Threading frameworks analysis with respect of production requirements, such as FuSa/ISO 26262

Maxym Dmytrychenko
Developer Relations Division
Intel
Munich Germany
maxym.dmytrychenko@intel.com

Ilya Burylov
Developer Product Division
Intel
Nizhny Novgorod Russia
ilya.burylov@intel.com

ABSTRACT

During this evaluation we are going to share major points of production requirements for threading models, as well as our recommendations while working on IA and with software stack Intel provides, such as Threading component of Intel® Autonomous Driving Library (Intel® ADL), C++17 parallel algorithms (Parallel STL) and Intel® Threading Building Blocks (Intel® TBB). This analysis and recommendations are not final and can be changed as we progress.

KEYWORDS

Threading, multicore, safety, production, safety,FuSa/ISO, API

ACM Reference format:

Maxym Dmytrychenko and Ilya Burylov. 2018. Threading frameworks analysis with respect of production requirements, such as FuSa/ISO 26262

1 Introduction

The objective of functional safety (FuSa) is freedom from unacceptable risk of physical injury or of damage to the health of people. Functional safety is inherently end-to-end in scope. Modern systems often have software intensively commanding and controlling safety-critical functions. Therefore, software functionality and correct software behavior must be part of the Functional safety engineering effort to ensure acceptable safety risk at the system level.

In particular, the standard ISO 26262 (Road Vehicles Functional Safety) addresses the automotive development cycle. It is a multi-part standard defining requirements and providing guidelines for achieving functional safety in E/E systems installed in series production passenger cars. The standard ISO 26262 is considered a best practice framework for achieving automotive functional safety.

2 Performance considerations

Software applications designed for Functional safety avoided multi-threading in the past for sake of simplicity of implementations. Nowadays we see a rapid increase of complexity of algorithms to satisfy high demands in autonomous driving space and single CPU core cannot always provide performance required to deliver results in predefined time budget. Majority of the algorithms therefore need to consider multi-threading implementation to utilize multiple cores available in modern CPUs.

Therefore different SW APIs, such as Parallel STL, Intel ADL threading API or Intel TBB should be considered to be used within FuSa application to distribute work across CPUs in multi-threading fashion.

3 Completeness for requirements

FuSa main goal in appliance to the software is to minimize the risk of malfunctioning of applications on different level of the systems. Different considerations should be taken to fulfill FuSa requirements, where we would like to concentrate on the following ones, primary related to deterministic execution time and deterministic results of computations with dynamic memory allocation and error handling mechanism reviewed in more details.

4 Deterministic execution time

Threading model is being applied to FuSa application when there are risks of not meeting predefined time budget and it is expected that the application will run faster as a result. In many cases that is true, but FuSa requires providing evidences that safety goals, time budget in this case, are to be met. In many threading use cases even weaker requirement of progress guarantee is an open questions.

Such potential issues like dead locks, live locks, data races, etc., which can be easily introduced into multi-threading application makes a domain of low level threading API, such as pthreads or C++11 threading API an area for threading experts only. At the same time, correctly designed and properly used high level APIs for parallelism help to avoid these typical issues.

FuSa C++ guidelines for parallelism are yet in development stage, but generally speaking, it is better to avoid using low level API (such as threads and locks) in favor of high level approaches such as Parallel STL, leaving the risks in the hands of experienced higher level library developers.

A proper way to guarantee a meeting of time budget beyond historical statistical measurements on previously available use

cases is an area for research. In some cases, requirements to application components go even beyond the evidence of meeting time budget, targeting deterministic time of execution, which is even greyer area for multi-threading domain.

4 Deterministic results of computation

There is no direct requirement in FuSa for results of software components to be run-to-run reproducible. However, such requirements typically arise during architecture design of the system, whether in order to implement the mechanism of parallel redundancy with further results comparison or for general simplification of system analysis. Such requirements raise the design complexity of multi-threaded application, and is yet another argument or using higher level threading APIs, which can provide relevant guarantees.

An example of high-level guarantees on threading API level can be the difference between Intel TBB functions: tbb::parallel_reduce and tbb::parallel_deterministic_reduce. Both functions provide the same functionality of parallel reduction over the user provided range, but the former one guarantees deterministic behavior with regard to splits and joins. Intel ADL is specifically designed to fulfill similar high-level requirements for the threading API. Another example can be in control of floating point environment state in worker threads and individual parallel tasks.

It is important to remember, that threading is not the only source of non-deterministic results of computation results non-determinism. E.g. compiler may generate several paths for same blocks of codes to optimize performance, depending on variably aligned input arguments.

5 Dynamic memory allocation

Dynamic memory allocation is one of the key areas, which require special attention within FuSa. Dynamic memory allocation problem in general has two related sub-problems. The first one - there should be worst-case upper estimation for the total amount of memory to be allocated. The second one – some memory usage patterns tend to cause memory fragmentation, which results in hidden increasing demands to total memory in the system, especially visible in long uptimes of the applications.

Most existing threading solutions are designed based on free usage of dynamic memory allocation with average memory consumption in mind without deep worst-case analysis.

6 Dynamic memory allocation

FuSa standards specify a detailed hierarchy of possible error conditions and bring in specific requirements for the FuSa applications. Multi-threading solutions are not significantly different from all other software components in respect of necessity to apply specific analysis on possible error conditions, but there several additional specific challenges, which are introduced in error handling mechanism.

Since the error can happen in different threads, it is important to make sure that worker threads errors are not lost on higher application level. Simple moving error from worker-thread to some master-thread brings a topic, which closely related with determinism of the result, where our result is an error.

Requirement to implement a mechanism of graceful degradation within a strict time-budget brings a topic of cancellation. Cancellation mechanism is useful, when some error condition met in concurrent task, and we can cancel remaining work to allow more time for recovery mechanisms, which brings in additional complications into result and time determinism of the execution.

The typical choice of C++ exceptions as a main error handling mechanism to introduce in C++ library, has own issues with time determinism and dynamic memory allocation underneath, which should be additionally analyzed.

7 Software implementation model

There are many technical options to use threading within software implementation of product, such wide variety of APIs, should be reviewed on FuSa solution level and a subset APIs should be chosen which fulfills the safety goals imposed on the product. Below shown list of high level threading frameworks only:

Intel TBB is a time-proven threading library with variety of high and low level APIs. These APIs should be reviewed on FuSa solution level and a subset APIs should be chosen which fulfills the safety goals imposed on the product. Also, Intel TBB is using dynamic memory allocation underneath and corresponding safety analysis should be made on FuSa product level to analyze corresponding safety implications.

Intel ADL is a new product under development, which follows ISO26262 process. This library is going to provide high level API, which minimize the safety risks. Dynamic memory allocation is avoided by design, and the safety manual clearly specifies the safety implications of the solution.

Parallel STL is another possible option. This library provides a variety of standard C++ high level functions, which minimize the safety risks of low level API usage. Additional safety analysis should be done to ensure underlying threading back-end used to enabling threading execution.

OpenMP based implementations have strong binding for latest specification changes, compiler and appropriate runtime support. As well as certain specifics when used in high level/C++ language based implementations. There is an existing activity to investigate functional safety implications on OpenMP specification.

8 Final conclusions and future work

Final decision for implementation should be based on safety goals of the applications, additional safety analysis of use cases, implementation language style preferences and provided documentation by corresponding product as it is yet too early to

strongly recommend any particular approach well adopted through the industry and production.
There is ongoing work to address not yet fully covered aspects. Clear recommendations for the selection should be expected.

Declaimer

REFERENCES

[1] S. Royuela, A. Duran, M. Serrano, E. Quinones, X. Martorell, A Functional Safety OpenMP* for Critical Real-Time Embedded Systems, https://www.bsc.es/research-and-development/publications/functional-safety-openmp-critical-real-time-embedded-systems.
[2] C++17 ISO/IEC 14882:2017 standard/Execution policy Parallelism TS version 2, https://www.iso.org/standard/68564.html.
[3] ISO 26262/"Road vehicles – Functional safety", https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en.
[4] Intel® Threading Building Blocks (Intel® TBB), https://www.threadingbuildingblocks.org/.
[5] Intel® Automated Driving SDK, https://software.intel.com/en-us/automated-driving-sdk.