| | |
|---|---|
| **Document Title** | Design guidelines for using parallel processing technologies on Adaptive Platform |
| **Document Owner** | AUTOSAR |
| **Document Responsibility** | AUTOSAR |
| **Document Identification No** | 884 |

| | |
|---|---|
| **Document Status** | Final |
| **Part of AUTOSAR Standard** | Adaptive Platform |
| **Part of Standard Release** | 17-10 |

# Document Change History

| Date | Release | Changed by | Change Description |
|---|---|---|---|
| 2017-10-27 | 17-10 | AUTOSAR Release Management | • Initial release |

# Table of Contents

# 1 Introduction to this document

## 1.1 Contents

This document specifies the guidelines for using parallel processing technologies on Adaptive Platform, or Parallel Processing Guidelines, in short.

The purpose of this document is to provide design guidelines for using parallel processing technologies on AP. The focus is on software, especially the application layer including the services. General hardware discussions are also included to build the base for software.

The document is organized as follows…**TBA**.

## 1.2 Prereads

This document is one of the high-level conceptual documents of AUTOSAR.
Useful pre-reads are [1] [2] [3] [4].

## 1.3 Relationship to other AUTOSAR specifications

**Refer to** Contents **and** Prereads**.**

# 2 Scope

## 2.1 Definition of parallel processing "technologies"

In this document, the meaning of parallel processing technologies is loosely defined. This is so on purpose, with hopes to provide design principles for parallel and related processing (see Distributed, concurrent, and parallel).

The term "parallel processing technologies" in this document, therefore, covers both hardware and software. In term of hardware, multicore, manycore, DFP (Data-Flow Processor), GPU (Graphical Processing Unit), FPGA (Field-ProGrammable-Array), or alike; in terms of software, multi-thread programming, pragma based techniques like OpenMP[1], various template programming such as TBB[2], accelerator programming language like OpenCL[3], and even various message passing APIs that are not by themselves parallel processing technologies but are tightly related to. The technologies also include various tooling that assist in designing and implementing the parallel processing technologies into an AP based system.

It is not a purpose of this document to list all the existing parallel processing technologies, to explain what they are, nor to guide how to use the technologies themselves. Nevertheless, the document may contain some references to the technologies as minimum as deemed necessary to describe the design guidelines.

## 2.2 Audience

This specification is for multiple domains of AP related designers and developers, namely the system designer who decides hardware/software partitioning, hardware designer who design and/or select computing hardware resources, software designer who design overall software system architecture, AP developers, and developers of AP services running on ARA.

AA developers, on the other hand, who may not directly use parallel processing technologies and only design sequential, single-threaded application, may find this irrelevant, if his/her software architect follows the architectural design guideline described in this document. However, nowadays it is becoming difficult to write an application without some form of multi-thread programming, and it is likely to be more so in future, so essentially everyone concerned with AP are advised to reference this document.

[1] http://www.openmp.org/
[2] https://www.threadingbuildingblocks.org/
[3] https://jp.khronos.org/opencl

# 3 Architectural design

## 3.1 Background

### 3.1.1 Evolving parallel processing technologies

The parallel processing technologies are still rapidly evolving, both in hardware and software. In hardware, GPGPU (General Purpose GPU) is one of them but never the only one - various manycore processors, dataflow processors, FPGA, and some dedicated accelerators are emerging, and it seems there are more to come, including the evolutions of these existing technologies.

The picture looks similar in software. Starting with the threading library offered by POSIX and C++ Standard libraries AP supports, and other threading libraries such as TBB, MTAPI[4], compiler directives based threading like OpenMP, accelerator programming language like OpenCL and CUDA® (proprietary), HLS[5] compiler based FPGA programming, and various parallelization compilers/tools, such as graph or process network based tools, which generally uses threading underneath but technologically not limited to, and there is even a model based parallelization tools that can take Simulink® model as its input. Also, there are various message passing APIs that often works along with these technologies. There are higher-level libraries such as OpenCV[6], OpenVX[7] - though these are not by themselves parallel processing libraries, they generally use parallel processing technologies underneath to accelerate the processing. At last, similar in a sense that they are higher-level, there are C++ AMP[8], and SYCL[9]. To further complicate the matter, OpenMP 4 now supports accelerator. And this is not a complete list.

### 3.1.2 Distributed, concurrent, and parallel

In most cases, AUTOSAR systems are distributed system. A distributed system is concurrent, meaning multiple tasks running at the same time. Each subsystem in the distributed system has some sort of processing elements, typically CPUs (but not necessarily) – therefore at the whole this is a multi-processor system, capable of both concurrent and parallel computing. Note that if AP runs on a single-core processor machine without any other computing elements, parallel processing is not possible, although concurrency still is, as OS provides the threading mechanism to switch processing (thread) triggered by some event.

Parallel processing may occur at different computing layers, from bit-level, instruction-level, thread-level, (and/or) task-level. The definition of “task-level” differs among computing models and methodologies. In AUTOSAR AP software point of view, however, they are strictly either thread-level, process level, or machine (platform) level. It is also noteworthy to recognize a process, in the context of AP that is based on POSIX multi-process OS, is just a container of threads and not an execution entity like threads by itself. The container provides an enclosure for a

[4] The Multicore Association http://www.multicore-association.org/workgroup/mtapi.php
[5] High Level Synthesis
[6] http://opencv.org/
[7] https://www.khronos.org/openvx/
[8] http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf
[9] https://www.khronos.org/sycl

certain unique set of resources, which include some logical memory accessible by the process. This is also the same for the machines. It is always the thread-concurrency and parallelism (if more than two processors are available) at the software level that is directly executed on top of the AUTOSAR AP OS.

There may be other processing elements that are either incapable of directly executing AP but offer some useful computing. GPU, FPGA, DFP (Data Flow Processor), and manycore processor, are representative examples today in 2017, although some of them can execute some executive or OS, and even AP itself, fully or partially. If they are incapable of running AP at least partially, then the parallel processing capability can only be accessed by some kind of specific interfaces from a thread running on AP, regardless of the mechanism behind the interface. They are still programmable in one way or another – but just not in the way ARA and C++ bindings AP defines.

The important architectural design consideration here is that parallel processing at large is a system level topic. Distributed, concurrent, and parallel processing are highly interrelated. One example is that, a well-designed multi-threaded program may run concurrently on a single-core processor, or in parallel on a multi-core processor, or even distributed over two machines provided it uses some processor/machine transparent thread communication.

### 3.1.3 TLP/DLP/PLP

In general, there are three types of parallel processing: Task Level Parallelism (TLP), Data Level Parallelism (DLP), and Pipeline Level Parallelism (PLP). The TLP refers performing multiple tasks at the same time as they are (mostly) independent and do not (mostly) dependent on each other. DLP refers to performing the same calculation with multiple, (mostly) non-interdependent sets of (large) data. The same calculation is multiplexed with the different set of data. PLP refers to executing multiple inter-dependent tasks in a pipeline fashion. Each task is assigned to a pipeline stage per the data dependency of the task input/output.

The three types of parallelism exist in multiple layers of system. There can be a system level parallelism, like two AP machines may have TLP or even PLP. Another example may be that, multiple camera based 360-degree real-time object recognition may be realized by multiple AP machines performing DLP against a large data set of (virtual) vehicle surrounding video image. The point here is that it is critical for a vehicle system designer to understand the system overall data flow and processing loads and allocate AP machines accordingly. This can be said as a **AP-machine level parallelism**.

The next physical level below is **OS-thread level parallelism**. The three types of parallelism described above can be implemented using OS threads.

Yet another physical level below is the **instruction level parallelism**. This is generally in the field of processor and compiler technologies. For example, a VLIW[10] processor architecture has multiple execution units that allows concurrent execution of multiple instruction streams, in either TLP, or DLP fashion. A SIMD (Single

[10] Very Long Instruction Word

Instruction Multiple Data) co-processor instruction extension enables DLP at instruction level. A GPGPU, in general, is a form of instruction level DLP in" 3.2.2 Accelerator-model". The SIMD extension, on the other hand, is DLP in "3.2.3 CPU/co-processor-model". A manycore processor, including most of DFP (Data Flow Processor), offer in general MIMD (Multiple Instruction Multiple Data) instruction level parallelism. Since it is not the same single instruction like SIMD, MIMD can be used to implement all three forms of parallelism, namely TLP, DLP, and PLP.

Also, regardless of the physical levels, namely AP-machine level, OS-thread level, or Instruction level, the TLP, DLP, and PLP are not always used independently. For example, for the multi-stage processing of large data, the combination of DLP and PLP are popular.

## 3.2 Service-based parallel processing

With the background provided in 3.1, the key concept of this guideline is to utilize the SOA of AP. That is, to push the use of parallel processing technologies underneath non-platform services, leaving the AA free from the various parallel processing technologies used. The service 'implementation', on the other hand, will be specific to the choice of parallel processing technologies used. We call this model of parallel processing as "3.2 Service-based parallel processing".

This model allows the maximum reuse of AA that require high-performance computing in realizing its functionality. The heavy lifting part will be separated into non-platform services, and the implementation of services are free to utilize the full capability of the parallel processing technologies of choice, provided they conform to the safety/security requirements of the project.

### 3.2.1 Layered architectural view

The

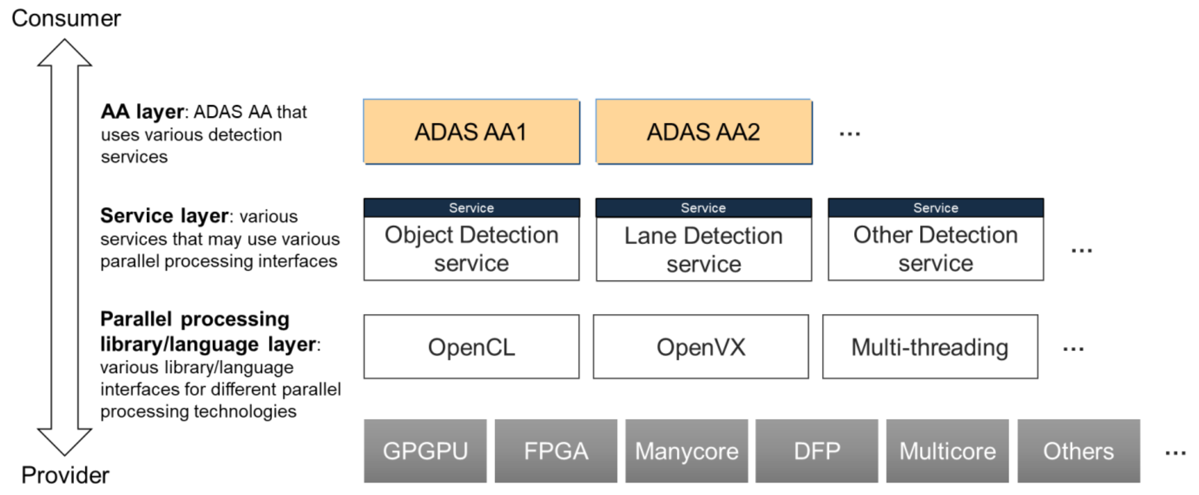
Consumer
AA layer: ADAS AA that uses various detection services
ADAS AA1
ADAS AA2
...
Service layer: various services that may use various parallel processing interfaces
Service
Object Detection service
Service
Lane Detection service
Service
Other Detection service
...
Parallel processing library/language layer: various library/language interfaces for different parallel processing technologies
OpenCL
OpenVX
Multi-threading
...
GPGPU
FPGA
Manycore
DFP
Multicore
Others
...
Provider

Figure 3-1 illustrate the overall architecture of the service-based parallel processing. The example is based on some ADAS domain application, but it is not the intention to limit the domain in any way.

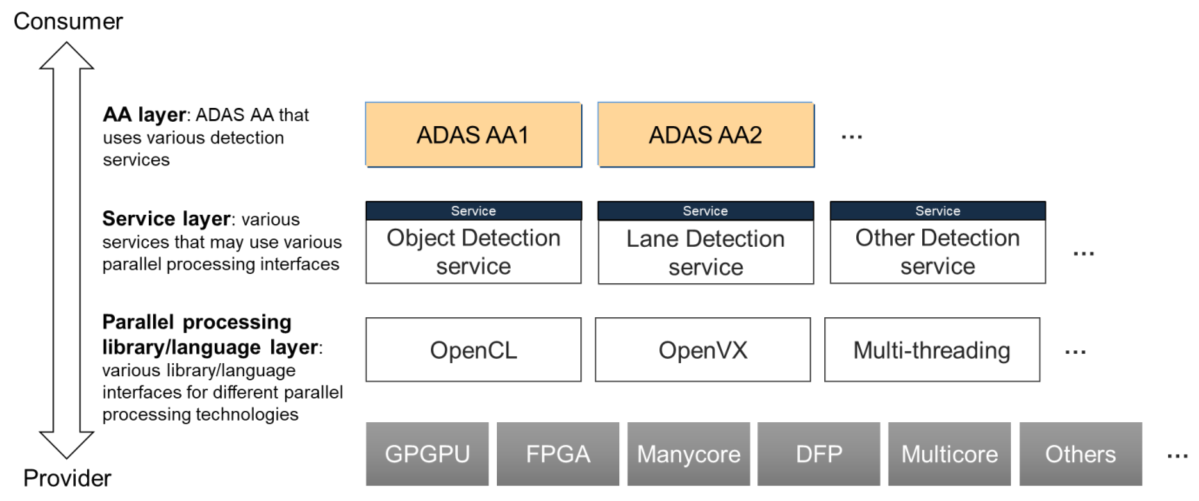
Consumer
AA layer: ADAS AA that uses various detection services
ADAS AA1
ADAS AA2
...
Service layer: various services that may use various parallel processing interfaces
Service
Object Detection service
Service
Lane Detection service
Service
Other Detection service
...
Parallel processing library/language layer: various library/language interfaces for different parallel processing technologies
OpenCL
OpenVX
Multi-threading
...
GPGPU
FPGA
Manycore
DFP
Multicore
Others
...
Provider

**Figure 3-1 Parallel processing consumer-provider layered view example**

The overall picture shows who uses what (consumers), and who provides what (providers), in a top-down layered fashion.

The **AA layer** is the AA that uses various services. The AA does (or should) not know the services it is using uses parallel processing underneath.

The **Service layer**, in the context of this guideline, consists of the services which uses parallel processing technologies. There are non-platform services that uses ara::com. They provide C++ interface library generated from the service definition, which are used by the AA. Note that these services may very well use other services internally. One example is that one may design a pre-processing or low-level sensing service, and a meta-data provider service that uses the output of the former service. Another example may be that one may design various detection services, and a predictor service that use the detection services to predict the object in a future horizon. Also, note that there may well be some common higher level library or engine used by the Service layer. Such a library may use some parallel processing library underneath. One example may be the relationship between OpenCV and OpenCL. OpenCV provides the vision processing framework/library, which underneath (can) use OpenCL to use programmable accelerators. The OpenCV library may be used by multiple services. This is similar for the relationship between OpenVX and OpenCL – however, unlike OpenCV, OpenVX is designed so that the OpenVX interface implementations can directly access the specific accelerators, without OpenCL in between. Therefore, it is drawn to be in the Parallel processing library/language layer in the figure.

The Service layer uses **Parallel processing library/language layer**, which can vary dependent on the choice of parallel processing technologies used in the service implementation. The programming interface for this layer varies as discussed in “3.1.1 Evolving parallel processing technologies”, and it is just not semantically possible to have a single unified interface to generalize all or even most of the different interface/languages, without severely impacting the performance benefit, which contradicts the purpose of employing the parallel processing in the first place.

### 3.2.2 Accelerator-model

The Parallel processing library/languages layer interacts with the parallel processing hardware in different ways. There are two general models. One is **accelerator-model,** where the parallel processing library/language calls underneath some form of device drivers that directly controls the parallel processing hardware. The device driver, depending upon the design of OS used, may be another process or some form of kernel module that executes in the context of OS kernel. The examples of this accelerator model include OpenCL/CUDA, OpenVX, etc. Figure 3-2 shows the combined process and physical architectural views for OpenCL based parallel processing.

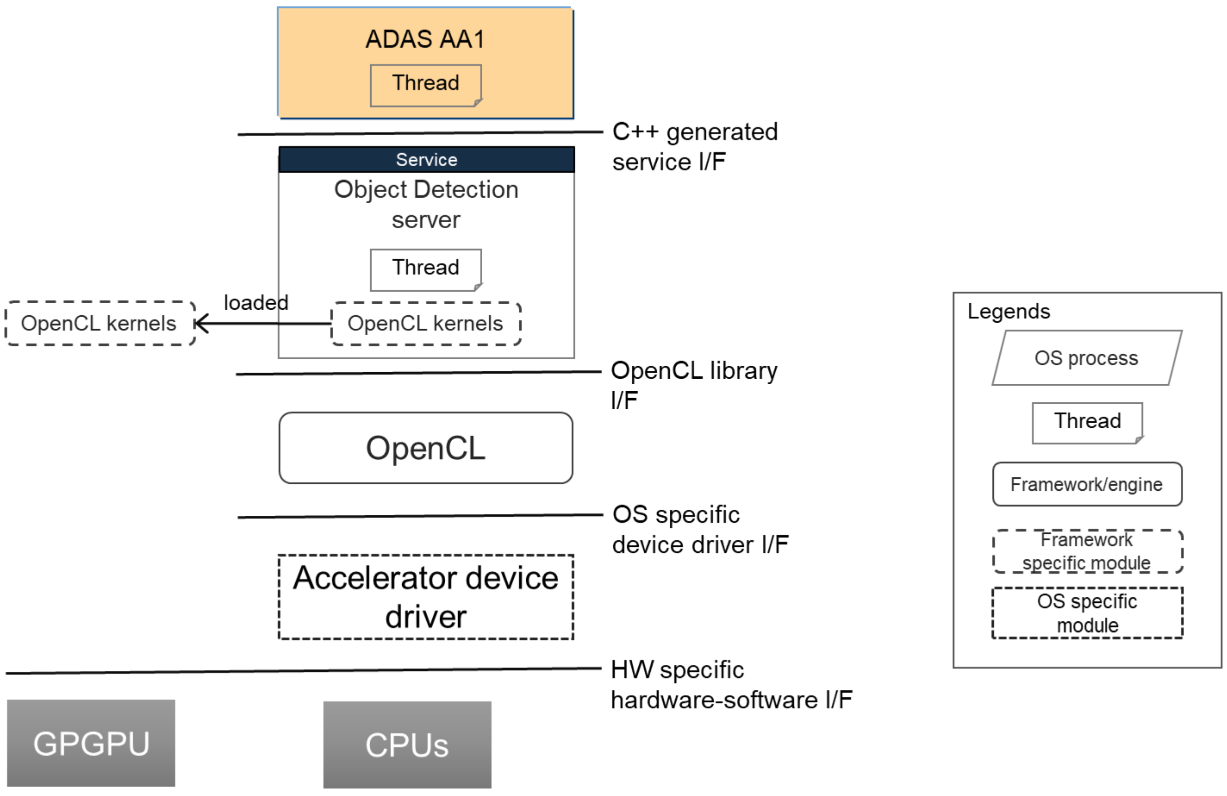

**Figure 3-2 Accelerator-model example**

### 3.2.3 CPU/co-processor-model

The other model is **CPU/co-processor-model**, where the parallel processing is executed directly by the CPUs with or without co-processor support. The most popular example is threading model, which uses multiple POSIX threads to parallelize the processing. This can be fully hand written, directive-based like OpenMP, or use some other vendor specific semi/full parallelization compiler technologies. Furthermore, there may be support for utilizing the specialized co-processor instructions, also may be manual or semi-automatic. Figure 3-3 shows the combined process and physical architectural views for threading-based (like POSIX threads) parallel processing.

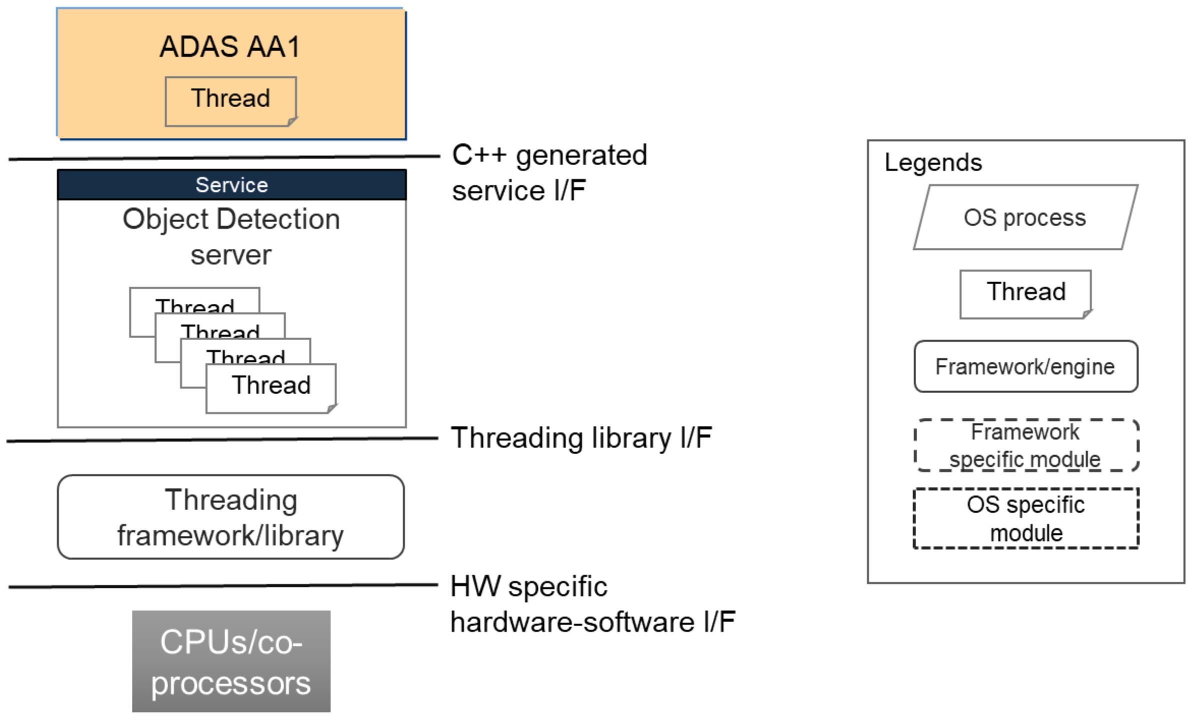

**Figure 3-3 CPU/co-processor-model example**

## 3.3 Rationale: decoupling of parallel processing specific knowledge from application development

Understanding the specifics of non-general computing hardware requires specific skills. As previously mentioned, the parallel processing technologies are still actively being developed and evolving, it is hard at the best to understand all these. Some standardization effort, such as OpenCL, aims to ease this problem by setting up a hardware independent API set. However, in order to cover various types of hardware and also to fully exploit the hardware features for best performance, the OpenCL in general is very low-level API, essentially requiring the similar level of detailed hardware level knowledge.

Our proposed model of Service-based parallel processing decouple the required knowledge of parallel processing hardware from AA developers via AP service interfaces. This frees the AA developers from acquiring the specific hardware knowledge every time a new, more efficient, or more suitable hardware is introduced, and also allows the system designer to do so if that the introduction of such hardware yields better system design. At the same time, this decoupling also frees hardware designers to come up with a new, innovative parallel processing technologies, as long as they can provide the AP services required by the users.

## 3.4 Adaptive Platform methodology consideration

Service-based parallel processing approach will not introduce any new AP methodology. It uses already defined service interface description to define the services.

# 4 Non-functional design topics

## 4.1 Performance

One of the primary purpose of using parallel processing is to achieve higher performance. Since “Service-based parallel processing” utilizes the SOA of AP, the general performance related design techniques also apply.

### 4.1.1 Interface granularity and communication overhead

The granularity of interface is the size of operation unit per API. If granularity is small, the service has many API. Finer the granularity, the service is more flexible in general, because the small granularity will allow different application to optimize its use.

In SOA, increasing the granularity will increase the communication between the client and server in general. A mechanism such as caching exists to circumvent, however, in real-time system such as AP, caching increases non-determinism, thus not a convenient choice.

In AP, there are two possible approaches to minimize the overhead of services. One is to make the service interface as coarse as possible, especially for the interface that has high frequency in its usage. For example, instead of providing an interface only for processing one datum, providing another interface for processing a batch of data at a time is recommended. The other approach is to optimize at the service interface library. This means that the service interface may cache some server side data for client-process local processing, and/or simple interfaces that sets up or read fields in an object stored locally in the client process heap. The two techniques can be mixed.

### 4.1.2 Data handling and throughput balancing

The overhead of moving very large data is costly. This is especially true if a lot of coping of data occurs. Often, parallel processing is used to perform processing large amount of data, and this is often performed against a stream of data, constituting a data-flow processing. It is therefore essential to design the whole chain of data flow, from a data-generating device, a device driver, a primary server to process the raw data, a secondary server to work on the primary server output, then finally an AA that uses the result of secondary server. One typical design to achieve the highest throughput is to have all these components forming PLP, each component forming a stage of a pipeline. For the servers that perform heavy computation, DLP and/or PLP is employed. The data that flows between the components have to be propagated in an efficient manner, avoiding copying of the data where possible.

## 4.2 Deterministic execution

If it is necessary to achieve high level of deterministic execution by the service-based parallel processing, the approach defined by [5] should be followed.

For CPU/co-processor-model, especially if there are enough processing elements to perform the redundant execution in parallel, the approach can be applied in a straight forward fashion. For accelerator-model, there are still the server process and also the main thread and some sub-threads, the main computation will be performed by the kernel executed on the accelerator. Although it varies, the accelerator is often only capable of executing a single kernel at a time. Therefore, it is not possible to perform the redundant execution in parallel, unless the accelerator is capable of running multiple kernels in parallel or by employing multiple accelerator units. With a single accelerator, one option is to perform the redundant execution in series. However, as this will impact the performance, a practical option is to abandon the redundant execution and take the system design approach of accepting the service-based parallel processing at ASIL-B, and another ASIL-B or higher sub-system for monitoring the result of accelerator-model service.

## 4.3 Safety considerations

Safety is a system design topic. The required ASIL for a “subsystem” depends on the system functionality it provides – e.g. parallel processing subsystem is used for ASIL-B system functionality, which computation result is safety-checked by ASIL-D subsystem. Or, one can go duplicate ASIL-B subsystem to achieve ASIL-D (though it may be expensive). The design guidelines of system design to achieve overall functional safety requirements are out of scope of this document.

To achieve the determinism that is essential for achieving ASIL-D by the service-based parallel processing, it is advised to follow the approach as discussed in “4.2 Deterministic execution”.

## 4.4 Prospects

This guideline, especially the parallel processing hidden under service model, should be capable of surviving long time to come, due to the intrinsic decoupling. There are two areas with foreseeable advancement in future; (1) AP standard application services and (2) more parallel processing directly within AA.

### 4.4.1 AP standard application services

It should be reasonable for one to expect such application services that uses parallel processing technologies to be standardized by AP. This indeed will not occur immediately, nor all services at once – however, even incremental introduction of such services should help both the providers of parallel processing technologies and also users of such. Higher level API standardization, that uses parallel processing technologies underneath, are already emerging in some areas, such as OpenVX.

### 4.4.2 More parallel processing within AA

Following the service-based parallel processing design, AA will use multiple services in parallel. As the number of services grow and if the AA remains single-threaded, then the AA itself can be a bottleneck in the whole processing chain. This will call for more parallel processing within AA eventually. AP already provides threading APIs of currently supported C++ standard and POSIX APIs, however, this may not be sufficient.

AUTOSAR AP adapts C++ standard, along with the CPP Coding Guidelines to use the language with safety and security in mind. The C++ standard is incrementally introducing parallel processing. The most of both open source and commercial compilers support the standard and widely used in the industry. Therefore, it is foreseeable and perhaps promising to introduce the more parallel processing technologies as the C++ standard progresses. One potentially promising standard is

SYCL, as it is purely based on standard C++ with template libraries to write parallel processing, part of it being introduced in C++17. The single source approach with the standard language and also capable of mixing with normal C++ multi-threading, may help to consolidate the situation in future.

# 5 References

[1] Glossary, AUTOSAR_TR_Glossary.pdf.
[2] Main Requirement, AUTOSAR_RS_Main.pdf.
[3] Methodology for Adaptive Platform, AUTOSAR_TR_AdaptiveMethodology.pdf.
[4] Explanations of Adaptive Platform Design, AUTOSAR_EXP_PlatformDesign.pdf.
[5] Specification of Execution Management, AUTOSAR_SWS_ExecutionManagement.pdf.