Embedded Multicore Building Blocks Tutorial

Contents

Introduction

Overview

The Embedded Multicore Building Blocks (EMB^2) are an easy to use yet powerful and efficient C/C++ library for the development of parallel applications. EMB^2 has been specifically designed for embedded systems and the typical requirements that accompany them, such as real-time capability and constraints on memory consumption. As a major advantage, low-level operations are hidden in the library which relieves software developers from the burden of thread management and synchronization. This not only improves productivity of parallel software development, but also results in increased reliability and performance of the applications.

EMB^2 is independent of the hardware architecture (x86, ARM, ...) and runs on various platforms, from small devices to large systems containing numerous processor cores. It builds on MTAPI, a standardized programming interface for leveraging task parallelism in embedded systems containing symmetric or asymmetric (heterogeneous) multicore processors. A core feature of MTAPI is low-overhead scheduling of fine-grained tasks among the available cores during runtime. Unlike existing libraries, EMB^2 supports task priorities and affinities, which allows the creation of soft real-time systems. Additionally, the scheduling strategy can be optimized for non-functional requirements such as minimal latency and fairness.

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

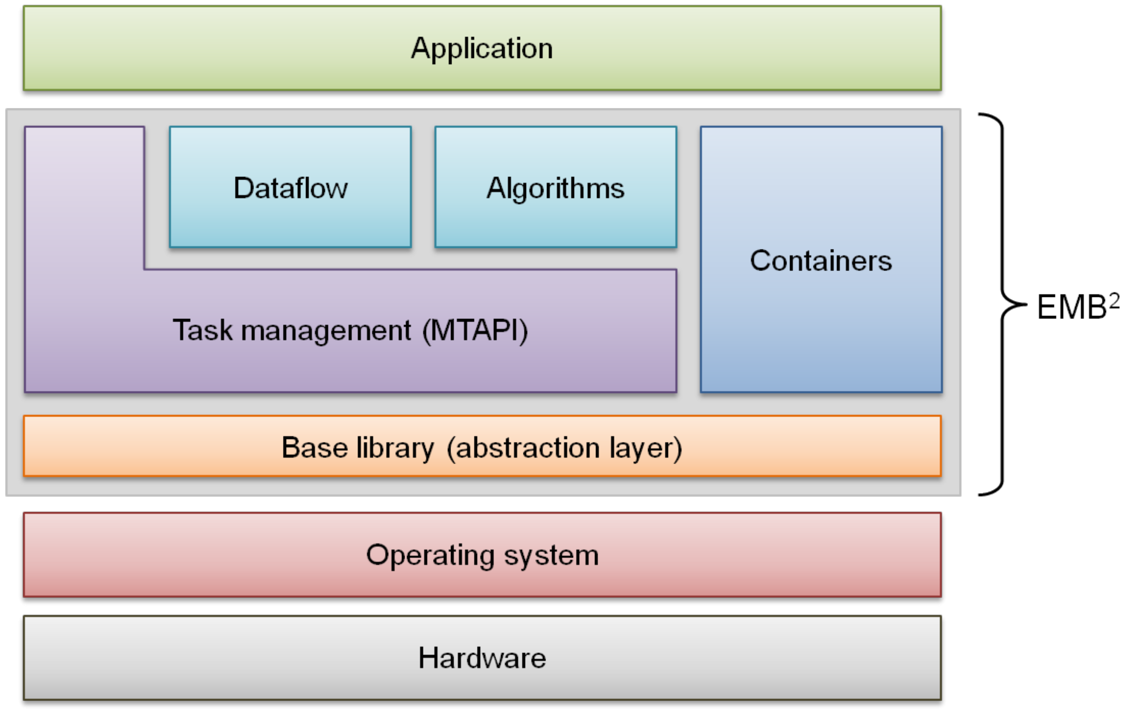

Figure 1: Main Building Blocks of EMB²

Outline

The purpose of this document is to introduce the basic concepts of EMB² and to demonstrate typical application scenarios by means of simple examples. The tutorial is not intended to be complete in the sense that it describes every feature of EMB². For a detailed description of the API, please see the reference manual.

In the next subsection, we briefly describe the concept of function objects which is essential for using EMB². We then present various parallel algorithms and the dataflow framework. After that, we explain the usage of MTAPI and how to leverage heterogeneous systems. The complete source code for the examples presented in the following can be found in the `examples` directory.

Functions, Functors, and Lambdas

Throughout this tutorial, we will encounter C++ types which model the C++ concept `FunctionObject`. The function object concept comprises function pointer, functor, and lambda types that are callable with suitable arguments by the function call syntax. Given a function object `f` and arguments `arg1`, `arg2`, `...`, the expression `f(arg1, arg2, ...)` is a valid function invocation. If you are already familiar with function objects, you can safely skip the rest of this section. Otherwise, it might be worth reading it to get an idea of what is meant when talking about a function objects.

Consider, for example, the transformation of an iterable range of data values. Specifically, consider a vector of integers initialized as follows:

```
std::vector<int> range(5);
for (size_t i=0; i < range.size(); i++) {
  range[i] = static_cast<int>(i) + 1;
}
```

The range consists of the values `(1, 2, 3, 4, 5)`. To double each value, we could simply iterate over the vector as follows:

```
for (size_t i=0; i < range.size(); i++) {
  range[i] *= 2;
}
```

The range then contains the values `(2, 4, 6, 8, 10)`. In order to demonstrate the concept of function objects, we are now going to use the `std::for_each` function defined in the `algorithm` header of the C++ Standard Library. This function accepts as argument a `UnaryFunction`, that is, a function object which takes only one argument. In case of `std::for_each`, the argument has to have the same type as the elements in the range, as these are passed to the unary function. In our example, the unary function's task is to double the passed value. We could define a function for that purpose:

```
void DoubleFunction(int& to_double) {
  to_double *= 2;
}
```

Since a function pointer models the concept of function objects, we can simply pass `&DoubleFunction` to `std::for_each`:

```
std::for_each(range.begin(), range.end(), &DoubleFunction);
```

Another possibility is to define a functor

```
struct DoubleFunctor {
  void operator()(int& to_double) {
    to_double *= 2;
  }
};
```

and to pass an instance of this class to `std::for_each`:

```
std::for_each(range.begin(), range.end(), DoubleFunctor());
```

Functors as well as function pointers separate the actual implementation from its usage which can be helpful if the functionality is needed at different places. In many cases, however, it is easier to have the implementation of the function object at the same place as it is used. C++11 provides lambda expressions for that purpose which make our example more concise:

```
std::for_each(range.begin(), range.end(),
              [] (int& to_double) { to_double *= 2; });
```

Of course, this example is too simple to really benefit from function objects and the algorithms contained in the C++ Standard Library. However, in combination with the parallelization features provided by EMB^2, function objects are very useful. Within this document, whenever a function object or one of its subtypes is required, one can use a function pointer, a functor, or a lambda. For simplicity, we will restrict ourselves to lambdas in subsequent examples, as they are most suitable for this tutorial.

Algorithms

The *Algorithms* building block of EMB^2 provides high-level constructs for typical parallelization tasks. They are similar to the functions provided by the C++ Standard Library, but contain additional functionality typical for embedded systems such as task priorities. Although the algorithms can be used in a black-box way, it is good to have a basic understanding of their implementation: The algorithms split computations to be performed in parallel into tasks which are executed by the MTAPI task scheduler (see chapter on MTAPI). For that purpose, the tasks are stored in queues and mapped to a fixed number of worker threads at runtime.

Note: *The algorithms are implemented using the MTAPI C++ interface. Since MTAPI allocates the necessary data structures during initialization, the maximum number of tasks in flight is fixed. In case one of the algorithms exceeds this limit, an exception is thrown. By calling* ***`embb::mtapi::Node::Initialize`****, the maximum number of tasks and other limits can be customized. Explicit initialization also eliminates unexpected delays when measuring performance. See the section on the MTAPI C++ Interface for details.*

Function Invocation

Let us start with the parallel execution of several work packages encapsulated in functions. Suppose that the following functions operate on different data sets and are thus independent of each other:

```
void WorkPackageA();
void WorkPackageB();
void WorkPackageC();
```

The functions can be executed in parallel using the `ParallelInvoke` construct provided by EMB²:

```
using embb::algorithms::Invoke;
Invoke(WorkPackageA, WorkPackageB, WorkPackageC);
```

Note that `ParallelInvoke` waits until all its arguments have finished execution.

Next, let us consider a more elaborate example. The following piece of code shows a sequential implementation of the quicksort algorithm, which we want to parallelize (do not care about the details of the `Partition` function for the moment):

```
void QuickSort(int* first, int* last) {
  if (last - first <= 1) return;
  int* mid = Partition(first, last);
  QuickSort(first, mid);
  QuickSort(mid, last);
}
```

A straightforward approach to parallelize this algorithm is to execute the recursive calls to `Quicksort` in parallel. With `ParallelInvoke` and lambdas, it is as simple as that:

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

The lambdas capture the `first`, `mid`, and `last` pointers to the range to be sorted and forward them to the recursive calls of quicksort. These are executed in parallel, where `Invoke` does not return before both have finished execution. The above implementation of parallel quicksort is not yet optimal. In particular, the creation of new tasks should be stopped when a certain lower bound on the size of the subranges has been reached. The subranges can then be sorted sequentially in order to reduce the overhead for task creation and management. Fortunately, EMB² already provides solutions for parallel sorting, which will be covered in the following section.

Sorting

For systems with constraints on memory consumption, the quicksort implementation provided by EMB² is usually the best choice, since it works in-place, which means that it does not require additional memory. Considering real-time systems, however, its worst-case runtime of $O(n^2)$, where n is the number of elements to be sorted, can be a problem. For this reason, EMB² also provides a parallel merge sort algorithm. Merge sort does not work in-place, but has a predictable runtime complexity of $(n \log n)$. Assume we want to sort a vector of integers:

```
std::vector<int> range;
```

Using quicksort, we simply write:

```
using embb::algorithms::QuickSort;
QuickSort(range.begin(), range.end());
```

The default invocation of `QuickSort` uses `std::less` with the iterators' `value_type` as comparison operation. As a result, the range is sorted in ascending order. It is possible to provide a custom comparison operation, for example `std::greater`, by passing it as a function object to the algorithm. Sorting the elements in descending can be accomplished as follows:

```
QuickSort(range.begin(), range.end(), std::greater<int>());
```

The merge sort algorithm comes in two versions. The first version automatically allocates dynamic memory for temporary values when the algorithm is called. Its name is `MergeSortAllocate` and it has the same parameters as `QuickSort`. To enable the use of merge sort in environments that forbid dynamic memory allocation after initialization, the second version can be called with a pre-allocated temporary range of values:

```
using embb::algorithms::MergeSort;
std::vector<int> temporary_range(range.size());
MergeSort(range.begin(), range.end(), temporary_range.begin());
```

The temporary range can be allocated at any time, e.g., during the initialization phase of the system.

Counting

EMB² also provides functions for counting the number of elements in a range. Consider a range of integers from 0 to 3:

```
int range[] = {0, 3, 2, 0, 1, 1, 3, 2};
```

To determine how often a specific value appears within the range, we could simply iterate over it and compare each element with the specified one. The `Count` function does this in parallel, where the first two arguments specify the range and the third one the element to be counted:

```
std::iterator_traits<int*>::difference_type count;
using embb::algorithms::Count;
count = Count(range, range + 8, 1);
```

For the range given above, we have `count == 2`.

In case the comparison operation is not equality, we can employ the `CountIf` function. Here, the third argument is a unary predicate which evaluates to `true` for each element to be counted. The following example shows how to count the number of values greater than 0:

```
using embb::algorithms::CountIf;
count = CountIf(range, range + 8,
                [](const int& value) -> bool { return value > 0; });
```

Foreach Loops

A frequently encountered task in parallel programming is to apply some operation to a range of values, as illustrated previously. In principle, one could apply the operation to all elements in parallel provided that there are no data dependencies. However, this results in unnecessary overhead if the number of elements is greater than the number of available processor cores p. A better solution is to partition the range into p blocks and to process the elements of a block sequentially. With the `ForEach` construct provided by EMB^2, users do not have to care about the partitioning, since this is done automatically. Similar to the Standard Library's `for_each` function, it is sufficient to pass the operation in form of a function object. The following piece of code shows how to double the elements of a range in parallel:

```
using embb::algorithms::ForEach;
ForEach(range.begin(), range.end(),
        [] (int& to_double) { to_double *= 2; });
```

There is also a `ForLoop` variant that accepts integers as limits of the range:

```
using embb::algorithms::ForLoop;
ForLoop(0, int(range.size()),
  [&](int to_double) { range[size_t(to_double)] = (to_double + 1) * 2; });
```

In the above code snippet, the results of the computation overwrite the input. If the input has to be left unchanged, the results must be written to a separate output range. Thus, the operation requires two ranges. EMB^2 supports such scenarios by the `ZipIterator`, which wraps two iterators into one. Consider the following revised example:

```
std::vector<int> input_range(5);
for (size_t i=0; i < input_range.size(); i++) {
  input_range[i] = static_cast<int>(i) + 1;
}
std::vector<int> output_range(5);
```

Using the `Zip` function as a convenient way to create a zip iterator, the doubling of elements can be performed as follows:

```
using embb::algorithms::Zip;
using embb::algorithms::ZipPair;
ForEach(Zip(input_range.begin(), output_range.begin()),
        Zip(input_range.end(), output_range.end()),
        [] (ZipPair<int&, int&> pair) {
          pair.Second() = pair.First() * 2;
        });
```

The argument to the lambda function is a `ZipPair` with the iterators' reference value as template parameters. The elements pointed to by the zip iterator can be accessed via `First()` and `Second()`, similar to `std::pair`.

Reductions

As mentioned in the previous section, the `ForEach` construct requires the loop iterations do be independent of each other. However, this is not always the case. Imagine we want to sum up the values of a range, e.g., a vector of integers:

```
std::vector<int> range(5);
for (size_t i = 0; i < range.size(); i++) {
  range[i] = static_cast<int>(i) + 1;
}
```

Sequentially, this can be done by a simple loop:

```
int sum = 0;
for (size_t i = 0; i < range.size(); i++) {
  sum += range[i];
}
```

One might be tempted to sum up the elements in parallel using a foreach loop. The problem is that parallel accesses to `sum` must be synchronized to avoid race conditions, which in fact sequentializes the loop. A more efficient approach is to compute intermediate sums for each block of the range and to sum them up at the end. For such purposes, EMB^2 provides the function `Reduce`:

```
using embb::algorithms::Reduce;
sum = Reduce(range.begin(), range.end(), 0, std::plus<int>());
```

The third argument to `Reduce` is the neutral element of the reduction operation, i.e., the element that does not change the result. In case of addition (`std::plus`), the neutral element is 0. If we wanted to compute the product of the vector elements, the neutral element would be 1.

Next, let us consider the parallel computation of a dot product. Given two input ranges, we want to multiply each pair of input elements and sum up the products. The second input range is given as follows:

```
std::vector<int> second_range(5);
for (size_t i = 0; i < range.size(); i++) {
  second_range[i] = static_cast<int>(i) + 5;
}
```

The reduction consists of two steps: First, the input ranges are transformed and then, the reduction is performed on the transformed range. For that purpose, the `Reduce` function allows to specify a transformation function object. By default, this is the identity functor which does not modify the input range. To implement the dot product, we can use the `Zip` function and a lambda function for computing the transformed range:

```
using embb::algorithms::Zip;
using embb::algorithms::ZipPair;
int dot_product = Reduce(Zip(range.begin(), second_range.begin()),
                         Zip(range.end(), second_range.end()),
                         0,
                         std::plus<int>(),
                         [](const ZipPair<int&, int&>& pair) {
                           return pair.First() * pair.Second();
                         });
```

Prefix Computations

Prefix computations (or scans) can be viewed as a generalization of reductions. They transform an input range x_i into an output range y_i with $i=1,\dots,n$ such that

$$y_0 = id \cdot x_0$$
$$y_1 = y_0 \cdot x_1$$

$$y_i = y_{i-1} \cdot x_i$$

$$y_n = y_{n-1} \cdot x_n,$$

where id is the identity (neutral element) with respect to the $\cdot$ operation. As an example, consider the following range:

```
std::vector<int> input_range(5);
for (size_t i = 0; i < input_range.size(); i++) {
  input_range[i] = static_cast<int>(i) + 1;
}
```

Computing the prefix sums of `input_range` sequentially is easy:

```
std::vector<int> output_range(input_range.size());
output_range[0] = input_range[0];
for(size_t i = 1; i < input_range.size(); i++) {
  output_range[i] = output_range[i-1] + input_range[i];
}
```

Note the dependency on loop iteration *i-1* to compute the result in iteration *i*. A special two-pass algorithm is used in the function `Scan` to perform prefix computations in parallel. Using `Scan` to compute the prefix sums, we get:

```
using embb::algorithms::Scan;
Scan(input_range.begin(), input_range.end(), output_range.begin(),
     0, std::plus<int>());
```

As in the case of reductions, the neutral element has to be given explicitly. Also, a transformation function can be passed as additional argument to `Scan`. The elements of the input range are then transformed before passed to the prefix operation.

Dataflow

EMB^2 provides generic skeletons for the development of parallel stream-based applications. These skeletons are based on dataflow networks, a model of computation widely employed in different domains like digital signal processing and imaging due to its simplicity and flexibility. As a major advantage, these networks are deterministic which significantly simplifies testing and debugging. This is particularly important in embedded systems, where high demands are put on correctness and reliability. Moreover, they are inherently parallel and lend themselves well for execution on a multicore processor. In fact, they can be viewed as a generalization of pipelining, a frequently encountered parallel pattern.

Note: *Dataflow networks are internally implemented using the MTAPI C++ interface. Since MTAPI does not allocate memory after initialization, the number of tasks and other resources are limited. By calling `embb::mtapi::Node::Initialize`, these limits can be customized. Explicit initialization also eliminates unexpected delays when measuring performance. See the section on the MTAPI C++ Interface for details.*

Linear Pipelines

Before we go into detail, we demonstrate the basic concepts of dataflow networks by means of a simple application which finds and replaces strings in a file. Let us start with the sequential implementation. The program shown in Listing 1 reads a file line by line and replaces each occurrence of a given string with a new string.

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>

using namespace std;

// replace all ocurrences of 'what' in 'str' with 'with'
void repl(string& str, const string &what,
          const string& with) {
  string::size_type pos = 0;
  while((pos = str.find(what, pos)) != string::npos) {
    str.replace(pos, what.length(), with);
    pos += with.length();
  }
}

int main(int argc, char *argv[]) {
  // check and read command line arguments
  if(argc != 4) {
    cerr << "Usage: replace <what> <with> <file>" << endl;
    exit(EXIT_FAILURE);
  }
  const string what(argv[1]), with(argv[2]);

  // open input file
  ifstream file(argv[3]);
  if(!file) {
    cerr << "Cannot open file " << argv[3] << endl;
    exit(EXIT_FAILURE);
  }

  // read input file line by line and replace strings
  string str;
  while(getline(file, str)) {
    repl(str, what, with);
    cout << str << endl;
  }

  // close file and exit
  file.close();
  exit(EXIT_SUCCESS);
}
```

Listing 1: Sequential program for replacing strings in a file

The main part consists of the `while` loop which performs three steps:

1. read a line from `file` and store it in the string `str`
2. replace each occurrence of `what` in `str` with `with`
3. write the resulting string to `cout`

To run this program on a multicore processor, we may execute the above steps in a pipelined fashion. In this way, a new line can be read from the hard disk while the previous one is still being processed. Likewise, processing a string and writing the result to standard output can be performed in parallel. Thus, the pipeline may consist of three stages as depicted in Figure 2.

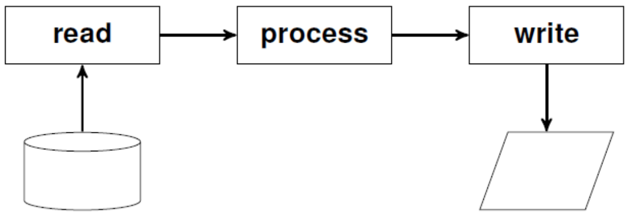

Figure 2: Pipeline for replacing strings in a file

This pipeline can be easily implemented using the dataflow networks. As the first step, we have to include the `dataflow.h` header file:

```
#include <embb/dataflow/dataflow.h>
```

Then, we have to construct a network. A network consists of a set of processes that are connected by communication channels. EMB^2 provides a class `Network` that handles data routing and scheduling of your processes:

```
typedef embb::dataflow::Network Network;
```

We need to prepare the network for the desired maximum number of elements that can be in the network at a time. The number of elements is limited to avoid that the network is flooded with new elements before the previous elements have been processed. In a linear pipeline, for example, this may happen if the source is faster than the sink. For our example we assume that at most four elements may be processed simultaneously: one in the source, one in the sink, and two in the middle stage.

```
Network network(4);
```

Finding an optimal value depends on the application and usually requires some experimentation. In general, large values boost the throughput but also increase the latency. Conversely, small values reduce the latency but may lead to a drop of performance in terms of throughput.

As the next step, we have to construct the processes shown in Figure 2. The easiest way to construct a process is to wrap the user-defined code in a lambda function and to pass it to the network. The network constructs an object for that process and executes the lambda function whenever new data is available. There are several methods for constructing processes depending on their type. The process **read** is a *source* process, since it produces data (by reading it from the specified file) but does not consume any data. Source processes are constructed from a function object

```
bool SourceFunction(std::string & str) {
  if (!file.eof()) {
    std::getline(file, str);
    return true;
  } else {
    return false;
  }
}
```

like this:

```
Network::Source<std::string> read(
  network, embb::base::MakeFunction(SourceFunction)
);
```

Note the template argument `std::string` to `Source`. This tells that the process has exactly one *port* of type `std::string` and that this port is used to transmit data to other processes. The user-defined code can access the ports via the parameters of the function. Thus, each parameter corresponds to exactly one port. In our example, the result of the process is stored in a variable `str`, which is passed by reference.

The replacement of the strings can be done by a *parallel* process, which means that multiple invocations of the process may be executed simultaneously. In general, processes that neither have any side effects nor maintain a state can safely be executed in parallel. This helps to avoid bottlenecks that arise when some processes are faster than others. Suppose, for example, that **replace** requires up to 50 ms to execute, whereas **read** and **write** each require 10 ms to execute. If only one invocation of **replace** could be executed at a time, the throughput would be at most 20 elements per second. Since **replace** is a parallel process, however, the network may start a new invocation every 10 ms. Hence, up to five invocations may be executed in parallel, yielding a throughput of 100 elements per second. To compensate for variations in the runtime of parallel stages, they may be executed *out-of-order*. As a result, the order in which the elements of a stream enter and leave parallel stages is not necessarily preserved. In our example, the runtime of **replace** may vary significantly due to the fact that not all lines have the same length and that the number of replacements depends on the content. Before we go into more detail, let us first consider the following function

```
void ReplaceFunction(std::string const & istr, std::string & ostr) {
  ostr = istr;
  repl(ostr, what, with);
}
```

and how to construct the corresponding **replace** process:

```
Network::ParallelProcess<
  Network::Inputs<std::string>,
  Network::Outputs<std::string> > replace(
    network, embb::base::MakeFunction(ReplaceFunction)
  );
```

The template parameter `Network::Inputs<std::string>` specifies that the process has one port serving as input. Analogously, `Network::Outputs<std::string>` specifies that there is one port serving as output.

Since the last process (**write**) does not have any outputs, we make it a *Sink*. Unlike parallel processes, sinks are always executed *in-order*. EMB^2 takes care that the elements are automatically reordered according to their original order in the stream. This way, the externally visible behavior is preserved even if some parallel stages may be executed out-of-order. The function

```
void SinkFunction(std::string const & str) {
  std::cout << str << std::endl;
}
```

is used to construct the sink:

```
Network::Sink<std::string> write(
  network, embb::base::MakeFunction(SinkFunction)
);
```

Note: *If you parallelize an application using EMB^2 and your compiler emits a lengthy error message containing lots of templates, it is very likely that for at least one process, the ports and their directions do not match the signature of the given function.*

As the last step, we have to connect the processes (ports). This is straightforward using the C++ stream operator:

```
read >> replace >> write;
```

Then, we can start the network:

```
network();
```

Note that you will probably not observe a speedup when you run this program on a multicore processor. One reason for this is that I/O operations like reading a file from the hard disk and writing the output to the screen are typically a bottleneck. Moreover, the amount of work done in the middle stage of the pipeline (**replace**) is rather low. To outweigh the overhead for parallel execution, the amount of work must be much higher. In image processing, for example, a single pipeline stage may process a complete image. To sum up, we have chosen this example for its simplicity, not for its efficiency.

Nonlinear Pipelines

Some applications exhibit a more complex structure than the linear pipeline presented in the previous section. Typical examples are applications where the result of a pipeline stage is used by more than one successor stage. Such pipelines are said to be nonlinear. In principle, every nonlinear pipeline can be transformed to a linear one as depicted in Figure 3. However, this increases the latency and complicates the implementation due to data that must be passed through intermediate stages. In Figure 3, for example, the data transferred from stage A to stage C must be passed through stage B in the linearized implementation.

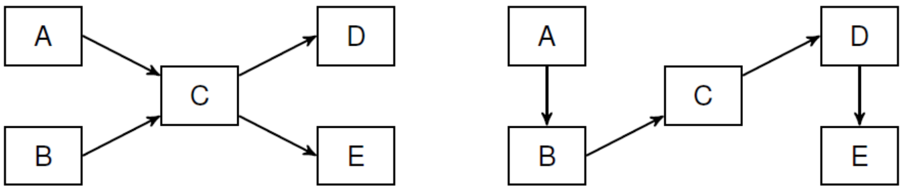

Figure 3: Nonlinear pipeline and linearized variant

Nonlinear pipelines can be implemented as they are using EMB², i.e., there is need not linearize them. As an example, let us consider the implementation of a sorting network. Sorting networks consist of a set of interconnected comparators and are used to sort sequences of data items. As depicted in Figure 4, each comparator sorts a pair of values by putting the smaller value to one output, and the larger one to the other output. Thus, a comparator can be viewed as a switch that transfers the values at the inputs to the outputs, either directly or by swapping them (cf. Figure 5).

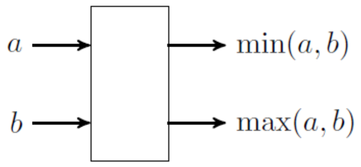

Figure 4: Block diagram of a comparator

Figure 5: Example for the operating principle of a comparator

Figure 6 shows a sorting network with four inputs/outputs and five comparators. The numbers at the interconnections exemplify a "run" of the network. As can be seen from Figure 6, the comparators *C₁*-*C₄* "sink" the largest value to the bottom and "float" the smallest value to the top. The final comparator C_5 simply sorts out the middle two values. This way it is guaranteed that the values at the outputs occur in ascending order.

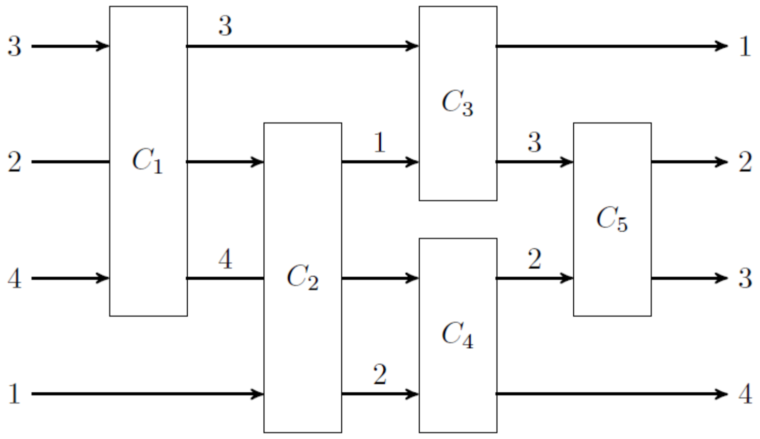

Figure 6: Sorting network with four inputs/outputs and five comparators

Let us now consider the implementation of the sorting network using EMB². As in the previous example, we need three types of processes: one or more sources that produce a stream of data items, a total number of five processes that implement the comparators, and one or more sinks that consume the sorted sequences. The processes should be generic so that they can be used with different types. For example, in one application we might use the network to sort integers, and in another application to sort floating point values.

The following listing shows the implementation of the source processes using classes instead of functions (a complete implementation can be found in the examples directory):

```
template <typename T>
class Producer {
 public:
  explicit Producer(int seed) : seed_(seed), count_(4) {}
  bool Run(T& x) {
    if (count_ >= 0) {
      // produce a new value x
      x = SimpleRand(seed_);
      count_--;
      return true;
    } else {
      return false;
    }
  }

 private:
  int seed_;
  int count_;
};
```

The class-based approach has several advantages besides the use of templates: Firstly, the creation of multiple processes is straightforward. Secondly, one can derive other processes from a given base class such as `Producer`. Thirdly, it eases migration of existing code. For example, if you want to use an object of an existing class `foo` as a process, you might derive a class `bar` from `foo` implementing any missing functionality.

To feed our sorting network `nw` with four streams of integer values, we may write:

```
Producer<int>
  producer1(1),
  producer2(2),
  producer3(3),
  producer4(4);

Network::Source<int>
  source1(
    network,
    embb::base::MakeFunction(producer1, &Producer<int>::Run) ),
  source2(
    network,
    embb::base::MakeFunction(producer2, &Producer<int>::Run) ),
  source3(
    network,
    embb::base::MakeFunction(producer3, &Producer<int>::Run) ),
  source4(
    network,
    embb::base::MakeFunction(producer4, &Producer<int>::Run) );
```

The code for the comparators looks like this:

```
template <typename T>
class Comparator {
public:
  void Run(const T& a, const T& b, T& x, T& y) {
    x = std::min(a,b);
    y = std::max(a,b);
  }
};
```

Since the comparators neither have any side effects nor maintain a state, we allow multiple invocations to be executed in parallel.

To check whether the resulting values are sorted, we use a single sink with four inputs:

```
template <typename T>
class Consumer {
public:
  void Run(const T& x1, const T& x2, const T& x3, const T& x4) {
    if (x1 <= x2 && x2 <= x3 && x3 <= x4) {
      // consume values
    }
  }
};
```

We could also have a sink for each output of the sorting network. There is no restriction on the number of sources and sinks a network may have.

Containers

Containers are essential for storing objects in an organized way. Unfortunately, the containers provided by the C++ Standard Library are not thread-safe. Attempts to read and write elements concurrently may corrupt the stored data. While such undefined behavior can be avoided by synchronizing all accesses using a mutex, this largely limits the available parallelism.

The containers provided by EMB^2 enable a high degree of parallelism by design. They are implemented in a lock-free or wait-free fashion, thus avoiding any blocking operations. This way, multiple threads or tasks may access a container concurrently without suffering from typical side effects like convoying. Wait-free algorithms even guarantee that an operation completes within a bounded number of steps. Consequently, threads are immune to starvation which is critical for real-time systems.

In embedded systems, memory is often preallocated in the initialization phase to minimize the effort for memory management during operation and to prevent unpredictable out-of-memory errors. EMB^2 containers have a fixed capacity and allocate the required memory at construction time. Consequently, they can be used in safety-critical application, where dynamic memory allocation after initialization is forbidden.

Object Pools

An object pool allocates a fixed number of objects at construction. Objects can then be allocated from the pool and returned for later reuse. When implementing lock-free or wait-free algorithms, the underlying memory allocation scheme also has to be lock-free or wait-free, respectively. However, memory allocation functions such as `new` and `delete` usually do not give any progress guarantees. To solve this problem, EMB^2 provides lock-free and wait-free object pools.

Listing 2 shows an example, where we create a pool containing five objects of type `int`. As the second step, we allocate five objects from the pool and store the obtained pointers in a temporary array. Finally, we deallocate them by calling `Free` on each pointer.

```
embb::containers::ObjectPool<int> objPool(5); // create

int* alloc[5];
```

```
for (int i = 0; i != 5; ++i) {
  alloc[i] = objPool.Allocate(); // allocate
}

for (int i = 0; i != 5; ++i) {
  objPool.Free(alloc[i]); // free
}
```

Listing 2: Object pool – initialization, allocation, and deallocation

For allocating and deallocating objects, the object pool's implementation relies on a value pool which keeps track of the objects in use. If the value pool is implemented in a lock-free manner, the object pool is lock-free as well (analogously for wait-free pools). Currently, EMB^2 provides two value pools: `WaitFreeArrayValuePool` and `LockFreeTreeValuePool`. Normally (if nothing else is specified), the wait-free pool is used. For having a lock-free object pool instead, one has to specify the corresponding value pool as additional template parameter. If we replace the first line of the previous example with the following lines, the object pool is not wait-free anymore but lock-free (the values are of type `int` and initialized with `0`):

```
embb::containers::ObjectPool<int,
  embb::containers::LockFreeTreeValuePool< int, 0 >> objPool(5);
```

This will result in a speed-up for most applications, but progress guarantees are weaker.

Stacks

As the name indicates, the class template `LockFreeStack` implements a lock-free stack which stores elements according to the LIFO (Last-In, First-Out) principle. The stack provides two methods, `TryPush` and `TryPop`, both returning a Boolean value indicating success of the operation: `TryPop` returns `false` if the stack is empty, and `TryPush` returns `false` if the stack is full. If successful, `TryPop` returns the element removed from the stack via reference. Listing 3 shows a simple example. First, we create a stack of integers with a capacity of 10 elements (due to necessary over-provisioning of memory in thread-safe memory management, the stack might be able to hold more than 10 elements, but is guaranteed to be able to hold at least 10 elements). Then, we try to pop an element from the empty stack, which has to fail. In the subsequent for-loop, we fill the stack with the values 0...4. Afterwards, we pop five values from the stack into variable `j`. According to the LIFO semantics, the values are popped in reverse order, i.e., we get the sequence 4...0, which is checked by the assertion.

```
embb::containers::LockFreeStack<int> stack(10); // create

int i, j;
bool result = stack.TryPop(i); // fail_pop
assert(result == false);

for (i = 0; i <= 4; ++i) { // loop1
  result = stack.TryPush(i); // push
  assert(result == true);
}

for (i = 4; i >= 0; --i) { // loop2
  result = stack.TryPop(j); // pop
  assert(result == true && i == j); // assert
}
```

Listing 3: Stack - initialization, push, and pop

Queues

There are currently two FIFO (First-In, First-Out) queue implementations in EMB², `LockFreeMPMCQueue` and `WaitFreeSPSCQueue`. The former can deal with multiple producers and multiple consumers (MPMC), whereas the latter is restricted to a single producer and a single consumer (SPSC). The interfaces are the same for both queues. The Boolean return value of the methods `TryEnqueue` and `TryDequeue` indicates success (`false` if the queue is full or empty, respectively).

Listing 4 shows an example for the `LockFreeMPMCQueue`. First, we create a queue with element type `int` and a capacity of (at least) 10 elements. Then, we try to dequeue an element from the empty queue, which has to fail. In the subsequent for-loop, we fill the queue with the values 0…4. Afterwards, we dequeue five values from the queue into variable `j`. According to the FIFO semantics, the values are dequeued in the same order as they were enqueued, i.e., we get the sequence 0…4, which is checked by the assertion.

```
embb::containers::LockFreeMPMCQueue<int> queue(10); // create

int i, j;
bool result = queue.TryDequeue(i); // fail_pop
assert(result == false);

for (i = 0; i <= 4; ++i) { // loop1
  result = queue.TryEnqueue(i); // push
  assert(result == true);
}

for (i = 0; i <= 4; ++i) { // loop2
  result = queue.TryDequeue(j); // pop
  assert(result == true && i == j); // assert
}
```

Listing 4: Queue – initialization, enqueue, and dequeue

MTAPI

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

While task schedulers are nowadays widely employed, especially in desktop and server applications, they are typically limited to a single operating system running on a homogeneous multicore processor. System-wide task management in heterogeneous embedded systems must be realized explicitly with low-level communication mechanisms. MTAPI [1] addresses those issues by providing an API which allows parallel embedded software to be designed in a straightforward way, covering homogeneous and heterogeneous multicore architectures, as well as accelerators such as GPUs or FPGAs. As a major advantage, it abstracts from the hardware details and lets software developers focus on the application. Moreover, MTAPI takes into account typical requirements of embedded systems such as real-time constraints and predictable memory consumption.

The remainder of this chapter is structured as follows: The next section explains the basic terms and concepts of MTAPI as given in the specification [1]. The section on the MTAPI C Interface describes the C API using a simple example taken from [1]. Finally, the section on the MTAPI C++ Interface outlines the use of MTAPI in C++ applications. Note that the C++ interface is provided by EMB^2 for convenience but it is not part of the standard. Readers who are familiar with MTAPI or just want to get impression on how to use MTAPI in heterogeneous systems may skip this chapter on go directly to Heterogeneous Systems or the Tutorial Application.

Foundations

Domains

An MTAPI system is composed of one or more MTAPI domains. An MTAPI domain is a unique system global entity. Each MTAPI domain comprises a set of MTAPI nodes. An MTAPI node may only belong to one MTAPI domain, while an MTAPI domain may contain one or more MTAPI nodes. This allows the programmer to use MTAPI domains as namespaces for all kinds of IDs (e.g., nodes, actions, queues, etc.).

Nodes

An MTAPI node is an independent unit of execution, such as a process, thread, thread pool, processor, hardware accelerator, or instance of an operating system. A given MTAPI implementation specifies what constitutes a node for that implementation.

The intent is to avoid a mixture of node definitions in the same implementation (or in different domains within an implementation). If a node is defined as a unit of execution with its private address space (like a process), then a core with a single unprotected address space OS is equivalent to a node, whereas a core with a virtual memory OS can host multiple nodes.

On a shared memory SMP processor, a node can be defined as a subset of cores. A quad-core processor, for example, could be divided into two nodes, one node representing three cores and one node representing the fourth core reserved exclusively for certain tasks. The definition of a node is flexible because this allows applications to be written in the most portable fashion supported by the underlying hardware, while at the same time supporting more general-purpose multicore and many-core devices.

The definition allows portability of software at the interface level (e.g., the functional interface between nodes). However, the software implementation of a particular node cannot (and often should not) necessarily be preserved across a multicore SoC product line (or across product lines from different silicon providers) because a given node's functionality may be provided in different ways, depending on the chosen multicore SoC.

Tasks

A task represents the computation associated with the data to be processed and is executed concurrently to the code starting it. The main API functions are `mtapi_task_start()` and `mtapi_task_wait()`. The semantics are similar to the corresponding thread functions (e.g., `pthread_create` / `pthread_join` in POSIX Threads). The lifetime of a task is limited; it can be started only once.

Actions

In order to cope with heterogeneous systems and computations implemented in hardware, a task is not directly associated with an entry function as it is done in other task-parallel APIs. Instead, it is associated with at least one action object representing the calculation. The association is indirect: one or more actions implement a job, one job is associated with a task. If the action is implemented in software, this is either a function on the same node (which can represent the same processor or core) or a function implemented on a different node that does not share memory with the core starting the task.

Starting a task consists of three steps:

1. Create the action object with a job ID (software-implemented actions only).

2. Obtain a job reference.
3. Start the task using the job reference.

Synchronization

The basic synchronization mechanism provided by MTAPI is waiting for task completion. Calling `mtapi_task_wait()` with a task handle blocks the current thread or task until the task referenced by the handle has completed. Depending on the implementation, the calling thread can be used for executing other tasks while waiting for the task to be completed. In order to synchronize with a set of tasks, every task can be associated with a task group. The methods `mtapi_group_wait_all()` and `mtapi_group_wait_any()` wait for a group of tasks or completion of any task in the group, respectively.

Queues

Queues are used for guaranteeing sequential order of execution of tasks. A common use case is packet processing in the communication domain: for every connection all packets must be processed sequentially, while the packets of different connections can be processed in parallel to each other.

Sequential execution is accomplished by using a queue for every connection and queuing all packets of one connection into the same queue. In some systems, queues are implemented in hardware, otherwise MTAPI implements software queues. MTAPI is designed for handling thousands of queues that are processed in parallel.

The procedure for setting up and using a queue is as follows:

1. Create the action object (software-implemented actions only).
2. Obtain a job reference.
3. Create a queue object and attach the job to the queue (software-implemented queues only).
4. Obtain a queue handle if the queue was created on a different node, or if the queue is hardware-implemented.
5. Use the queue: enqueue the work using the queue.

Another important purpose of queues is that different queues can express different scheduling attributes for the same job. For example, in contrast to order-preserving queues, non-order-preserving queues can be used for load-balancing purposes between different computation nodes. In this case, the queue must be associated with more than one action implementing the same task on different nodes (i.e., different processors or cores implementing different instruction set architectures). If a queue is configured this way, the order will not be preserved.

Attributes

Attributes are provided as a means to extend the API. Different implementations may define and support additional attributes beyond those predefined by the API. To foster portability and implementation flexibility, attributes are maintained in an opaque data object that may not be examined directly by the user. Each object (e.g., task, action, queue) has an attributes data object associated with it, and many attributes have a small set of predefined values that must be supported by MTAPI implementations. The user may initialize, get, and set these attributes. For default behavior, it is not necessary to call the initialize, get, and set attribute functions. However, to get non-default behavior, the typical four-step process is:

1. Declare an attributes object of the `mtapi_<object>_attributes_t` data type.
2. `mtapi_<object>attr_init()`: Returns an attributes object with all attributes set to their default values.
3. `mtapi_<object>attr_set()` (Repeat for all attributes to be set): Assigns a value to the specified attribute of the specified attributes object.
4. `mtapi_<object>_create()`: Passes the attributes object modified in the previous step as a parameter when creating the object.

At any time, the user can call `mtapi_<object>_get_attribute()` to query the value of an attribute. After an object has been created, some objects allow to change attributes by calling `mtapi_<object>_set_attribute()`.

C Interface

The calculation of Fibonacci numbers is a simple example for a recursive algorithm that can easily be parallelized. Listing 5 shows a sequential version:

```
int fib(int n) {
  int x,y;
  if (n < 2) {
    return n;
  } else {
    x = fib(n - 1);
    y = fib(n - 2);
    return x + y;
  }
}

int fibonacci(int n) {
  return fib(n);
}

void main(void) {
  int n = 6;
  int result = fibonacci(n);
  printf("fib(%i) = %i\n", n, result);
}
```

Listing 5: Sequential program for computing Fibonacci numbers

This algorithm can be parallelized by spawning a task for one of the recursive calls (`fib(n - 1)`, for example). When doing this with MTAPI, an action function that represents `fib(int n)` is needed. It has the following signature:

```
void fibonacciActionFunction(
  const void* args,
  mtapi_size_t arg_size,
  void* result_buffer,
  mtapi_size_t result_buffer_size,
  const void* /*node_local_data*/,
  mtapi_size_t /*node_local_data_size*/,
  mtapi_task_context_t* task_context
  ) {
```

Within the action function, the arguments should be checked, since the user might supply a buffer that is too small:

```
/* check size of arguments (in this case we only expect one int
   value) */
mtapi_status_t status;
if (arg_size != sizeof(int)) {
  printf("wrong size of arguments\n");
  mtapi_context_status_set(task_context, MTAPI_ERR_ARG_SIZE,
                           &status);
  MTAPI_CHECK_STATUS(status);
  return;
}
```

```
/* cast arguments to the desired type */
int n = *(int*)args;
```

Here, `mtapi_context_status_set()` is used to report errors. The error code will be returned by `mtapi_task_wait()`. Also, care has to be taken when using the result buffer. The user might not want to use the result and supply a `NULL` pointer or accidentally a buffer that is too small:

```
/* if the caller is not interested in results, result_buffer may be
   MTAPI_NULL. Of course, this depends on the application */
int* result = MTAPI_NULL;
if (result_buffer == MTAPI_NULL) {
  mtapi_context_status_set(
    task_context, MTAPI_ERR_RESULT_SIZE, &status);
  MTAPI_CHECK_STATUS(status);
} else {
  /* if results are expected by the caller, check result buffer
     size... */
  if (result_buffer_size == sizeof(int)) {
    /* ... and cast the result buffer */
    result = (int*)result_buffer;
  } else {
    printf("wrong size of result buffer\n");
    mtapi_context_status_set(
      task_context, MTAPI_ERR_RESULT_SIZE, &status);
    MTAPI_CHECK_STATUS(status);
    return;
  }
}
```

At this point, calculation of the result can commence. First, the terminating condition of the recursion is checked:

```
if (n < 2) {
  *result = n;
} else {
```

After that, the first part of the computation is launched as a task using `mtapi_task_start()` (the action function is registered with the job `FIBONACCI_JOB` in the `fibonacci()` function and the resulting handle is stored in the global variable `mtapi_job_hndl_t fibonacciJob`):

```
int a = n - 1;
int x;
mtapi_task_hndl_t task = mtapi_task_start(
  MTAPI_TASK_ID_NONE,               /* optional task ID */
  fibonacciJob,                     /* job */
  (void*)&a,                        /* arguments passed to action
                                       functions */
  sizeof(int),                      /* size of arguments */
  (void*)&x,                        /* result buffer */
  sizeof(int),                      /* size of result buffer */
  MTAPI_DEFAULT_TASK_ATTRIBUTES,    /* task attributes */
  MTAPI_GROUP_NONE,                 /* optional task group */
  &status                           /* status out - parameter */
);
MTAPI_CHECK_STATUS(status);
```

The second part can be executed directly:

```
int b = n - 2;
int y;
```

```
fibonacciActionFunction(
  &b, sizeof(int),
  &y, sizeof(int),
  MTAPI_NULL, 0,
  task_context);
```

Then, completion of the MTAPI task has to be waited for by calling `mtapi_task_wait()`:

```
mtapi_task_wait(task, MTAPI_INFINITE, &status);
```

Finally, the results can be added and written into the result buffer:

```
*result = x + y;
```

The `fibonacci()` function gets a bit more complicated now. The MTAPI runtime has to be initialized first by (optionally) initializing node attributes and then calling `mtapi_initialize()`:

```
mtapi_status_t status;

/* initialize node attributes to default values */
mtapi_node_attributes_t node_attr;
mtapi_nodeattr_init(&node_attr, &status);
MTAPI_CHECK_STATUS(status);

/* set node type to SMP */
mtapi_nodeattr_set(
  &node_attr,
  MTAPI_NODE_TYPE,
  MTAPI_ATTRIBUTE_VALUE(MTAPI_NODE_TYPE_SMP),
  MTAPI_ATTRIBUTE_POINTER_AS_VALUE,
  &status);
MTAPI_CHECK_STATUS(status);

/* initialize the node */
mtapi_info_t info;
mtapi_initialize(
  THIS_DOMAIN_ID,
  THIS_NODE_ID,
  &node_attr,
  &info,
  &status);
MTAPI_CHECK_STATUS(status);
```

Then, the action function needs to be associated to a job. By calling `mtapi_action_create()`, the action function is registered with the job `FIBONACCI_JOB`. The job handle of this job is stored in the global variable `mtapi_job_hndl_t fibonacciJob` so that it can be accessed by the action function later on:

```
/* create action */
mtapi_action_hndl_t fibonacciAction;
fibonacciAction = mtapi_action_create(
  FIBONACCI_JOB,                     /* action ID, defined by the
                                        application */
  (fibonacciActionFunction),         /* action function */
  MTAPI_NULL,                        /* no shared data */
  0,                                 /* length of shared data */
  MTAPI_DEFAULT_ACTION_ATTRIBUTES,   /* action attributes */
  &status                            /* status out - parameter */
);
MTAPI_CHECK_STATUS(status);
```

```
/* get job */
mtapi_task_hndl_t task;
fibonacciJob = mtapi_job_get(FIBONACCI_JOB, THIS_DOMAIN_ID, &status);
MTAPI_CHECK_STATUS(status);
```

Now that the action is registered with a job, the root task can be started with `mtapi_task_start()`:

```
/* start task */
int result;
task = mtapi_task_start(
  MTAPI_TASK_ID_NONE,                 /* optional task ID */
  fibonacciJob,                       /* job */
  (void*)&n,                          /* arguments passed to action
                                         functions */
  sizeof(int),                        /* size of arguments */
  (void*)&result,                     /* result buffer */
  sizeof(int),                        /* size of result buffer */
  MTAPI_DEFAULT_TASK_ATTRIBUTES,      /* task attributes */
  MTAPI_GROUP_NONE,                   /* optional task group */
  &status                             /* status out - parameter */
);
MTAPI_CHECK_STATUS(status);
```

After everything is done, the action is deleted (`mtapi_action_delete()`) and the runtime is shut down (`mtapi_finalize()`):

```
/* delete action */
mtapi_action_delete(fibonacciAction, 100, &status);
MTAPI_CHECK_STATUS(status);

/* finalize the node */
mtapi_finalize(&status);
MTAPI_CHECK_STATUS(status);
```

C++ Interface

As mentioned previously, EMB² provides C++ wrappers for the MTAPI C interface. The full interface provides functions for all MTAPI related tasks and even supports heterogeneous systems. For ease of use, a simpler version for SMP systems is also provided.

Full Interface

The signature of an action function for the C++ interface is the same as for the C interface:

```
void fibonacciActionFunction(
  const void* args,
  mtapi_size_t arg_size,
  void* result_buffer,
  mtapi_size_t result_buffer_size,
  const void* /*node_local_data*/,
  mtapi_size_t /*node_local_data_size*/,
  mtapi_task_context_t* task_context
  ) {
```

Checking argument and result buffer sizes is the same as in the C example. Also, the terminating condition of the recursion still needs to be checked:

```
if (n < 2) {
  *result = n;
} else {
```

After that, the first part of the computation is launched as an MTAPI task using `embb::mtapi::Node::Start()` (the action function is registered with the job `FIBONACCI_JOB` in the `fibonacci()` function and the resulting handle is stored in the global variable `embb::mtapi::Job fibonacciJob`):

```
int a = n - 1;
int x;
embb::mtapi::Task task = node.Start(fibonacciJob, &a, &x);
```

The second part can be executed directly:

```
int b = n - 2;
int y;
fibonacciActionFunction(
  &b, sizeof(int),
  &y, sizeof(int),
  MTAPI_NULL, 0,
  task_context);
```

Then, completion of the MTAPI task has to be waited for using `embb::mtapi::Task::Wait()`:

```
mtapi_status_t task_status = task.Wait(MTAPI_INFINITE);
if (task_status != MTAPI_SUCCESS) {
  printf("task failed with error: %d\n\n", task_status);
  exit(task_status);
}
```

Finally, the two parts can be added and written into the result buffer:

```
*result = x + y;
```

Note that there is no need to do error checking everywhere, since errors are reported as exceptions. In this example there is only a single try/catch block in the main function:

```
EMBB_TRY {
  int result = fibonacci(6);
  std::cout << "result: " << result << std::endl;
} EMBB_CATCH(embb::mtapi::StatusException &) {
  std::cout << "MTAPI error occured." << std::endl;
}
```

The `fibonacci()` function is about the same as in the C version. The MTAPI runtime needs to be initialized first:

```
/* initialize the node with default attributes */
embb::mtapi::Node::Initialize(THIS_DOMAIN_ID, THIS_NODE_ID);
```

Then, the node instance can be fetched:

```
embb::mtapi::Node& node = embb::mtapi::Node::GetInstance();
```

After that, the action function needs to be associated to a job. By instantiating an `embb::mtap::Action` object, the action function is registered with the job `FIBONACCI_JOB`. The job is stored in the global variable `embb::mtapi::Job fibonacciJob` so that it can be accessed by the action function later on:

```
/* create action */
embb::mtapi::Action fibonacciAction = node.CreateAction(
  FIBONACCI_JOB,                    /* action ID, defined by the
                                       application */
  (fibonacciActionFunction)         /* action function */
);

/* get job */
fibonacciJob = node.GetJob(FIBONACCI_JOB, THIS_DOMAIN_ID);
```

Now that the action is registered and the job is initialized, the root task can be started:

```
int result;
embb::mtapi::Task task = node.Start(fibonacciJob, &n, &result);
```

Again, the started task has to be waited for (using `embb::mtapi::Task::Wait()`) before the result can be returned.

The registered action will be unregistered when it goes out of scope. The runtime needs to be shut down by calling:

```
embb::mtapi::Node::Finalize();
```

Simplified Interface for SMP actions

The signature of an action function for the simplified API (SMP systems) looks like this:

```
void simpleActionFunction(
  TaskContext & task_context
) {
  // something useful
}
```

The action function does not need to be registered with a job. Instead, a preregistered job is used that expects an `embb::base::Function<void, embb::mtapi::TaskContext &>` object. Therefore, a task can be scheduled directly using only the function above:

```
embb::mtapi::Task task = node.Start(simpleActionFunction);
```

Plugins

The implementation of MTAPI provides an extension to allow for custom actions that are not executed by the scheduler for software actions as described in the previous sections. Three plugins are delivered with EMB^2, one for supporting distributed systems through TCP/IP networking and the other two for OpenCL or CUDA-capable GPUs.

Plugin API

The plugin API essentially consists of a single function contained in the `mtapi_ext.h` header file:

```
mtapi_ext_plugin_action_create()
```

This function is used to associate the plugin action with a specific job ID:

```
mtapi_action_hndl_t mtapi_ext_plugin_action_create(
  MTAPI_IN mtapi_job_id_t job_id,
  MTAPI_IN mtapi_ext_plugin_task_start_function_t task_start_function,
  MTAPI_IN mtapi_ext_plugin_task_cancel_function_t task_cancel_function,
  MTAPI_IN mtapi_ext_plugin_action_finalize_function_t action_finalize_function,
  MTAPI_IN void* plugin_data,
  MTAPI_IN void* node_local_data,
  MTAPI_IN mtapi_size_t node_local_data_size,
  MTAPI_IN mtapi_action_attributes_t* attributes,
  MTAPI_OUT mtapi_status_t* status
);
```

The plugin action is implemented through three callbacks: task start, task cancel, and action finalize.

`task_start_function` is called when the user requests execution of the plugin action by calling `mtapi_task_start()` or `mtapi_task_enqueue()`. To those functions the fact that they operate on a plugin action is transparent, they only require the handle of the job the action was registered with.

`task_cancel_function` is called when the user requests cancelation of a task by calling `mtapi_task_cancel()` or by calling `mtapi_queue_disable()` on a non-retaining queue.

`action_finalize_function` is called when the node is finalized and the action is deleted, or when the user explicitly deletes the action by calling `mtapi_action_delete()`.

For illustration, our example plugin will provide a no-op action. The task start callback in that case looks like this:

```
void plugin_task_start(
  MTAPI_IN mtapi_task_hndl_t task,
  MTAPI_OUT mtapi_status_t* status) {
  mtapi_status_t local_status = MTAPI_ERR_UNKNOWN;

  // do we have a node?
  if (embb_mtapi_node_is_initialized()) {
    // get the node instance
    embb_mtapi_node_t * node = embb_mtapi_node_get_instance();

    // is this a valid task?
    if (embb_mtapi_task_pool_is_handle_valid(node->task_pool, task)) {
      // get the tasks storage
      embb_mtapi_task_t * local_task =
        embb_mtapi_task_pool_get_storage_for_handle(node->task_pool, task);

      // dispatch the task
      plugin_task_schedule(local_task);

      local_status = MTAPI_SUCCESS;
    }
    else {
      local_status = MTAPI_ERR_TASK_INVALID;
    }
  }
  else {
    local_status = MTAPI_ERR_NODE_NOTINIT;
  }

  mtapi_status_set(status, local_status);
}
```

The scheduling operation is responsible for bringing the task to execution. This might involve instructing some hardware to execute the task or pushing the task into a queue for execution by a separate worker thread. Here, however, the task is executed directly:

```
void plugin_task_schedule(embb_mtapi_task_t* local_task) {
  // here the task might be dispatched to some hardware or separate thread

  // mark the task as running
  embb_mtapi_task_set_state(local_task, MTAPI_TASK_RUNNING);

  // nothing to do to execute the no-op task

  // just mark the task as done
  embb_mtapi_task_set_state(local_task, MTAPI_TASK_COMPLETED);
}
```

Since the task gets executed right away, it cannot be canceled and the task cancel callback implementation is empty:

```
void plugin_task_cancel(
  MTAPI_IN mtapi_task_hndl_t task,
  MTAPI_OUT mtapi_status_t* status
  ) {
  EMBB_UNUSED(task);
  // nothing to cancel in this simple example
  mtapi_status_set(status, MTAPI_SUCCESS);
}
```

The plugin action did not acquire any resources, so the action finalize callback is empty as well:

```
void plugin_action_finalize(
  MTAPI_IN mtapi_action_hndl_t action,
  MTAPI_OUT mtapi_status_t* status
  ) {
  EMBB_UNUSED(action);
  // nothing to do for tearing down the plugin action
  mtapi_status_set(status, MTAPI_SUCCESS);
}
```

Now that the callbacks are in place, the action can be registered with a job after the node was initialized using `mtapi_initialize()`:

```
action = mtapi_ext_plugin_action_create(
  PLUGIN_JOB_ID,
  plugin_task_start,
  plugin_task_cancel,
  plugin_action_finalize,
  MTAPI_NULL,
  MTAPI_NULL,
  0,
  MTAPI_DEFAULT_ACTION_ATTRIBUTES,
  &status);
```

The job handle can now be obtained the normal MTAPI way. The fact that there is a plugin working behind the scenes is transparent:

```
job = mtapi_job_get(
  PLUGIN_JOB_ID,
  PLUGIN_DOMAIN_ID,
  &status);
```

Using the job handle, tasks can be started like normal MTAPI tasks:

```
task = mtapi_task_start(
  MTAPI_TASK_ID_NONE,
  job,
  MTAPI_NULL, 0,
  MTAPI_NULL, 0,
  MTAPI_DEFAULT_TASK_ATTRIBUTES,
  MTAPI_GROUP_NONE,
  &status);
```

This call will lead to the invocation of the `plugin_task_start` callback function, where the plugin implementer is responsible for bringing the task to execution.

Network

The MTAPI network plugin provides a means to distribute tasks over a TCP/IP network. As an example, the following vector addition action is used:

```
void AddVectorAction(
  void const * arguments,
  mtapi_size_t arguments_size,
  void * result_buffer,
  mtapi_size_t result_buffer_size,
  void const * node_local_data,
  mtapi_size_t node_local_data_size,
  mtapi_task_context_t * context) {
  EMBB_UNUSED(context);
  EMBB_UNUSED(result_buffer_size);
  EMBB_UNUSED(node_local_data_size);
  int elements = static_cast<int>(arguments_size / sizeof(float) / 2);
  float const * a = reinterpret_cast<float const *>(arguments);
  float const * b = reinterpret_cast<float const *>(arguments)+elements;
  float * c = reinterpret_cast<float*>(result_buffer);
  float const * d = reinterpret_cast<float const *>(node_local_data);
  for (int ii = 0; ii < elements; ii++) {
    c[ii] = a[ii] + b[ii] + d[0];
  }
}
```

It adds two float vectors and a float from node local data, and writes the result into the result float vector. In the example, code the vectors will hold `kElements` floats each.

To use the network plugin, its header file needs to be included first:

```
#include <embb/mtapi/c/mtapi_network.h>
```

After initializing the node using `mtapi_initialize()`, the plugin itself needs to be initialized:

```
mtapi_network_plugin_initialize("127.0.0.1", 12345, 5,
  kElements * 4 * 3 + 32, &status);
```

This will set up a listening socket on the localhost interface (127.0.0.1) at port 12345. The socket will allow a maximum of five connections and has a maximum transfer buffer size of `kElements * 4 * 3 + 32`. This buffer size needs to be large enough to fit at least the argument and result buffer sizes at once. The example uses three vectors of `kElements` floats using `kElements * sizeof(float) * 3` bytes.

Since the example connects to itself on localhost, the "remote" action needs to be registered with the `NETWORK_REMOTE_JOB`:

```
float node_remote = 1.0f;
local_action = mtapi_action_create(
  NETWORK_REMOTE_JOB,
  AddVectorAction,
  &node_remote, sizeof(float),
  MTAPI_DEFAULT_ACTION_ATTRIBUTES,
  &status);
```

After that, the local network action is created that maps `NETWORK_LOCAL_JOB` to `NETWORK_REMOTE_JOB` through the network:

```
network_action = mtapi_network_action_create(
  NETWORK_DOMAIN,
  NETWORK_LOCAL_JOB,
  NETWORK_REMOTE_JOB,
  "127.0.0.1", 12345,
  &status);
```

Now, `NETWORK_LOCAL_JOB` can be used to execute tasks by simply calling `mtapi_task_start()`. Their parameters will be transmitted through a socket connection and are consumed by the network plugin worker thread. The thread will start a task using the `NETWORK_REMOTE_JOB`. When this task is finished, the results will be collected and sent back through the network. Again, the network plugin thread will receive the results, provide them to the `NETWORK_LOCAL_JOB` task and mark that task as finished.

When all work is done, the plugin needs to be finalized. This will stop the plugin worker thread and close the sockets:

```
mtapi_network_plugin_finalize(&status);
```

After that, the node may be finalized by calling `mtapi_finalize()`.

OpenCL

The MTAPI OpenCL plugin allows the user to leverage the computational power of an OpenCL accelerator, if one is available in the system.

Let us reuse the vector addition example from the network plugin. However, the action function is an OpenCL kernel now:

```
const char * kernel =
"__kernel void AddVector(\n"
"  __global void* arguments,\n"
"  int arguments_size,\n"
"  __global void* result_buffer,\n"
"  int result_buffer_size,\n"
"  __global void* node_local_data,\n"
"  int node_local_data_size) {\n"
"  int ii = get_global_id(0);\n"
"  int elements = arguments_size / sizeof(float) / 2;\n"
"  if (ii >= elements)"
"    return;"
"  __global float* a = (__global float*)arguments;\n"
"  __global float* b = ((__global float*)arguments) + elements;\n"
"  __global float* c = (__global float*)result_buffer;\n"
"  __global float* d = (__global float*)node_local_data;\n"
"  c[ii] = a[ii] + b[ii] + d[0];\n"
"}\n";
```

The OpenCL plugin header file needs to be included first:

```
#include <embb/mtapi/c/mtapi_opencl.h>
```

As with the network plugin, the OpenCL plugin needs to be initialized after the node has been initialized:

```
mtapi_opencl_plugin_initialize(&status);
```

Then, the plugin action can be registered with the `OPENCL_JOB`:

```
float node_local = 1.0f;
action = mtapi_opencl_action_create(
  OPENCL_JOB,
  kernel, "AddVector", 32, 4,
  &node_local, sizeof(float),
  &status);
```

The kernel source and the name of the kernel to use (`AddVector`) need to be specified while creating the action. The kernel will be compiled using the OpenCL runtime and the provided node local data will be transferred to the accelerator memory. The local work size is the number of threads that will share OpenCL local memory, in this case 32. The element size tells the OpenCL plugin how many bytes a single element in the result buffer consumes, in this case 4, as a single result is a single float. The OpenCL plugin will launch `result_buffer_size/element_size` OpenCL threads to calculate the result.

Now, the `OPENCL_JOB` can be used like a normal MTAPI job to start tasks.

After all work is done, the plugin needs to be finalized. This will free all memory on the accelerator and delete the corresponding OpenCL context:

```
mtapi_opencl_plugin_finalize(&status);
```

CUDA

Similar to the OpenCL plugin, the CUDA plugin can be used to start tasks on an Nvidia GPU.

The vector addition example looks slightly different in CUDA:

```
extern "C" __global__ void AddVector(
  void* arguments,
  int arguments_size,
  void* result_buffer,
  int result_buffer_size,
  void* node_local_data,
  int node_local_data_size) {
  int ii = blockDim.x * blockIdx.x + threadIdx.x;
  int elements = arguments_size / sizeof(float) / 2;
  if (ii >= elements)
    return;
  float* a = (float*)arguments;
  float* b = ((float*)arguments) + elements;
  float* c = (float*)result_buffer;
  float* d = (float*)node_local_data;
  c[ii] = a[ii] + b[ii] + d[0];
}
```

The kernel needs to be precompiled and will be transformed into a header file containing the resulting binary in a `char const *` array named `imageBytes`.

As with the OpenCL plugin, the CUDA plugin header file needs to be included first:

```
#include <embb/mtapi/c/mtapi_cuda.h>
```

Then, the CUDA plugin needs to be initialized after the node has been initialized:

```
mtapi_cuda_plugin_initialize(&status);
```

Now, the plugin action can be registered with the `CUDA_JOB`:

```
float node_local = 1.0f;
action = mtapi_cuda_action_create(
  CUDA_JOB,
  reinterpret_cast<char const *>(imageBytes), "AddVector", 32, 4,
  &node_local, sizeof(float),
  &status);
```

The precompiled kernel binary and the name of the kernel to use need to be specified while creating the action. The kernel and node local data provided are transferred to the accelerator memory. The local work size is the number of threads that will share CUDA local memory, in this case 32. The element size tells the CUDA plugin how many bytes a single element in the result buffer consumes, in this case 4, as a single result is a single float. The CUDA plugin will launch `result_buffer_size/element_size` CUDA threads to calculate the result.

Now, the `CUDA_JOB` can be used like a normal MTAPI job to start tasks.

After all work is done, the plugin needs to be finalized. This will free all memory on the accelerator and delete the corresponding CUDA context:

```
mtapi_cuda_plugin_finalize(&status);
```

Heterogeneous Systems

Algorithms

All of the algorithms provided by EMB^2 can also be used on heterogeneous systems. This allows to transparently offload work on different kinds of compute units, thus leveraging the available hardware resources in an optimal way. In the following, we focus on the key concepts—for more detailed information, please see the source code contained in the `examples` directory.

In addition to functions, functors or lambdas, the algorithms accept MTAPI job handles that implement the intended functionality. The action functions will be given structures containing the arguments and results according to the signatures used above. For the sake of simplicity, CPU actions are used to simulate a heterogeneous system. The CPU actions are functions with the following signature:

```
void Action(
  const void* args,
  mtapi_size_t args_size,
  void* result_buffer,
  mtapi_size_t result_buffer_size,
  const void* node_local_data,
  mtapi_size_t node_local_data_size,
  mtapi_task_context_t* task_context
);
```

A node handle is retrieved and used by the following examples like this:

```
embb::mtapi::Node & node = embb::mtapi::Node::GetInstance();
```

Invoke

First, we consider `Invoke` to start two jobs in parallel. For that purpose, we define two action functions `InvokeA` and `InvokeB` that have no parameters and just increment a global value (`a` and `b`, respectively). For `InvokeA`, we have:

```
static int a = 0;

static void InvokeA(
  const void* /*args*/,
  mtapi_size_t /*args_size*/,
  void* /*result_buffer*/,
  mtapi_size_t /*result_buffer_size*/,
  const void* /*node_local_data*/,
  mtapi_size_t /*node_local_data_size*/,
  mtapi_task_context_t* /*task_context*/
) {
  a++;
}
```

The actions are associated with the job IDs `JOB_A` and `JOB_B`. The job handles are retrieved as follows:

```
static const mtapi_job_id_t JOB_INVOKE_A = 10;
embb::mtapi::Job job_a = node.GetJob(JOB_INVOKE_A);
```

```
static const mtapi_job_id_t JOB_INVOKE_B = 11;
embb::mtapi::Job job_b = node.GetJob(JOB_INVOKE_B);
```

After that, the jobs can be started:

```
embb::algorithms::Invoke(job_a, job_b);
```

The global variables `a` and `b` are now both set to `1`.

Sorting

To use `QuickSort`, we need a comparison function. `DescendingCompare` has two arguments of type `int` and one result of type `bool`. Since the function signature is fixed, we pack the arguments into a struct:

```
typedef struct {
  int lhs;
  int rhs;
} InT;
```

The same holds for the result:

```
typedef struct {
  bool out;
} OutT;
```

`args` needs to be casted to `InT` and `result_buffer` to `OutT`:

```
InT const * inputs = static_cast<InT const *>(args);
OutT * outputs = static_cast<OutT *>(result_buffer);
```

Now, the arguments can be accessed, compared and the result can be written:

```
outputs->out = inputs->lhs > inputs->rhs;
```

`DescendingCompare` is associated with the job ID `JOB_COMPARE` and the job handle can be retrieved as follows:

```
static const mtapi_job_id_t JOB_COMPARE = 10;
embb::mtapi::Job job_compare = node.GetJob(JOB_COMPARE);
```

Then, we prepare a vector with `int`'s to be sorted:

```
static const size_t kCountSize = 10;
std::vector<int> vector(kCountSize);
for (size_t i = 0; i < kCountSize; i++) {
  vector[i] = static_cast<int>(i + 2);
}
```

Finally, we call `QuickSort`

```
embb::algorithms::QuickSort(vector.begin(), vector.end(), job_compare);
```

and the `int`'s in the vector are now in descending order.

Counting

The predicate supplied to `CountIf` can be implemented by an action function `CheckZero` that takes one argument of type `int` and returns one result of type `bool`. The argument is again packed into a struct:

```
typedef struct {
  int val;
} InT;
```

The result struct is the same as in the sorting example. Also, `args` and `result_buffer` need to be casted to `InT` and `OutT`. Then, the body of `CheckZero` is simply:

```
outputs->out = inputs->val == 0;
```

After retrieving the job handle

```
static const mtapi_job_id_t JOB_CHECK_ZERO = 10;
embb::mtapi::Job job_check_zero = node.GetJob(JOB_CHECK_ZERO);
```

we prepare a vector with `int`'s to count (if they are zero):

```
static const size_t kCountSize = 10;
std::vector<int> vector(kCountSize);
for (size_t i = 0; i < kCountSize; i++) {
  vector[i] = int(i) % 2;
}
```

Finally, we call `CountIf`

```
std::vector<int>::iterator::difference_type count =
  embb::algorithms::CountIf(vector.begin(), vector.end(), job_check_zero);
```

which returns the number of zeros in the vector.

Foreach Loops

`ForEach` accepts functions taking a reference to an iterator as argument in order to work on the referenced object. Consider, for example, an action function `Double` that doubles a given value. It has one argument of type `int` and one result of type `int`. The argument struct is thus the same as in the counting example. The result resides in a struct containing one `int`:

```
typedef struct {
  int out;
} OutT;
```

`args` and `result_buffer` need to be casted to `InT` and `OutT` once more. The body of `Double` is defined as follows:

```
outputs->out = inputs->val * 2;
```

After retrieving the job handle

```
static const mtapi_job_id_t JOB_DOUBLE = 10;
embb::mtapi::Job job_double = node.GetJob(JOB_DOUBLE);
```

we can call `ForEach` with this handle:

```
embb::algorithms::ForEach(vector.begin(), vector.end(), job_double);
```

Reductions and Prefix Computations

`Reduce` and `Scan` use a reduction and a transformation function. The reduction function has two arguments and one result that all have the same type. The transformation function has one argument and one result with potentially different types. In our example, they are all of type `int`. For simplicity, we reuse the `Double` action and `JOB_DOUBLE` from the previous example as our transformation function. For the reduction function, we introduce an action `Add` with the following simple body:

```
outputs->out = inputs->lhs + inputs->rhs;
```

The job handles are retrieved as follows:

```
static const mtapi_job_id_t JOB_DOUBLE = 10;
embb::mtapi::Job job_double = node.GetJob(JOB_DOUBLE);

static const mtapi_job_id_t JOB_ADD = 11;
embb::mtapi::Job job_add = node.GetJob(JOB_ADD);
```

Next, we create a vector of `int`'s:

```
static const size_t kCountSize = 10;
std::vector<int> vector(kCountSize);
for (size_t i = 0; i < kCountSize; i++) {
  vector[i] = int(i);
}
```

The elements of the vector can then be reduced by:

```
int result =
  embb::algorithms::Reduce(vector.begin(), vector.end(), 0,
    job_add, job_double);
```

The prefix sum is computed similarly:

```
std::vector<int> output(kCountSize);
embb::algorithms::Scan(vector.begin(), vector.end(), output.begin(), 0,
  job_add, job_double);
```

Dataflow

Dataflow networks can be used on heterogeneous systems as well. In addition to functions and functors, dataflow sources, sinks, and processes accept MTAPI job handles implementing the intended functionality. The action functions will be given structures containing the arguments and results according to the signatures used previously. To simulate a heterogeneous system, CPU actions with the following signature are used:

```
void Action(
  const void* args,
  mtapi_size_t args_size,
  void* result_buffer,
  mtapi_size_t result_buffer_size,
  const void* node_local_data,
  mtapi_size_t node_local_data_size,
  mtapi_task_context_t* task_context
);
```

Suppose, for example, we want to double the integers from 0 to 9 and sum them up. For that purpose, we define three action functions, one for a source called `Generate`, one for a process called `Double`, and one for a sink called `Accumulate`. Each of them is associated with a different job.

Generate

The source function generates integers from 0 to 9. It receives no arguments and returns a `bool` that indicates whether generating integers shall continue, and an `int` that represents the generated value. The results are packed into a struct:

```
typedef struct {
  bool more;
  int out;
} OutT;
```

The `result_buffer` pointer needs to be casted to `OutT`:

```
OutT * outputs = static_cast<OutT *>(result_buffer);
```

Then, the body of the function is:

```
static int value = 0;
outputs->out = value;
```

```
outputs->more = value < 10;
value++;
```

Double

The process function expects and `int` and returns the double of its value. Both, the argument and the result are packed into a struct:

```
typedef struct {
  int val;
} InT;
```

```
typedef struct {
  int out;
} OutT;
```

As usual, `args` as well as `result_buffer` need to be casted:

```
InT const * inputs = static_cast<InT const *>(args);
```

```
OutT * outputs = static_cast<OutT *>(result_buffer);
```

Finally, the actual calculation is simply:

```
outputs->out = inputs->val * 2;
```

Accumulate

The sink is supposed to add up all incoming values. It returns no result and expects a value of type `int` which is packed into a struct:

```
typedef struct {
  int val;
} InT;
```

After casting `args` to `InT`, the inputs can be accumulated:

```
InT const * inputs = static_cast<InT const *>(args);
```

```
accumulated_result += inputs->val;
```

Network

In the main function, we first retrieve the node handle:

```
embb::mtapi::Node & node = embb::mtapi::Node::GetInstance();
```

Then, we obtain the job handles:

```
static const mtapi_job_id_t JOB_GENERATE = 10;
embb::mtapi::Job job_generate = node.GetJob(JOB_GENERATE);

static const mtapi_job_id_t JOB_DOUBLE = 11;
embb::mtapi::Job job_double = node.GetJob(JOB_DOUBLE);

static const mtapi_job_id_t JOB_ACCUMULATE = 12;
embb::mtapi::Job job_accumulate = node.GetJob(JOB_ACCUMULATE);
```

After that, we define the network and its processes:

```
typedef embb::dataflow::Network Net;
Net net(4);

Net::Source<int> source(net, job_generate);
```

```
Net::ParallelProcess<Net::Inputs<int>, Net::Outputs<int> >
  filter(net, job_double);

Net::Sink<int> sink(net, job_accumulate);
```

Finally, we connect the processes and run the network:

```
source >> filter >> sink;
```

```
net();
```

Tutorial Application

In the following, we create a video processing application that applies the concepts described in the previous sections to a more complex problem. The application is supposed to read a video file, apply some filters, and write the resulting output video to a file. For handling video files, we are going to use FFmpeg. Details on how to build and run the application can be found in the `README.md` file in `doc/tutorial/application`.

The application consists of five parts, the main program, the filters, the input video handler, the frame format converter, and the output video builder. Three of them relate to FFmpeg video decoding and encoding. The input video handler opens a given video file and is used to read consecutive frames from the stream until there are no more frames. The frame format converter is used to convert from the source color format to RGB and vice versa, since the filtering is done in RGB color space. The output video builder encodes the resulting image stream and writes it back to a video file. For the sake of brevity, we will not cover these three parts in detail but focus on parallelizing the filters as well as the whole pipeline. The filters come in three flavors: sequential, parallel using the algorithms library, and as OpenCL kernels. The main application connects the pipeline stages into a working whole.

Filters

The filters are essentially loops iterating over the pixels and applying some operation to them. Here is a simple color to greyscale filter:

```
void applyBlackAndWhite(AVFrame* frame) {
  av_frame_make_writable(frame);

  int const width = frame->width;
  int const height = frame->height;

  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      int p = mapToData(x, y, width);
      int r = frame->data[0][p];
      int g = frame->data[0][p + 1];
      int b = frame->data[0][p + 2];
      int mean = (r + g + b) / 3;
      frame->data[0][p] = mean;
      frame->data[0][p + 1] = mean;
      frame->data[0][p + 2] = mean;
    }
  }
}
```

The other filters contained in `doc\tutorial\application\src\filters.cc` work similarly—feel free to experiment with them. By default, the application “cartoonifies” the incoming video stream (see function `filter` in `main.cc`).

Parallelizing the above filter is straightforward using the algorithms building block: We replace the loops by a single call to `ForLoop` and compute the `x` and `y` pixel coordinates from the given index:

```
void applyBlackAndWhiteParallel(AVFrame* frame) {
  av_frame_make_writable(frame);

  int const width = frame->width;
  int const height = frame->height;

  embb::algorithms::ForLoop(0, width*height, [&](int idx) {
    int x = idx % width;
    int y = idx / width;
    int p = mapToData(x, y, width);
    int r = frame->data[0][p];
    int g = frame->data[0][p + 1];
    int b = frame->data[0][p + 2];
    int mean = (r + g + b) / 3;
    frame->data[0][p] = mean;
    frame->data[0][p + 1] = mean;
    frame->data[0][p + 2] = mean;
  });
}
```

Now, the filter runs in parallel but the control flow of the whole application is still sequential which limits scalability.

Control Flow

Applying video decoding, format conversion, filtering, another format conversion, and video encoding to a sequence of frames can be implemented using the following loop:

```
while (readFromFile(frame)) {
  convertToRGB(frame, convertedFrame);
  filter(convertedFrame);
  convertToOriginal(convertedFrame, originalFrame);
  writeToFile(originalFrame);
}
```

The steps executed in the loop body can be seen as a pipeline which allows us to parallelize them using dataflow networks. Each of the operations is wrapped into a source, a (parallel) process, or a sink. The resulting objects are then connected to a network:

```
Network nw(8);

Network::Source<AVFrame*> read(nw, embb::base::MakeFunction(readFromFile));

Network::ParallelProcess<Network::Inputs<AVFrame*>,
  Network::Outputs<AVFrame*> >
    rgb(nw, embb::base::MakeFunction(convertToRGB));

Network::ParallelProcess<Network::Inputs<AVFrame*>,
  Network::Outputs<AVFrame*> >
    original(nw, embb::base::MakeFunction(convertToOriginal));

Network::ParallelProcess<Network::Inputs<AVFrame*>,
  Network::Outputs<AVFrame*> >
    filter(nw, embb::base::MakeFunction(applyFilter));
```

```
Network::Sink<AVFrame*> write(nw, embb::base::MakeFunction(writeToFile));

read >> rgb >> filter >> original >> write;

nw();
```

This way, scalability is significantly improved, as all parts now run in parallel.

Heterogeneous Systems

As discussed in the chapter on heterogeneous systems, some systems feature additional accelerators, e.g. a GPU, to further improve processing speed. Using OpenCL (or CUDA), we can leverage the power of such accelerators to speed up the filters. Let us consider the following OpenCL implementation:

```
char const * mean_kernel =
"__kernel void mean(\n"
"  __global void* arguments,\n"
"  int arguments_size,\n"
"  __global void* result_buffer,\n"
"  int result_buffer_size,\n"
"  __global void* node_local_data,\n"
"  int node_local_data_size) {\n"
"  int idx = get_global_id(0);\n"
"  int elements = (arguments_size - sizeof(int) * 3) / 3;\n"
"  if (idx >= elements)\n"
"    return;\n"
"  __global int * param = (__global int*)arguments;\n"
"  __global unsigned char * in_buffer ="
"    ((__global unsigned char*)arguments) + sizeof(int) * 3;\n"
"  __global unsigned char * out_buffer ="
"    (__global unsigned char*)result_buffer;\n"
"  int width = param[0];\n"
"  int height = param[1];\n"
"  int size = param[2];\n"
"  int size_lt = (size % 2 != 0) ? size / 2 : size / 2 - 1;\n"
"  int const size_rb = size / 2;\n"
"  int x = idx % width;\n"
"  int y = idx / width; \n"
"  int close = 0;\n"
"  int p = (x + y * width) * 3;\n"
"  int r = 0;\n"
"  int g = 0;\n"
"  int b = 0;\n"
"  for (int k1 = y - size_lt; k1 <= y + size_rb; k1++) {\n"
"    for (int k2 = x - size_lt; k2 <= x + size_rb; k2++) {\n"
"      if (k1 >= 0 && k1 < height && k2 >= 0 && k2 < width) {\n"
"        close++;\n"
"        int p2 = (k2 + k1 * width) * 3;\n"
"        r += in_buffer[p2];\n"
"        g += in_buffer[p2 + 1];\n"
"        b += in_buffer[p2 + 2];\n"
"      }\n"
"    }\n"
"  }\n"
"  out_buffer[p] = r / close;\n"
"  out_buffer[p + 1] = g / close;\n"
"  out_buffer[p + 2] = b / close;\n"
"}\n";
```

First, this kernel needs to be wrapped into an MTAPI action:

```
mtapi_opencl_action_create(JOB_MEAN, filters::mean_kernel,
  "mean", 32, 3, &node_local, sizeof(int), &status);
if (status != MTAPI_SUCCESS) {
  std::cout << "Could not create OpenCL action..." << std::endl;
  mtapi_opencl_plugin_finalize(MTAPI_NULL);
  embb::mtapi::Node::Finalize();
  return;
}
```

Then, the filter can be used like any job. Note, however, the additional cost for copying the frame and the parameters of the filter into a single buffer:

```
args = new unsigned char[n_bytes + sizeof(int) * 4];
res = new unsigned char[n_bytes + sizeof(int) * 4];

size = n_bytes + sizeof(int) * 3;
param = reinterpret_cast<int*>(args);
data = args + sizeof(int) * 3;
param[0] = width;
param[1] = height;
param[2] = 3;
memcpy(data, frame->data[0], n_bytes);
job = node.GetJob(JOB_MEAN);
task = node.Start(MTAPI_TASK_ID_NONE, job.GetInternal(), args, size,
  res + sizeof(int)*4, n_bytes, MTAPI_DEFAULT_TASK_ATTRIBUTES);
task.Wait();
```

When wrapped into a dataflow process, it can even be used as a part of the pipeline outlined before:

```
Network::ParallelProcess<Network::Inputs<AVFrame*>,
  Network::Outputs<AVFrame*> >
  filter(nw, embb::base::MakeFunction(applyFilterOpenCL));
```

Bibliography

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