After the initialization the tasks (i.e. the task templates, not the running instances thereof) are cleared in order to avoid memory leaks. The programmer is free to choose whether he is willing to do so. In case only a very limited amount of memory will be used by the tasks the clearance might not be deemed necessary. In the second loop the futures of all running tasks are used to retrieve and accumulate all partial π results. In the end their memory is freed since they are located on the heap. Again, the freeing is up to the programmer and in the end of the program might not be even useful. The translation of the new code becomes:

```
1 int32 main(int32 argc, string[] argv) {
2     Task[RANGECOUNT] calculators;
3     Future[RANGECOUNT] partialResults;
4
5     ... // task declarations
6
7     for (int8 __i = 0; __i < RANGECOUNT; __i++) {
8         partialResults[__i] = runTaskAndGetFuture(calculators[__i]);
9         free (calculators[__i].args);
10    }
11
12    for (int8 __i = 0; __i < RANGECOUNT; __i++) {
13        result += *((long double*) getFutureResult(&partialResults[__i]));
14        free((long double*) getFutureResult(&partialResults[__i]));
15    }
16 }
```

As the code shows the future type becomes the type of the according generic **Future** struct. Like the translation of the task type it loses the type parameterization **long double*** which is not necessary any more. The task running expression **calculators[i].run** and result retrieval expression **partialResults[i].result** are translated to calls of the newly generated generic functions **runTaskAndGetFuture()**, respectively **runTaskAndGetFuture()**. Their definitions can be seen in the previous section 3.1.2. Due to its genericity **runTaskAndGetFuture()** returns a result of type **void*** which means that its result must be cast by the compiler to an appropriate pointer type which in this

case is **long double*** like in the original task and future definitions. Finally, the task clearance is simply reduced to a call of C's free function parameterized with the pointer to an argument struct instance that the task holds.

3.3 Shared memory

The previous chapters introduced the means to enable parallel execution of code in terms of tasks and futures. Still missing is the communication between tasks. The communication model of choice for ParallelMbeddr is shared memory. The reason for this choice follows from the objectives for a communication model: it should offer a reasonable performance, considering that it is supposed to be used in embedded systems; it should be reasonably safe by design in order to avoid the trip hazards that are involved with low-level synchronization approaches like mutexes. Transactional memory seems to be not ready for the embedded domain for performance reasons. By following argumentation message passing does not offer profound advantages in comparison to shared memory if the access to the shared memory is controlled in a sane way. Usually message passing forbids shared memory between two parallel units of execution. Instead communication is realized via messages sends. A strict separation of memory does not fit the usual C workflow concerning pointer arithmetic. Therefore some form of memory sharing would have to be introduced into a message passing model in order to reduce performance loss. As this would lead to the same problems that the general shared memory model already suffers from the opposite way is chosen: Instead of introducing shared memory in a message passing model a shared memory model is designed on top of which message passing can ever be attached in order to simplify the communication between tasks²³.

Memory that is to be shared between two tasks must be explicitly declared by an according type. A variable of this type denotes a *shared ressource*. Thus, a shared ressource can be regarded as a wrapper of data that is to be shared. In order to make use of a shared ressource it has to be synchronized first. Specific language elements are used to access and change the value of a shared ressource. The chosen approach enables the programmer to use shared data both globally and locally and like any other data nest it inside structs and arrays. Additionally the new data type enables the IDE to ensure despite the arbitrary structuring of shared ressources that data is shared in a sane way.

3.3.1 Design

The ressources to be shared are typed with the shared ressource type:

$$t ::= \dots | \text{shared}\langle u \rangle | \text{shared}\langle \text{shared}\langle u \rangle * \rangle$$

$$u ::= t \quad u \neq t *$$

The type parameterization denotes the base type of a shared type, i.e. the type of the data that is wrapped by a shared ressource. Due to reasons that will be explained in the SAFETY CHAPTER a shared ressource may have an arbitrary base type that does not denote a pointer to a value that is not shared²⁴. Further restrictions apply to shared types which are also investigated in the aforementioned section. The same applies to the **.set** expression by which the value of a shared ressource can be modified; the value can be retrieved via **.get**:

$$e ::= e.\text{get} | e.\text{set}(e)$$

$$\text{SharedGet} \frac{e \vdash \text{shared}\langle t \rangle}{e.\text{get} \vdash t} \quad \text{SharedSet} \frac{e \vdash \text{shared}\langle t \rangle \quad e' \vdash t' \quad t' <: t}{e.\text{set}(e') \vdash \text{void}}$$

The syntax **stmts** of statements in mbeddr is extended by the synchronization statement **sync** that contains a *synchronization list* of shared ressources to synchronize over and a block of statements whose referenced shared ressources may be synchronized by a surrounding synchronization statement:

$$\text{stmt} ::= \dots | \text{sync}(\text{res}, \dots, \text{res}) \{ \text{stmt} \dots \text{stmt} \}$$

res denotes the syntax of possible *synchronization references*. Each synchronization reference **res** wraps a reference **e** to a shared ressource. **e** can either be of type **shared<t>** or of type **shared<t>***. A shared ressource can be synchronized as is or be named by a synchronization reference:

$$\text{res} ::= e | e \text{ as } [\text{resName}]$$

²³ Such an extension would be future concern and is not implemented in this work.

²⁴ Differently said: A pointer wrapped in a shared ressource must point to a shared ressource itself. Due to the comprehensive type system that mbeddr is equipped with and that is only partially existant in C99 the compatibility of base types of shared types was mainly tested with the primitive types of C99 as well as pointer types, array types, struct types and type definitions (typedefs). Future work will have to be done to demonstrate and establish full compatibility with the rest of mbeddr's types.

The latter allows the programmer to refer to the result of an arbitrary complex expression which evaluates to a shared resource inside the **sync** statement. Hence, a named resource (i.e. a synchronization reference with a name for its referenced shared resource) can be seen as syntactic sugar for a local variable declaration that binds a shared resource to which a shared resource reference evaluates. Due to the copy semantics of C the type of a named resource **e** is restricted to **shared<t*>**. This enforcement ensures that the synchronized resource is actually the original shared resource and not a copy thereof over which a synchronization would be useless. The scope of a named resource is restricted to the according synchronization statement. More precisely, a named resource *n* of a synchronization statement *s* can be referenced from anywhere inside the abstract syntax tree (AST) of the statement list of *s*. Furthermore it can be referenced from within the expression of any synchronization reference that follows *n* in the synchronization list of *s*, e.g.:

```

1 | shared<shared<int32>> v;
2 | // vContent is declared before it is used in the list => valid
3 | sync(v, &(v.get) as vContent, vContent->get as vContentContent) {
4 |   vContentContent.set(0);
5 | }
6 | // vContentContent is not in scope, here => invalid
7 | vContentContent.set(1);

```

The type of a reference to a named resource is given by the corresponding shared resource that is bound by the synchronization reference.

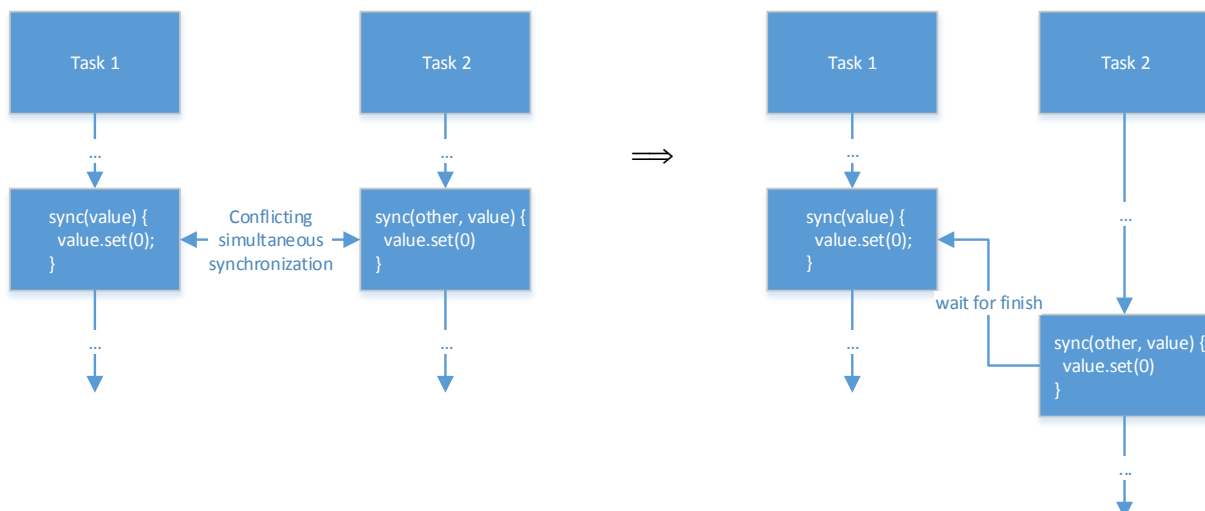
In contrast to Java's synchronization blocks and methods[34, p. 279] the synchronization of tasks is not computation oriented but data oriented. The crucial difference is that a synchronized block **A** in Java is only protected against simultaneous executions by multiple threads. Thus, it is valid to access the data that is involved in **A** by some other computation whose protecting block (if any) is completely unrelated to **A**. Since low-level data races can obviously not be safely excluded with this scheme ParallelMbeddr ties the protection to the data that shall be shared. Every shared resource is therefore protected separately and application-wide. Concluding, consider two synchronization blocks which are about to be executed in parallel. If they contain synchronization references which overlap in terms of their referenced shared resources their executions will be serialized. For instance take two tasks *t1* and *t2* that have access to the same shared resource which is referenced by a global variable **value**. *t1* wants to synchronize **value**, simultaneously *t2* wants to synchronize **valuePointer** which points to **value**'s shared resource and some other shared resource that is available via the variable **other**:

```

1 | shared<int32> value;
2 | shared<int32>* valuePointer = &value;
3 | shared<double> other;

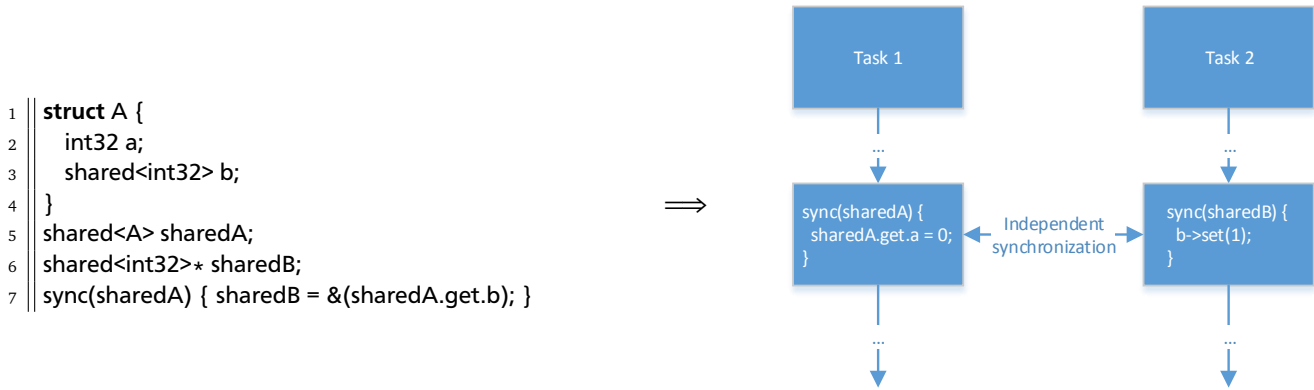
```

The synchronization semantics will then cause one thread to wait for the other to finish the execution of the blocking synchronization statement before it starts the execution of its own synchronization statement. The execution order might therefore be changed in the following manner:



The possibility to refer to multiple shared resources in the synchronization list of a synchronization statement is not mere syntactic sugar for nested synchronization statements. Instead the semantics of a synchronization list is that all referenced shared resources are synchronized over at once, but with a possible time delay. By the design of the underlying implementation deadlocks by competing synchronization statements are thus avoided²⁵.

As a result of the fact that generally the access to shared resources is resource-centric, a value wrapped in a shared resource which in turn contains nested shared resources is independently protected from the latter. Therefore a shared resource of a struct with a shared member **b** is independently synchronized from **b**:



3.3.2 Translation of shared types

In order to fully understand the translation of synchronization statements the translation of shared types is given first. For the implementation of shared types in C two main solutions are conceivable which differ in the coupling that they exhibit between the data that is to be shared and the additional data required for access restriction, i.e. synchronization. In any case a solution must make use of additional data that can be used to synchronize two threads which try to read or write the shared data. To this end the most basic synchronization primitive, the mutex was chosen: each protected data item is assigned exactly one.

In the first solution the data to be shared is stored as if no protection scheme existed, at all. Additionally all mutexes that are created by the application are stored in one global map which indexes each mutex by the memory address of its corresponding shared datum. This approach offers the advantage that access to the data itself is not influenced by the mutex protection: Every reference to the value of a shared resource **e.get** can directly be translated into a reference to the wrapped value. Additionally, since the mutexes are globally managed all data that is returned by a library can be easily made (pseudo-) synchronization safe: E.g. if a pointer to an arbitrary memory location *loc* is returned the pointed-to address can be used to create a new mutex and add a mapping to the global mutex map. However, this on-the-fly protection of memory locations can incur synchronization leaks: The compiler cannot guarantee that addresses returning functions with unknown implementation will not leak their returned values to some other computation which accesses the according memory unsynchronized. This implies that such protection would only be safe if any reference to *loc* was wrapped in some shared resource which in this scenario is not feasible. Hence, the design was chosen to not allow for such protection and consequently a global map would not be beneficial in this regard. A map solution would entail a space-time tradeoff. For illustrative purposes consider Google's C++ *dense_hash_map*²⁶ which provides comparatively fast access to its members but imposes additional memory requirements to slower hashmap implementations²⁷. An issue of hashmaps is the non-deterministic performance²⁸ and increased access time that may be induced by hash collisions[20].

The second solution for the implementation of shared types keeps each shareable datum and its mutex together: An instance of a struct with member fields for both components is used in place of the bare datum to be shared. In contrast to the aforementioned solution a reference to the value **e.get** needs one level of indirection via the struct

²⁵ Nevertheless deadlocks can obviously still occur if nested synchronization statements compete for the same resources in an unsorted order.

²⁶ http://goog-sparsehash.sourceforge.net/doc/dense_hash_map.html

²⁷ <http://incise.org/hash-table-benchmarks.html>

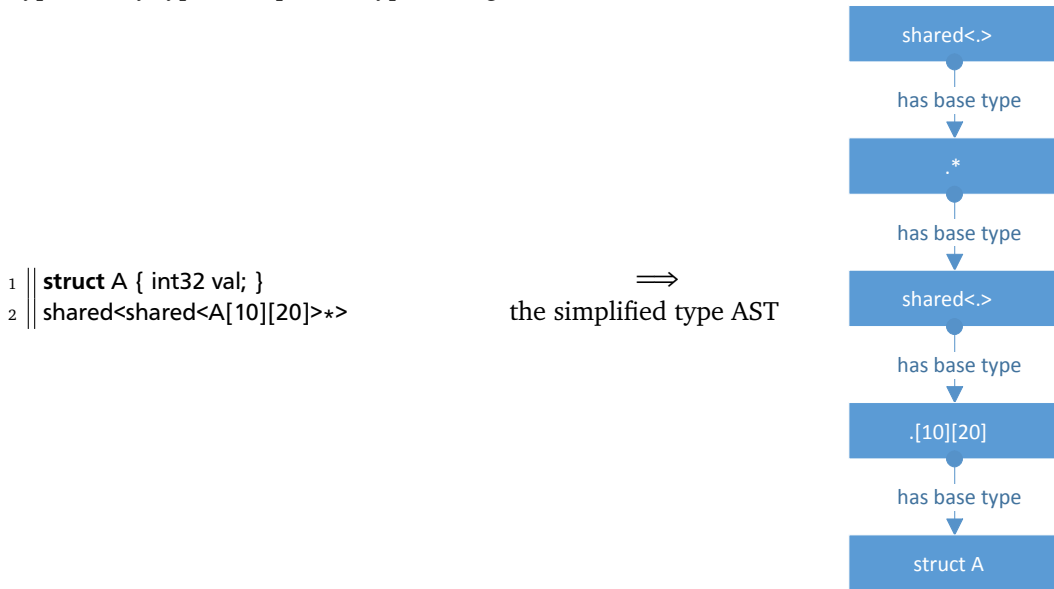
²⁸ Although real-time applications with their time-wise constraints are not a primary concern of this work it should nevertheless be kept in mind that they are part of the embedded domain. Therefore a solution that considers the characteristics of this domain is at least regarded advantageous.

instance. On the contrary the access to a mutex is simplified. As with the value it can be retrieved by a member access to the corresponding struct field whereas the map solution requires a map lookup to get the mutex (plus additional delay to make any modifying access to the map thread-safe). The computational overhead imposed by a struct member lookup is deemed negligible in the overall application performance. On the other hand the space required for the struct equals that of the individual fields plus additional padding[23, pp. 303 ff.] that is required to arrange the struct fields along valid memory addresses. The latter depends on the size of the data to be stored. In order to keep the padding as small as possible smart member ordering should be applied.

For this work the struct solution was chosen in order to keep the access time of mutex lookups small and deterministic while not imposing too much space overhead and datum lookup overhead. For each kind of shared type **shared<t>** with the translated basetype **t'** a separate struct is generated:

```
1 struct SharedOf_t {
2     pthread_mutex_t mutex;
3     t' value;
4 }
```

Any nested shared types are, hence, translated first. For implementation reasons nested *typedefs* and constants used in array types are also resolved in this process²⁹. Depending on the base type of the shared type the generated struct declaration is stored either in a generic module or in a newly generated other module: Consider the tree for a type **shared<t>** whose nodes are made of types and whose edges are formed by the base type relationship of shared types, array types and pointer types³⁰, e.g:



If the leave of the tree is a primitive C type the struct declaration is stored in a generic module. In any other case the leave must be some struct type **s**. In order to preserve the visibility of the corresponding struct in the newly generated struct **SharedOf_t** the latter is stored in the same module. Since the generation of code related to shared types can diminish the legibility of the resulting code profoundly for every such module a specific *SharedTypes_X* module is created and imported into the module that declares **s**. *SharedTypes_X* is used to store struct declarations like **SharedOf_t**. Furthermore **s** is lifted into it in order to make it visible in the member declaration **value** of **SharedOf_t**. For any field of **s** whose type tree contains another user-defined struct type the corresponding struct declaration is either lifted, as well (and recursively treated in the same way) or imported by its module. Should this separation of generated code used for shared type declarations and other user-defined code not proof well in praxis it could easily be deactivated. With the former generation of the struct declaration in mind a type **shared<t>** is reduced to:

```
1 || SharedOf_t
```

²⁹ Although the resolution of typedefs and constants impedes corresponding (testwise) adjustment made by the programmer in the translated C code this is not deemed an issue since generally changes should always be made from within mbeddr, i.e. on the original code.

³⁰ Clearly, this tree will have only one branch.

The reduction of an expression **e.get** and **e.set(f)** make use of the **value** field of the **SharedOf_t** struct. They are basically reduced to a retrieval of, respectively an assignment to the field:

1 || **e'.value** respectively 1 || **e'.value = f'**

If **e'** is has a pointer type the expressions **e->get** and **e->set(f)** are reduced to:

1 || **e'->value** respectively 1 || **e'->value = f'**

The mutex of the struct **SharedOf_t** in the equally named struct field **mutex** which is used to synchronize one variable of an according type must be initialized prior to any usage. This is done implicitly by generated code in order to free the programmer from this task. Accordingly, mutexes must be released before they get out of scope in order to prevent memory leaks. In the generated code both functionalities make use of corresponding functions, i.e. for every type who is one of the following types a pair of **mutexInit-mutexDestroy** functions is generated:

- shared types whose base types are shared types or for whom mutex functions are recursively generated;
- array types who are not base types of array types themselves and whose base types are either shared types or struct types for whom mutex functions are recursively generated;
- struct types whose structs contain at least one field with a type for which the same relation holds as for the aforementioned types.

For example, a type **shared<int32>[42][24]** would enforce the generation of one mutex initialization and one mutex destruction function. **shared<int32>*** on the other hand would not which makes sense as any variable **v** of this type would only point to a shared ressource which must be referenced directly by another variable **v'** of type **shared<int32>** or be contained in the memory-addressable value of some variable **v''** of a more complex type. The declaration of **v'**, respectively **v''** would then trigger the initialization of the mutex of the shared ressource that **v** points to. The resulting mutex initialization functions for types of the aforementioned kind are declared as follows:

```
1 // for a proper shared type shared<t> and the type t' that t is reduced to
2 void mutexInit_X(SharedOf_t* var) {
3     pthread_mutex_init(&var->mutex, &mutexAttribute);
4
5     // either if t is a shared type or a struct type:
6     mutexInit_X'(&var->value);
7
8     // or if t is an array type t[i_1]...[i_n] of 1 to n dimensions:
9     mutexInit_X'((SharedOf_t'...*var->value, i_1, ..., i_n);
10 }
```

For a shared ressource first the mutex of the corresponding translated struct is initialized by a call to the POSIX function **pthread_mutex_init()** which takes a mutex pointer and a mutex attribute pointer³¹. Then—since the shared ressource contains by aforementioned conditions another shared ressource—a call to the appropriate mutex function for the contained value is triggered. Depending on whether the base type of the current shared type is an array additional the dimension sizes for this base type may need to be provided as well (see below for details).

```
1 // for a proper array type t[...] of 1 to n dimensions where ... denotes the occurrence of accordingly many symbols
2 // and t' denotes the reduced-to type
3 void mutexInit_X(t'*...* var, int32 size_0, ..., int32 size_n) {
4     for (int32 __i_0 = 0; __i_0 < size_0; __i_0++) {
5         ...
6         for (int32 __i_n = 0; __i_n < size_n; __i_n++) {
7             // in case t is a struct type
8             mutexInit_X'(&var[__i_0]...[__i_n]);
9
10            // or, in case t is a shared type with generic C base type:
11            pthread_mutex_init(&var[__i_0]...[__i_n].mutex, &mutexAttribute);
12        }
13        ...
14    }
15 }
```

³¹ The meaning of the mutex attribute will be explained later on.

For shared resources that are nested in arrays a nested iteration over all elements of the corresponding possibly multidimensional array with calls to either the generated mutex functions or the function defined by the POSIX standard are triggered.

For structs with nested shared resources for each field that is or contains a shared resource either the **pthread_mutex_init()** function is called directly or the mutex initialization is done by a call to the already generated function that is type compatible with this field (by possibly providing additional array dimensions):

```

1 // for a proper struct type t of a struct t { u_1 f_1; ...; u_n f_n } and according reduced field types u_1' to u_n'
2 void mutexInit_X(SharedOf_t* var) {
3     ...
4     // in case u_i demands further initialization and it is a struct type or a shared type
5     mutexInit_X'(&var->f_i);
6
7     // in case u_i demands further initialization and it is an array type u_i[j_1]...[j_n] of 1 to n dimensions:
8     mutexInit_X'((SharedOf_u_i'*)var->f_i, j_1, ..., j_n);
9
10    // or, in case u_i is a shared type with generic C base type:
11    pthread_mutex_init(&var->f_i.mutex, &mutexAttribute);
12    ...
13 }
```

The signature of **mutexInit_X()** for arrays shows that those are not passed as arrays to the mutex functions but as pointers. This is due to the necessity of declaring multidimensional arrays at least partially with the size for each dimension (e.g. **int[][]** would be missing at least one dimension size). Nevertheless it would not make sense to declare one mutex function for each shape of dimension size. Since arrays are treated like pointers internally when they are passed as function arguments it is completely safe to cast them to appropriate pointer types and to equally type the according function parameters. The deletion of mutexes is defined quite similar to the initialization with the main difference that the utilized according pthreads function only takes one mutex. Therefore only the deletion for mutexes nested in resources of shared types is shown:

```

1 void mutexDestroy_X(SharedOf_t* var) {
2     pthread_mutex_destroy (&var->mutex);
3     // ... further call to a mutexDestroy_X function equivalently to mutexInit_X shown above
4 }
```

The presented functions are used to initialize mutexes at the begin of their life span and delete them right before the corresponding end. For mutexes referred to by global variables this means that they must be initialized at the beginning of the entry function of the program³². As forced by mbeddr the programmer thus has to specify a main function in one of the implementation modules. Similarly mutexes of local variables are initialized right after their declaration whereas mutexes of function arguments are declared at the beginning of the related function³³. The deletion of mutexes for shared resources must be accomplished before they get out of scope which, again, depends on the kind of variables they are referred to.

TODO: Markus fragen, ob immer per main-Fkt. eingestiegen wird und ob Bibliotheksverwendung auch moeglich ist

The mutexes of local variables must be destroyed before they get out of scope. Hence, mutex destruction calls are added at the end of their surrounding scopes if there is no control flow breaking statement (*return*, *break*, *continue*, *goto*) as such cases are covered by the following rules. Take any control flow breaking statement *c* that occurs in

³² Due to the way mutexes are used in ParallelMbeddr (recursively—look below—and nested in structs) and the peculiarities of C and the POSIX standard there is no way to combine the definition and the initialization of mutexes.

³³ The possibility to have function arguments which contain or are shared resources and, hence mutexes is not an obvious design choice: C's pass-by-value semantics of function parameters causes parameters to be copied into functions. Therefore a shared resource which is not passed by its addresses but by its actual value will provoke the generation of an equal shared resource at the beginning of the function execution. This copy is synchronization-wise completely unrelated to the original shared resource since by the mutex copies cannot be used to lock with their origins[1]. Furthermore they have to be initialized and destroyed separately. The use of shared resources in such a manner can confuse programmers who are not aware of this fact. Nevertheless ParallelMbeddr allows this kind of utilization of shared resources in order to not burden the programmer with having to copy large structs that contain shared resources component-wise if the shared resource data is not of relevance. Depending on the feedback of future users it should be considered whether warnings for unintended misuse of shared resources in this way might be helpful.

the AST of the same function as the declaration l of some local variable does which refers to a (nested) shared resource. If c is part of the AST of any statement that follows l ³⁴ and

- c is a *return* statement and refers to a function or a closure whose AST contains l ;
- c is a *break* statement and refers to a loop or a *switch* statement case whose AST contains l ;
- c is a *continue* statement and refers to a loop whose AST contains l ;
- c is a *goto* statement and refers to a label outside the AST of any statement that follows l

c must be preceded with a destruction call of the mutex of the shared resource of l (compare with the synchronization stopping rules below). The proper destruction of a mutex of an argument of function f on the other hand just requires according function calls at the end of f and before any return statement that refers to f . Like inside the declarations of the mutex destruction functions explained above the actual function to call for a variable or argument is either one of the **mutexDestroy_X()** functions or a direct call of **pthread_mutex_destroy()** for “simple” shared resources of generic C types, e.g.:

<pre> 1 // simple shared resource 2 shared<int32> v1; 3 4 // complex shared resource 5 shared<int32>[2][3] v2;</pre>	⇒	<pre> 1 SharedOf_int32_0 v1; 2 pthread_mutex_init(&v1.mutex, &mutexAttribute); 3 4 SharedOf_int32_0[2][3] v2; 5 mutexInit_0((SharedOf_int32_0**)v2, 2, 3);</pre>
--	---	--

To recap: The mutex of a shared resource is either directly initialized and destroyed via appropriate pthreads functions or it is indirectly handled via functions that are based on the types of the values that shared resources are nested inside. This approach was chosen in order to reduce the amount of code duplication that would otherwise occur if mutexes of shared resources would be handled inline for every according variable. As a result the generated code’s readability is enhanced. The additional computational overhead due to function calls and returns should be regarded as an optimization concern of a further compilation step by a compiler like gcc

TODO: or: by the compiler gcc

• **TODO:** Prüfen, ob mutex-init-Reihenfolge richtig ist

ParallelMbeddr does not prevent the programmer from structuring the synchronization statements in such a way that a task will synchronize a shared variable multiple times (*recursive synchronization*). The following code depicts such behavior:

```

1 shared<int32> sharedValue;
2 sync(sharedValue) {
3     sync(sharedValue) {
4         sharedValue.set(42);
5     }
6 }
```

Since each synchronization statement locks the mutexes of the referred shared resources (see below for details) a recursive synchronization results in a *recursive lock* of the corresponding mutex. Mutexes as defined by the POSIX standard must be specifically initialized in order to allow for this behavior³⁵: A mutex attribute that specifies the recursiveness must be defined and initialized first. It can then be used by arbitrarily many mutexes. For this purpose an application-wide attribute is defined in a generic module that is imported by all user-defined modules. It is initialized at the beginning of the main function:

```

1 // inside the generic module:
2 pthread_mutexattr_t mutexAttribute
3 // at the beginning of main:
4 pthread_mutexattr_init(&mutexAttribute);
5 pthread_mutexattr_settype(&mutexAttribute, PTHREAD_MUTEX_RECURSIVE);
```

³⁴ In other words: Consider only those control flow breaking statements cs that are either one of the statements $stmts$ which have the same AST parent p and follow l in the statement list of p or are contained in the AST of some $stmt$

³⁵ By default a recursive lock results in undefined behaviour because a default mutex does not have a lock count which is required to make recursive locks work: http://linux.die.net/man/3/pthread_mutex_trylock

3.3.3 Translation of synchronization statements

Every synchronization statement is reduced to its statement list—as a block—surrounded with calls to functions that control the synchronization of the mutexes. The reduction for such a statement is given by either

$$1 \parallel \text{sync}(e) \text{ stmt_list} \quad \Rightarrow \quad \begin{array}{l} 1 \parallel \text{startSyncFor1Mutex}(\&e.\text{mutex}); \\ 2 \parallel \text{stmt_list}' \\ 3 \parallel \text{stopSyncFor1Mutex}(\&e.\text{mutex}); \end{array}$$

in case it contains only one synchronized resource, or else by

$$1 \parallel \text{sync}(e_1, \dots, e_n) \text{ stmt_list} \quad \Rightarrow \quad \begin{array}{l} 1 \parallel \text{startSyncForNMutexes}(\&e_1.\text{mutex}, \dots, \&e_n.\text{mutex}); \\ 2 \parallel \text{stmt_list}' \\ 3 \parallel \text{stopSyncForNMutexes}(\&e_1.\text{mutex}, \dots, \&e_n.\text{mutex}); \end{array}$$

The statements are kept inside their statement list block in order to keep the scope of local variables inside synchronization statements. A synchronization statement list block is reduced to another block where statements that break the program flow structure may be preceded by an identical call of the **stopSyncForNMutexes()** function as is present after the list: Let s be a synchronization statement and c be a control flow breaking statement which is nested on some level in the AST of s' statement list. Then c is preceded with a call to **stopSyncForNMutexes()** if one of the following cases hold:

c is a *return* statement and refers to a function or a closure whose AST contains s ;

c is a *break* statement and refers to a loop or a *switch* statement case whose AST contains s ;

c is a *continue* statement and refers to a loop whose AST contains s .

c is a *goto* statement and refers to a label outside the AST of s .

In this manner inconsistent synchronization states of shared resources due to a control flow break by the aforementioned statements are omitted. Since the occurrence of such a statement may also force ParallelMbeddr to insert **mutexDestroy_X()** calls a careless mixture of mutex unlocking and destruction calls can cause runtime errors[1]. The generator therefore takes care of putting any destruction calls behind the generated unlocking calls.

For each arity of synchronization resources separate versions of the **start-** and **stopSyncForNMutexes()** functions are declared inside a generic C module. A **stopSyncForNMutexes()** function straightforwardly redirects its mutex parameters to calls of the **pthread_mutex_unlock** function:

```
1 // the corresponding function 'stopSyncFor1Mutex()' for exactly one mutex is skipped, here
2 void stopSyncForNMutexes(pthread_mutex_t* mutex_1, ..., pthread_mutex_t* mutex_n) {
3     pthread_mutex_unlock (mutex_1);
4     ...
5     pthread_mutex_unlock (mutex_n);
6 }
```

Abstracted over the details of the actual implementation, synchronization statements synchronize their resources atomically as was mentioned in the preceding design section. Since one or more mutexes can be tentatively locked by multiple threads simultaneously specific contention management has to be taken care of. The illusion of atomic synchronization is realized by an implementation of the obstruction-free³⁶ busy-wait protocol *Polite*. In order to resolve conflicts *Polite* uses exponential backoff. The according backoff function is explained further down. The synchronization function tries to lock every mutex as given by its arguments. On failure it releases every mutex

³⁶ Busy-waiting means that the thread will repeatedly test a condition until it is met, without doing actual useful work[28, p. 166]. Thus, it is an alternative to suspending a thread and revoking it later on when some condition is met (which can, e.g., be realized by *condition variables* as provided by POSIX threads[5, p. 77]). Obstruction-free means that the execution of any thread which is at some time run in isolation such that the execution of obstructing other threads is interrupted meanwhile will progress. The existence of obstruction-freedom guarantees that no deadlocks will occur[2]. However, livelocks and starvation are not necessarily avoided. Stronger degrees of non-blocking algorithms like lock-freedom and wait-freedom tackle these problems (partially) but are not relevant for this work. Further information on the latter is for instance provided by <http://preshing.com/20120612/an-introduction-to-lock-free-programming/>.

that was locked so far, uses the backoff function to delay its execution for a randomized amount of time and repeats afterwards. This scheme enables competing threads to (partially) proceed and avoid deadlocks due to unordered overlapping mutex locks³⁷.

```

1 // again, the equivalent function declaration for 1 mutex is skipped
2 void startSyncForNMMutexes(pthread_mutex_t* mutex_0, ..., pthread_mutex_t* mutex_m, pthread_mutex_t* mutex_n) {
3     uint8 waitingCounter = 0;
4     uint16 mask = 16;
5     uint32 seed = (uint32)(uintptr_t) &waitingCounter;
6
7     while (true) {
8         if ([| pthread_mutex_trylock (mutex_0) |] != 0) {
9             backoffExponentially(&waitingCounter, &mask, &seed);
10        }
11        else if ([| pthread_mutex_trylock (mutex_1) |] != 0) {
12            [| pthread_mutex_unlock (mutex_0) |];
13            backoffExponentially(&waitingCounter, &mask, &seed);
14        } ...
15        else if ([| pthread_mutex_trylock (mutex_n) |] != 0) {
16            [| pthread_mutex_unlock (mutex_m) |];
17            ...
18            [| pthread_mutex_unlock (mutex_0) |];
19            backoffExponentially(&waitingCounter, &mask, &seed);
20        }
21        else {
22            break;
23        }
24    }
25 }

```

The backoff realized by Polite randomizedly delays the execution by less than $limit = 2^{n+k}$ ns[33]. n denotes the retry counter and k denotes some constant offset which can be machine-tuned. The randomized wait time of the exponential backoff is used to avoid livelocks which could happen if two threads would repeatedly compete for the same resources and delay their execution for equal amounts of time. In the current implementation **backoffExponentially()** of the contention management the offset k is set to 4 and a threshold m of 17 denotes the number of rounds after which k is reset³⁸. Thus, maximum delays of about 100 ms (specifically 131 ms) are allowed³⁹.

```

1 inline void backoffExponentially(uint8* waitingCounter, uint16* mask, uint32* seed) {
2     *mask |= 1 << *waitingCounter;
3     randomWithXorShift(seed);
4     struct timespec sleepingTime = (struct timespec){ .tv_nsec = *seed & *mask };
5     nanosleep(&sleepingTime, null);
6     *waitingCounter = (*waitingCounter + 1) % 13;
7 }

```

Note that **backoffExponentially()** keeps its main state inside the **startSyncForNMMutexes()** function. The state will therefore be re-initialized before the execution of every synchronization block. The generation of the pseudo-randomized delay is realized via utilization of the Marsaglia's Xorshift random number generator[24]:

```

1 void randomWithXorShift(uint32* seed) {
2     *seed ^= *seed << 13;
3     *seed ^= *seed >> 17;
4     *seed ^= *seed << 5;
5 }

```

³⁷ Again, the presented scheme does not prevent the programmer from nesting the synchronization statements in such a manner that deadlocks in nested synchronization statements occur. It is rather a prevention of deadlocks that are caused solely by synchronization statements on the same nesting level.

³⁸ k 's value is reflected in the initial value ($2^4 = 16$) of **mask** whereas m 's value is composed of mask's base and the divisor (13) in the calculation of the next **waitingCounter**.

³⁹ The search for machine- or application-specific optimal offsets and thresholds is a task for future enhancements of ParallelMbeddr.

The generator was chosen for its high performance, low memory consumption and thread safety due to the utilization of the stack-managed **seed** parameter as opposed to the global state usage of the standard C random generator **rand()**. The fact that after a certain number—which may be smaller than in other generators—of repeated calls with the same seed value (i.e. the same memory address of a seed) repetitions of the sequence of calculated numbers will occur is not of relevance for the purpose of this work.

3.3.4 Example code

In the previous sections 3.1.3 and 3.2.3 the running example approximated the number π by using tasks which calculate exactly one fraction of the result each and retrieving their results via futures. The amount of work was therefore partitioned in advance. In this section a more dynamic approach is chosen instead: The work of every task comprises major and minor rounds. A minor round is equivalent to the full calculation loop in the previous π solution. In every step of a major round initiates a minor round by first coordinating with the other tasks which range of π it should calculate. After having calculated the sum in a minor round the task then uses a queue to store its next partial result. A dedicated task is used to collect these partial results from the queue and accumulate them to a complete sum which in the end is the result of the over-all approximated π . The communication-based solution can be seen as a map-reduce implementation where partial results are mapped onto the queue by a certain number of tasks and from there reduced to a final result by a separate task (compare with [8]). In order to understand the new implementation first consider that a thread-safe queue of a certain size and slots of type **long double** is given as a blackbox. Further suppose that functions for the initialization, for adding a value and getting the next value (or wait for the next value) exist:

```
1 struct Queue {...}
2 void queueInit(shared<Queue>* queue);
3 void queueSafeAdd(shared<Queue>* queue, long double item);
4 void queueSafeGet(shared<Queue>* queue, long double* result);
```

Like in the previous approach the amount of work to be done is defined by a range size (number of minor task rounds) and the number of ranges altogether. Additionally the number of mapper tasks is set to a certain value that should be in the order of the number of processors:

```
1 #constant RANGESIZE = 300000000;
2 #constant RANGECOUNT = 4;
3 #constant THRESHOLD = RANGESIZE * RANGECOUNT;
4 #constant MAPPERCOUNT = 2;
```

These constants are used to initialize the mappers and the reducer in the main function appropriately:

```
1 exported int32 main(int32 argc, string[] argv) {
2     shared<Queue> queue;
3     queueInit(&queue);
4     shared<Queue>* queuePointer = &queue;
5
6     shared<uint32> counter;
7     shared<uint32>* counterPointer = &counter;
8     sync(counter) { counter.set(0); }
9
10    Task<void> mapperTask = |map(THRESHOLD, counterPointer, queuePointer)|;
11    Future<void>[MAPPERCOUNT] mappers;
12    for (i ++ in [0..MAPPERCOUNT]) {
13        mappers[i] = mapperTask.run;
14    }
15    mapperTask.clear;
16
17    shared<long double> result;
18    shared<long double>* resultPointer = &result;
19
20    |reduce(RANGECOUNT, resultPointer, queuePointer)|.run.join;
21
22    return 0;
23 }
```

First the queue is defined as being shared in order to be accessible by all tasks. After the initialization a pointer to the queue is created which is necessary since any other reference inside a task expression to the queue variable would otherwise cause a copy of the queue struct instance into the task, regardless of whether later on the address of the referred value is retrieved via the `&` operator. This ‘shortcoming’ of the current semantics could be addressed by the introduction of a new address operator that creates a temporary variable of the addressed value and should be considered for future extensions of ParallelMbeddr. Similar to the queue a **counter** variable is introduced which will be used by the tasks to check and communicate how many ranges have been processed so far. Then one mapper task is defined and initialized with a task expression of a call to the **map()** function. For communication purposes every it gets the queue pointer and the counter pointer. Additionally, although due to the use of constants not necessary in the current example every mapper is told the maximum number of items that need to be calculated. This single task template is used to create multiple running tasks and save their handles as futures in a **mappers** array. The reducer uses a pointer to a result value memory location to save its result. Further it gets access to the queue via a pointer thereof and is told how many items (**RANGECOUNT**) it shall read from the queue before termination. The code shows the first example for a task chain: First the task is declared, then an instance of it is run in parallel and immediately the main task joins the reducer. Since between the run and the join nothing is done in the main task the serialized execution could also be realized by a simple function call to **reduce()**. For demonstrative purposes the code was chosen this way nevertheless. The main task joins solely the reducer task since after its termination every mapper task will also be finished. The **map()** function iteratively calculates complete ranges of fractions of π until the maximum number of items as given by **threshold** is reached:

```

1 void map(uint32 threshold, shared<uint32>* counter, shared<Queue>* resultQueue) {
2     while (true) {
3         uint32 start;
4         uint32 end;
5
6         sync(counter) {
7             start = counter->get();
8             if (start == threshold) {
9                 break;
10            }
11            uint32 possibleEnd = start + RANGESIZE;
12            end = (possibleEnd <= threshold)?(possibleEnd):(threshold);
13            counter->set(end);
14        }
15
16        queueSafeAdd(resultQueue, calcPiBlock(start, end));
17    }
18 }

```

In every iteration the function synchronizes the shared resources that **counter** points to in order to retrieve its value and increment it by the number of items that **map()** is going to calculate in the current round. It uses the **calcPiBlock()** function that was presented in section 3.1.3 to calculate a partial sum. Afterwards the result is added to the queue. The **reduce()** function uses the queue to iteratively read all partial results and updates the value of the shared resource of the final result accordingly:

```

1 void reduce(uint32 numberOfItems, shared<long double>* finalResult, shared<Queue>* partialResultQueue) {
2     sync(finalResult) {
3         for (uint32 i = 0; i < numberOfItems; ++i) {
4             long double item;
5             queueSafeGet(partialResultQueue, &item);
6             finalResult->set(item + finalResult->get());
7         }
8     }
9 }

```

During the whole calculation **reduce()** synchronizes the result variable in order to keep the synchronization overhead small. It is able to do this because no other task will try to access the result before the termination of the single reducer task.

In the beginning of the translated main function the global mutex attribute is initialized which will be reused for every mutex. Afterwards the declared queue is initialized:

```

1 | pthread_mutexattr_settype(&mutexAttribute_0, PTHREAD_MUTEX_RECURSIVE);
2 | pthread_mutexattr_init(&mutexAttribute_0);
3 | initAllGlobalMutexes_0();
4 | SharedOf_Queue_0 queue;
5 | mutexInit_2(&queue);
6 | queueInit(&queue);
7 | SharedOf_Queue_0* queuePointer = &queue;

```

The translated struct type **SharedOf_Queue_0** of the original type **shared<Queue>** refers to a struct that contains a field for the protected queue and a mutex field:

```

1 | struct SharedOf_Queue_0 {
2 |     pthread_mutex_t mutex;
3 |     Queue value;
4 | };

```

The mutex initialization function **mutexInit_2()** initializes the mutex of the shared queue and calls another initialization function which takes care of initializing any nested mutexes of the **Queue** struct. Similarly a destruction function is generated:

```

1 | void mutexInit_2(SharedOf_Queue_0* var) {
2 |     pthread_mutex_init(&var->mutex, &mutexAttribute_0);
3 |     mutexInit_1(&var->value);
4 | }
5 | void mutexDestroy_2(SharedOf_Queue_0* var) {
6 |     pthread_mutex_destroy(&var->mutex);
7 |     mutexDestroy_1(&var->value);
8 | }

```

Similar to the declarations for the queue the mbeddr generator creates declarations of a struct for the **result** variable. The initialization and destruction of **result** is accomplished inline by calls to the pthreads functions since there exist no nested mutexes for the fields of the struct:

```

1 | struct SharedOf_long_double_0 {
2 |     pthread_mutex_t mutex;
3 |     long double value;
4 | };
5 | ... // in main()
6 | SharedOf_long_double_0 result;
7 | pthread_mutex_init(&result.mutex, &mutexAttribute_0);
8 | SharedOf_long_double_0* resultPointer = &result;

```

Lastly the **counter** variable shows how translated shared resources can be synchronized⁴⁰. For the duration of the setting of **counter**'s value its mutex is locked. The setting of the value is done by an assignment to the **value** field of the generated struct that keeps the value of the shared resource:

```

1 | SharedOf_uint32_0 counter;
2 | pthread_mutex_init(&counter.mutex, &mutexAttribute_0);
3 | SharedOf_uint32_0* counterPointer = &counter;
4 |
5 | startSyncFor1Mutex(&counter.mutex);
6 | { counter.value = 0; }
7 | stopSyncFor1Mutex(&counter.mutex);

```

The mutexes of all shared resources are destroyed at the end of the main function. Although this should not be necessary for local variables of the entry function of the program the compiler currently does not distinguish between the main function and any other function for which such calls would be necessary **TODO: maybe fix this**:

```

1 | mutexDestroy_2(&queue);
2 | pthread_mutex_destroy(&counter.mutex);
3 | pthread_mutex_destroy(&result.mutex);

```

⁴⁰ The declarations of the struct and the mutex functions for it are skipped, here

The initializations of the tasks and the declarations of the futures is quite similar to those in section 3.1.3:

```
1 Task mapperTask = taskInit_0(queuePointer, counterPointer);
2 // mappers:
3 VoidFuture[MAPPERCOUNT] mappers;
4 for (int8 __i = 0; __i < MAPPERCOUNT; __i++) {
5     mappers[__i] = runTaskAndGetVoidFuture(mapperTask);
6 }
7 free(mapperTask.args);
8
9 // reducer:
10 saveAndJoinVoidFuture(futureInit_0(resultPointer, queuePointer));
```

Since in the new solution the tasks do not return any results directly, the **VoidFuture** struct and respective functions are used. For the lack of valuable new insight the declarations of the **taskInit_0()** function and **Args_X** structs is skipped, here. Instead recall the original code **|reduce(RANGECOUNT, resultPointer, queuePointer)|.run.join**. As was shown in section 3.2.2 the run call and the task declaration are reduced to a call of a function which combines the initialization of a task with the one of a future of a parallel running instance of this task. This call is reflected by **futureInit_0(resultPointer, queuePointer)**. Furthermore the join of the task must first bind the created future handle to some addressable location before it can use this handle by its address. For this reason **saveAndJoinVoidFuture** is used in place of **joinVoidFuture**. The declaration of **futureInit_0** follows:

```
1 VoidFuture futureInit_0(SharedOf_long_double_0* resultPointer, SharedOf_Queue_0* queuePointer) {
2     Args_1* args_1 = malloc(sizeof(Args_1));
3     args_1->resultPointer = resultPointer;
4     args_1->queuePointer = queuePointer;
5     pthread_t pth;
6     pthread_create (&pth, null, :parFun_1, args_1) |];
7     return (VoidFuture){ .pth = pth };
8 }
```

The original declaration of the reducer task contains references to the local variables **resultPointer** and **queuePointer** which is why they are bound to equally named fields in the argument struct. An instance of the **VoidFuture** struct is returned as the parallel task does not return a value and, thus, had the type **task<void>**.

Inside the helper function **map()** the synchronization statement of **counter** is replaced by its statement list which is surrounded by calls to appropriate functions:

```
1 while (true) {
2     uint32 start;
3     uint32 end;
4
5     startSyncFor1Mutex(&counter->mutex);
6     {
7         start = counter->value;
8         if (start == threshold) {
9             stopSyncFor1Mutex(&counter->mutex);
10            break;
11        }
12        uint32 possibleEnd = start + RANGESIZE;
13        end = (possibleEnd <= threshold)?(possibleEnd):(threshold);
14        counter->value = end;
15    }
16    stopSyncFor1Mutex(&counter->mutex);
17
18    queueSafeAdd(resultQueue, calcPiBlock(start, end));
19 }
```

The break statement is preceded by another call to the synchronization stop function as the function would otherwise return in a state where **counter** would still be locked which would ultimately cause the program to fail. The former expressions to get and set the value of **counter** are translated to accesses of the translated **value** field of the **counter** struct instance. The synchronization statement of **reduce()** is likewise translated to its state-

ment list with surrounding synchronization calls. The translation of the expression **finalResult->set(item + finalResult->get)** contains two accesses to the aforementioned **value** field, one for **.set** and one for **.get**:

```
1 startSyncFor1Mutex(&result->mutex);
2 {
3     for (uint32 i = 0; i < numberOfItems; ++i) {
4         long double item;
5         queueSafeGet(resultQueue, &item);
6         result->value = item + result->value;
7     }
8 }
9 stopSyncFor1Mutex(&result->mutex);
```

3.4 Safety measures

So far the basic blocks that constitute parallel code execution and shared data synchronization, namely tasks, shared resources and synchronization thereof, were introduced. Still missing are most of the rules which ensure that only shared resources may be shared and that these can only be used in a sane way. The current section fills this gap by giving an informal overview of the rules that were implemented in ParallelMbeddr, categorized by their objectives. In the following paragraphs **t** denotes some arbitrary type.

3.4.1 Avoidance of implicitly shared unprotected data

Global variables can be accessed by any function for which they are visible. Therefore they have to have any type **shared<t>** in order to restrict any modifications of their values to synchronized contexts. This restriction can be too strong if a global variable is only accessed by exactly one thread. Nevertheless the conservative approach was chosen in order to establish a safe foundation. Future static code analysis should be leveraged to reliably detect the cases where restrictions can be loosened. Another class of data that is inherently vulnerable for unsafe data sharing arises from static variables. In C local variables that are declared static have a “global lifetime”[26, p. 439] which means that like global variables the addresses of their allocated memory does not change. Thus, they keep their values from one function call to the next. The main difference of static local variables to global variables is the respective visibility. Consequently, static variables must have a type **shared<t>**, as well. Finally, a base type **t** of a shared type may never be a pointer type with a base type other than a shared type. Otherwise the value of a shared resource would point to data that is not synchronized and would enable unprotected inter-task communication. E.g. in the following example any access the functions **foo()** and **bar()** do not block one another since they synchronize over different shared resources. Nevertheless they both write to the same location in memory which causes a data race.

```
1 // global variables:
2 shared<int32*> v1;
3 shared<int32*> v2;
4
5 int32 main(int32 argc, string[] argv) {
6     int32 sharedValue;
7     sync(v1, v2) {
8         v1.set(&sharedValue);
9         v2.set(&sharedValue);
10    }
11    |foo()|.run;
12    |bar()|.run;
13 }
14
15 foo() {
16     sync(v1) { *v1.get = 0; }
17 }
18 bar() {
19     sync(v2) { *v2.get = 1; }
20 }
```

3.4.2 Copying pointers to unshared data into tasks

The pass-by-value semantics of C generally already ensure that any local data which is referred from within a task expression is safely copied into the task. On execution the task will not access the original data but a copy thereof. This approach on the other hand becomes unsafe as soon as local variables are copied whose values are plain pointers: When such a copied pointer is used inside a task to access a pointed-to memory location in an unsynchronized manner, it accesses data that might simultaneously be accessed by another task, e.g. the task by which this task was created, who knows the address of the data. To avoid this behavior every pointer that might be copied into a task by accessing a local variable or a function argument from within a task expression must point to a shared resource, i.e. must be of type `shared<t>*`. Note that this does not only hold for the variables themselves but also for e.g. nested fields of struct instances and array elements. Furthermore arrays must not be copied into tasks unless they are wrapped in a struct field. Due to the internal treatment of pointers in C (see section 2.11) the access to an array holding local variable would cause a copy of the address of the array into the task as a pointer. In consequence, references of local variables and arguments with type `t[]...[]` inside task expressions are not allowed. On the other hand it is safe completely safe to have a struct be copied into a task also if the struct contains an array field. In contrast to the former case the array would then be entirely copied along its surrounding struct instance.

3.4.3 Unsynchronized access to synchronizable data

As was already mentioned in section 3.3 the value of a shared resource can only be accessed (retrieved or rewritten) from within a proper synchronization context. This approach ensures that no write to shared data invalidates any other write or read of the data. The according rule is that an expression `e.get` or `e.set` is only allowed if `e` is either a reference to a named resource in scope, i.e. a shared resource which is synchronized in a surrounding synchronization statement and bound to a new name; or if `e` is a reference to a variable with a shared resource as value which is also referred to by a synchronization reference of a surrounding synchronization statement. By this restriction the following code would trigger an error message in ParallelMbeddr:

```
1 | shared<shared<int32>> v;  
2 | sync(v) {  
3 |     sync(v.get) {  
4 |         // error: e.get seems to be unsynchronized  
5 |         e.get.set(0);  
6 |     }  
7 | }
```

Although the previous code would not produce any synchronization gap ParallelMbeddr does not recognize this since the expression `e.get` or `.set` does not refer to a named resource or a variable with a synchronized shared resource. Instead for exactly this purpose named resources were implemented which allow to rewrite the code in the following valid manner:

```
1 | shared<shared<int32>> v;  
2 | sync(v) {  
3 |     sync(&(v.get) as w) {  
4 |         w->set(0);  
5 |     }  
6 | }
```

The reasoning behind this was to ease the implementation of the safety checking analysis. Again, the chosen approach can in certain cases be overly conservative. If write conflicts can never happen for a shared resource and, in consequence, data races thereof are impossible it would be safe to access the variable outside any synchronization context despite the error message that is generated by the IDE. Moreover, by the sole resolution by name ParallelMbeddr is not able to detect whether a shared resource is recursively synchronized across multiple function calls:

```
1 | shared<int32> v;  
2 | sync(v) { foo(&v); }  
3 | ...  
4 |
```

```

5 | void foo(shared<int32>* v) {
6 |     sync(v) { v->set(0); }
7 | }

```

Both problems should be addressed by static analysis in order to (partially) detect such cases. The second problem could further be addressed by the introduction of a *synced* type for shared resources that are already synchronized in the current scope.

3.4.4 Address leakage of shared resource values

In order to restrict any writes or reads of the values of shared resources to synchronization contexts it is crucial to not leak the memory addresses of these values outside the protected synchronization context from where they could be accessed via the address operator (**&**). The measures to keep the addresses encapsulated constrain the use of the address operator and the use of arrays: The first rule forbids any expressions **&e** where **e** contains an expression sub path **eSub.get** and **e** does not evaluate to a shared resource. The latter condition allows the programmer to get the address of an encapsulated shared resource which is fine since shared resources may not be overwritten as is explained in the next section 3.4.5. The second rule states that an expression **e** of some array type contains an expression sub path **eSub.get** and the parent of **e** does not access a specific element of **e** is not allowed. Thus, any access to an (multidimensional) array that is encapsulated in a shared resource must actually be extended to an access of the innermost element of the array. Otherwise it would be possible to assign the array itself or in the case of a multidimensional array take an element of the array which itself is an array and assign it to a unprotected pointer variable. Hence, the address of the array or of an element of this array would be leaked. For instance in the following example `ParallelMbeddr` would complain about an address leakage of the first element of the array wrapped inside **v**:

```

1 | shared<int32[5][10]> v;
2 | int32* pointer;
3 | // address leakage!
4 | sync(v) { pointer = v.get[0]; }

```

3.4.5 Overwriting shared resources

Shared resources may never be overwritten. The reason for this regulation results from the following consideration. If a shared resource **r** shall be overwritten, e.g. by a direct assignment or by a **set** if **r** is nested inside another shared resource, it must be synchronized first since the overwriting could overlap with another access to **r** from within another task. Before the resource is rewritten the mutex of **r** must be destroyed in order to prevent memory leaks. Furthermore after the rewriting is done the newly created mutex for **r** must be initialized prior to any usage. Hence, in the meantime between the destruction and the re-initialization the mutex of **r** (respectively, **r'** after the rewrite is done, because the variable will refer to a new value) cannot be accessed in the synchronization attempt of any other simultaneously executed synchronization statement. Any such task would thus have to be blocked which would complicate the compiler and decrease the performance of the executed code without offering any worthy advantage. In addition to this problem the overwriting of a shared resource of a struct instance that itself contains a shared resource field **f** would invalidate any pointer **p** to **f**. **p** could therefore not be used anymore afterwards which on the hand goes with the usual C semantics but does not fit the safety-first approach of `ParallelMbeddr`. Nevertheless it is safe to copy a shared resource into the memory of a local variable declaration or of a function argument, i.e. a pass of a shared resource to a function or an initialization of a newly created local variable with a shared resource is valid: In these cases the shared resources can only be used after their initializations with shared resource copies. The safety enforcing rules are as follows: A variable that refers to a shared resource a value that contains a nested shared resource may not be assigned a new value (the initialization of a declaration is not a classical assignment). Additionally an expression **e.set(e')** is not allowed if **e'** is a shared resource or contains a shared resource because **e.set(e')** is translated into an assignment (see section ??).

4 Evaluation

- show if and how objectives mentioned in the introduction are met by the implementation
- use illustrating examples

Literaturverzeichnis

- [1] The Open Group pthread_mutex_destroy, pthread_mutex_init - destroy and initialize a mutex. http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_mutex_destroy.html. Accessed: 2014-07-15.
- [2] David Basin, Samuel J Burri, and Günter Karjoth. Obstruction-free authorization enforcement: Aligning security with business objectives. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 99–113. IEEE, 2011.
- [3] L.W. Baugh and University of Illinois at Urbana-Champaign. *Transactional Programmability and Performance*. University of Illinois at Urbana-Champaign, 2008.
- [4] T.D. Brown. *C for BASIC Programmers*. Silicon Press, 1987.
- [5] D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley professional computing series. Addison-Wesley, 1997.
- [6] G.C. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency: Predictability vs. Efficiency*. Series in Computer Science. Springer, 2006.
- [7] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/software Approach*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Morgan Kaufmann Publishers, 1999.
- [8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [9] D.M. Dhamdhere. *Operating Systems: A Concept-based Approach, 2E*. McGraw-Hill Higher Education, 2006.
- [10] E.H. D'Hollander, G.R. Joubert, F. Peters, and U. Trottenberg. *Parallel Computing: Fundamentals, Applications and New Directions: Fundamentals, Applications and New Directions*. Advances in Parallel Computing. Elsevier Science, 1998.
- [11] I.A. Dhotre. *Operating Systems*. Technical Publications, 2007.
- [12] Martin Fowler. Language workbenches: The killer-app for domain specific languages? <http://www.martinfowler.com/articles/languageWorkbench.html>. Published: 2005-06-12.
- [13] J.M. Garrido, R. Schlesinger, and K. Hoganson. *Principles of Modern Operating Systems*. Jones & Bartlett Learning, 2011.
- [14] R. Guerraoui, M. Kapalka, and N. Lynch. *Principles of Transactional Memory*. Synthesis lectures on distributed computing theory. Morgan & Claypool Publishers, 2010.
- [15] S. Haldar and A.A. Aravind. *Operating Systems*. Pearson Education, 2009.
- [16] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.
- [17] T. Harris, J.R. Larus, and R. Rajwar. *Transactional Memory*. Synthesis lectures in computer architecture. Morgan & Claypool, 2010.
- [18] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI'73*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
- [19] A. Holub. *Taming Java Threads*. Apresspod Series. Apress, 2000.

-
- [20] Kun Huang, Gaogang Xie, Rui Li, and Shuai Xiong. Fast and deterministic hash table lookup using discriminative bloom filters. *Journal of Network and Computer Applications*, 36(2):657–666, 2013.
- [21] B. Lewis and D.J. Berg. *Multithreaded Programming with Java Technology*. Sun Microsystems Press Java series. Prentice Hall, 2000.
- [22] C.S. LLC, M.L. Mitchell, A. Samuel, and J. Oldham. *Advanced Linux Programming*. Pearson Education, 2001.
- [23] R. Love. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, 2013.
- [24] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.
- [25] Robin Milner. The polyadic pi-calculus: a tutorial. Technical report, Logic and Algebra of Specification, 1991.
- [26] A. Mittal. *Programming In C: A Practical Approach*. Pearson Education, 2010.
- [27] Robert H. B. Netzer, Timothy W. Brennan, and Suresh K. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '96, pages 31–40, New York, NY, USA, 1996. ACM.
- [28] P. Pacheco. *An Introduction to Parallel Programming*. An Introduction to Parallel Programming. Elsevier Science, 2011.
- [29] Bhatt P.C.P. *Introduction To Operating Systems: Concepts And Practice An 2Nd Ed*.
- [30] S. Prata. *C Primer Plus*. Pearson Education, 2013.
- [31] J. Protic, M. Tomasevic, and V. Milutinović. *Distributed Shared Memory: Concepts and Systems*. Systems Series. Wiley, 1998.
- [32] R. Reese. *Understanding and Using C Pointers*. O'Reilly Media, 2013.
- [33] William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005.
- [34] J. Shirazi. *Java Performance Tuning*. Java Series. O'Reilly Media, Incorporated, 2003.
- [35] C. Steven Hernandez. *Official (ISC)2 Guide to the CISSP CBK, Second Edition*. (ISC)2 Press. Taylor & Francis, 2009.
- [36] Markus Völter. Generic tools, specific languages.
- [37] E. White. *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly Media, 2011.