Experiences with HPX on Embedded Real-Time Systems

Remko van Wagensveld
Technische Hochschule Ingolstadt
Esplanade 10, 85049 Ingolstadt
Germany
Email: remko.vanwagensveld@thi.de

Ulrich Margull
Technische Hochschule Ingolstadt
Esplanade 10, 85049 Ingolstadt
Germany
Email: ulrich.margull@thi.de

Abstract—**Recently more and more embedded devices use multi-core processors. For example, the current generation of high-end engine-control units exhibit triple-core processors. To reliably exploit the parallelism of these cores, an advanced programming environment is needed, such as the current C++17 Standard, as well as the upcoming C++20 Standard. Using C++ to co-operatively parallelize software is comprehensively investigated, but not in the context of embedded multi-core devices with real-time requirements. For this paper we used two algorithms from Continental AG's power-train which are characteristic for real-time embedded devices and examined the effect of parallelizing them with C++17/20, represented by HPX as a C++17/20 runtime implementation. Different data sizes were used to increase the execution times of the parallel sections. According to Gustafson's Law, with these increased data sizes, the benefit of parallelization increases and greater speed-ups are possible. When keeping Continental AG's original data sizes, HPX is not able to reduce the execution time of the algorithms.**

I. Introduction

In 2022 "multi-core will be everywhere" [1, p. 45]. The automotive industry is already using multi-core processors for the power-train control systems, which have real-time requirements. This evolution calls for programming environments which are optimized for exploiting multi-core processors for embedded devices with real-time requirements. C++17 [2] has additions to the parallel execution of algorithms in form of the Parallelism Technical Specification (TS) [3]. Additionally, the Concurrency TS [4] adds ways to manage the concurrent execution of tasks. However, is is not yet merged into the C++ Standard. Both changes are implemented by HPX [5], a general purpose C++ runtime system based on ParalleX [6].

In this paper we will investigate if HPX is suitable to be used in the automotive industry for embedded systems. These real-time systems usually execute algorithms which are not arithmetical intense and use only small data sets; also the hardware used has very little resources. These circumstances make a parallelization more difficult [7]. This is a completely different set of problems, compared to the common circumstances in High Performance Computing for which HPX is optimized.

Additionally, the current generation of power-train engine-control systems conform to the AUTOSAR [8] standard and are therefore programmed in C. The standard's upcoming extension Adaptive AUTOSAR [9], allows using C++ as a programming language for these kinds of systems. Therefore, it is necessary to know if the use of C++ language features would be beneficial for the current generation of engine-control systems.

This paper is structured as follows. First, the current state of research is shown, and afterwards we summarize some new concepts in C++17 and C++20, as well as HPX. After that we present the algorithms of our case studies and our parallelization approach respectively. Subsequently, we present and discuss our measurement results. Finally, we will address our examined overhead with HPX.

II. Related Work

This chapter describes the current research in the field of parallel programming environments and the usage of C++ for embedded real-time systems.

There is a multitude of ways to exploit parallelism and concurrency in C++. OpenMP [10] is a library for parallel shared-memory systems that lets the user expose parallelism via pragmas in the source code. OpenMP is widely used, but is not suitable for embedded systems and needs additional extensions to be usable as parallel programming model for such systems [11].

Intel developed the Threading Building Blocks (TBB) [12], a C++ template library providing access to parallel algorithms, data containers, locks and atomic operations, a task scheduler, and a memory allocator [13]. Microsoft developed the Parallel Patterns Library (PPL) [14] which provides parallel algorithms. PPL is built on top of the Microsoft Concurrency Runtime which provides a Task Scheduler and a Resource Manager [15]. The PPL only works on Microsoft Windows as an Operating System (OS) and is thus not suitable for embedded systems where GNU/Linux is common. Both Intel TBB and Microsoft PPL have an Application Programming Interface (API) that is not standardized, which makes the code reliant on the library implementation. The use of C++17 does not have such a constraint since every runtime implementing C++17 will be able to execute a C++17 conform program.

ISBN 978-80-261-0642-5, © University of West Bohemia, 2017

Another research field that should be addressed is the use of C++ on embedded devices. Most of the advanced C++ features introduce either no or a fixed overhead [16, p. 78ff.]. This makes the execution time for these features predictable and therefore suitable for use in real-time systems. However, dynamic memory allocation, dynamic data-type casting, and exceptions introduce a low, but undetermined overhead. This prevents a timing prediction [16], and thus they are not recommended for use in real-time systems [17, p. 6]. These concepts are not used in the automotive industry and therefore C++ does not have a disadvantage in comparison to C for real-time systems.

When comparing C++ with the Real-Time Specification for Java (RTSJ), it is shown that the "C++ programming model is simpler and safer" than RTSJ. [18, p. 1]. The reason for this is the missing Garbage Collection when using RTSJ with hard real-time, where on the other hand, C++ implicitly deconstructs the objects when leaving scope [18]. This helps to prevent memory leaks and access of data after its release. C++ also prevents illegal memory mutation by the usage of constant references [18] which increases the integrity of the data.

Research has been done in the field of C++ on embedded devices with real-time requirements. But, to the best of our knowledge, there are no studies covering parallelism with C++ on embedded devices with real-time requirements.

III. HPX and C++17/20

"HPX is a general purpose C++ runtime system for parallel and distributed applications of any scale" [19]. It implements the ParalleX execution model [6]. HPX uses lightweight userspace threads to minimize the overhead needed for thread creation and scheduling. At startup HPX allocates a predefined number of system cores and executes the HPX task scheduler on each of those cores. The task schedulers are able to schedule and execute HPX lightweight threads on the allocated core.

When executing a parallel algorithm, the lightweight threads needed are created and put into the queues used by the task schedulers. The task schedulers then execute the next lightweight thread in their queue.

HPX implements C++17 and C++20 language features, especially those introduced in the Parallelism TS and the Concurrency TS.

The Parallelism TS adds execution policies to the language and is merged into C++17 [2]. An execution policy defines how an algorithm can be executed, for example, if it is able to be executed in parallel or has to be executed sequentially. These policies are added to the method signature of algorithms in the standard library in order to enable parallel versions. This enables a very elegant way to parallelize legacy software.

Before discussing the Concurrency TS, which is not yet merged into the C++ Standard, a short overview of the *future* class, which has been extended by the Concurrency TS, is given. A *future* object saves the state of an asynchronous operation and is ready when the operation has been finished. The operation can return values which are then saved in the enclosing *future* object. Through this concept it is possible to declare a value that is required in the "future" but not now. Therefore, when the value, represented by the *future* object, is needed, the `future.get()` method is called, which blocks until the value is ready. This is equivalent to a synchronization of the current calling thread and the asynchronous operation.

In the Concurrency TS the concept of continuations is added into C++. When using a *future* to save the state of an asynchronous operation, it is now possible to continue the asynchronous operation with another operation by the use of `future.then(chained_operation)`.

Additional to the continuations, in C++20 it is possible to create a *future* that gets ready when all *future*s of a set are ready. To accomplish this, `future.when_all()` is used.

In combination with the continuations there exists an elegant way to describe complex task graphs.

IV. Case Studies

For our work we identified two key algorithms used by the automotive industry with parallel executable sections. These algorithms also have been examined by Hartmann *et al.* [20] for the use in a GPGPU environment.

The first algorithm is a curve-sketching algorithm and the second is a finite element simulation. In sequential form, the curve-sketching algorithm is already in production and the finite element simulation is currently in pre-production. In this section the two examined algorithms are presented.

A. Curve-Sketching Algorithm

The first examined algorithm is a curve-sketching problem where the challenge is to find a specific inflection point in a given characteristic. The characteristic is a voltage curve measured by an Analog-to-digital-Converter (ADC). After the calculation of the second derivation, the second biggest negative area under the curve marks the bend. The biggest negative area under the second derivation curve results from triggering the mechanical device which overshoots and can therefore be ignored.

In Fig. 1 a typical characteristic is shown, as are the corresponding first and second derivation. The data for the characteristic is given in discrete values.

Only the second derivation is needed, therefore the *central difference approximation* for $f''(x)$ is used, as shown in Equation (1).

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \qquad (1)$$

This calculation is done for every point in a curve. The calculations have no dependencies on each other and the results are written in a second array; therefore, the calculation of the derivation can be executed in parallel for every point. To achieve this, the parallel *for*-loop

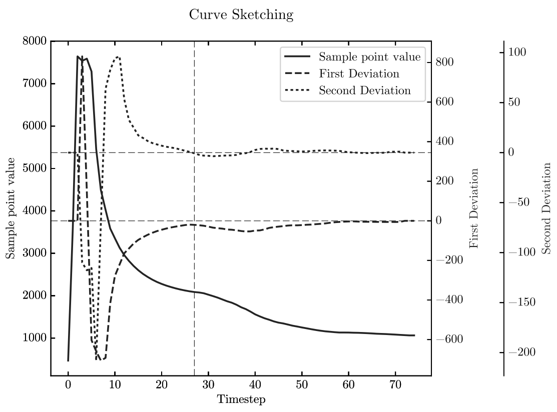

Fig. 1. Curve-Sketching with bend as vertical dashed line

was used, shown in listing 1. This loop works like a regular *for*-loop but iterations are executed in parallel.

However, the calculation of the second derivation needs to be finished in order to determine the roots of the function. This synchronization is necessary for keeping the data consistent. Because every step in this algorithm builds upon the preceding step, a synchronization is inevitable between each step of the algorithm.

```
int h = 3;
hpx::parallel::for_loop(policy, h,
 ↪  f.size() - h, [&](int x) {
        derivation[x] =
         ↪  double(f[x+h] - 2 * f[x]
         ↪  + f[x-h])/(h*h);
});
```

Listing 1: Implementation of the derivation with HPX

The integration of the curve is not data intensive enough to be successfully parallelized and is therefore not executed in parallel.

B. Finite Element Simulation

The second algorithm is a simulation using the finite element method (FEM) which calculates physical quantities inside a mechanical subsystem of the engine. This simulation helps to reduce the fuel consumption of the engine, but is not in production because of its execution time, which currently exceeds the relative deadline, which is the simulated discreet time step. Due to confidentiality reasons, the structure of the algorithm will only be discussed abstractly.

The system contains a set of mechanical components with physical quantities that are simulated by solving associated differential equations. The simulations for the separate mechanical components are implemented as a set of function calls, where each function call accesses a different data set to calculate the differential equation of that specific mechanical component. Therefore, these function calls can be executed fully in parallel. The used hardware, described more in detail in Section V-A, has four cores, and therefore we decided to bundle the method calls into four lambda functions and execute them in parallel.

```
std::vector<hpx::future<void>> f(4);
f[0] = hpx::async([&] {
  calc_diff_eq(component1);
  // ...
  calc_diff_eq(component8);
});
// ... similar for f[1] and f[2]
f[3] = hpx::async([&] {
  calc_diff_eq(component24);
  // ...
  calc_diff_eq(component31);
});
auto res = hpx::when_all(f.begin(),
 ↪  f.end());
res.get();
```

Listing 2: Calculation of the differential equations; each function call calculates a different mechanical component

When executing the method calls in parallel, two problems arise. On the one hand, there is a larger scheduling overhead because every function call will be handled as a separate lightweight thread. On the other hand, the execution time of one function call is very short which further increases the impact of the scheduling overhead.

Our solution is shown in listing 2 with the creation and asynchronous execution of the four lambda functions which bundle a quarter of the total method calls. To synchronize the four parallel executed lambdas the *future*s are combined into one using `when_all`.

V. Measurements

In this section our measurement methodology is presented and the results of our tests are shown.

A. Methodology

The examined algorithms were executed on a Wandboard Quad [21] with four Cortex A9 Cores clocked at 1.2 GHz and 2GB DDR3-RAM. To get a distribution of the execution times, the algorithms were executed 10,000 times for each data size. To measure the timing behavior of an algorithm when more data would be available, the used data sizes are scaled from the original data set used in the automotive industry to a considerably larger data set. This increase in data size also increases the execution time for the sections which are executed in parallel, thus reaping a greater benefit for the parallel execution.

Besides measuring the execution time as a Linux userspace program we also executed the algorithms as a Xenomai [22] real-time enabled program. This did not yield a significant difference in execution time and is therefore not further mentioned.

When not otherwise noted, the algorithms were executed with four cores.

B. Curve-Sketching Algorithm

In Continental's original configuration, the measured curve includes 75 sample points. To examine the dependency between the amount of sample points and execution time, the measured curve with 75 sample points was increased in sample points by linear interpolation with factors ranging from one to the factor 10,000, a data set with 750,000 sample points.

Fig. 2 depicts the execution time depending on the amount of sample points ranging from 75 sample points up to 750,000 sample points. Fig. 3 is a detail of Fig. 2 to better illustrate the intercept of the sequential with the parallel execution curve. With low amounts of sample points, the sequential execution time is the lowest. In this case the parallel execution slowed down the original execution time. This is due to the overhead, that parallel execution introduces, while in an optimal scenario without overhead, everything executed in parallel would give a positive speed-up.

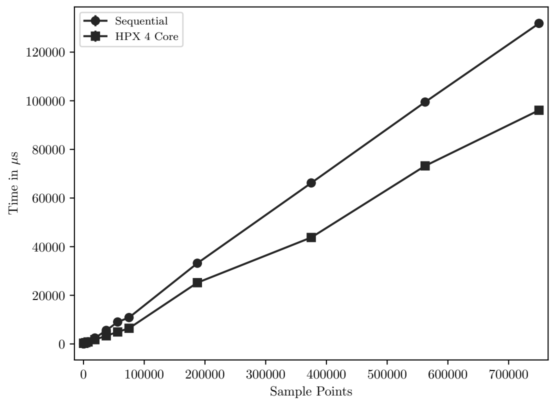

Fig. 2. Measurement results of the curve-sketching algorithm

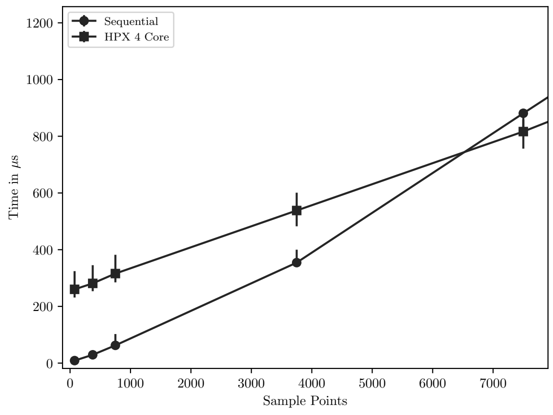

Fig. 3. Detail of the measurement results in Fig. 2

C. Finite Element Simulation

The measurement results for the finite element simulation is depicted in Fig. 4. In this algorithm the number of units inside a component can be increased – and thus the problem size – by making the FEM more fine-granular. The original number of units is 5, which was increased up to 150 units. The measurement results are similar to the results for the curve-sketching algorithm. With small data sizes the sequential execution time is lower than the parallel execution time and only with larger data sizes the parallel execution results in a significant benefit. Again, the original configured data

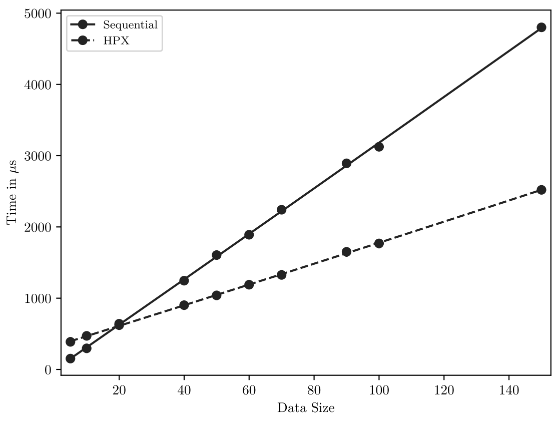

Fig. 4. Measurement results of the finite element simulation

size Continental uses is too small and, due to overhead, parallel execution is slower than the sequential one.

VI. Overhead Examination

To examine the overhead of HPX we measured a fully parallel program in sequential form without any HPX additions and in parallel form with HPX. HPX can be configured at startup with a total usable core count of one, up to all available cores.

The two algorithms from the case study were not suitable for analyzing the overhead because the inevitable synchronizations needed in the processing of the algorithms distort the amount of overhead. A program with synchronization has overhead caused by executing code in parallel and also overhead that is spent waiting on threads when synchronizing. To avoid the synchronization overhead, the chosen algorithm is fully parallel and only synchronizes when the grid is finished. It iterates over a two-dimensional grid and applies stencil operations to each element. The used stencil operation updates an element according to its north, east, south, and west neighbors.

The overhead results from the distribution of lightweight tasks to different cores. If only one core is allocated to HPX, the execution time consists both of the sequential execution time and the execution time spent on scheduling overhead. Fig. 5 depicts the results of the execution times measured once as sequential program and four times as a HPX-enabled program with different allocated core counts. The error-bars in Fig. 5 range from the 5% up to the 95% percentile. Note that, for a better overview, only every fifth data point is plotted with a marker and an error-bar.

The difference between the execution time of the HPX implementation with one core and the sequential implementation is the time HPX spends on overhead which is depicted in Fig. 6. This overhead augments with an increasing data size because of the additionally needed scheduling, which results from a higher amount of computation packages. In the range from 9,000 to 13,000 sample points, the difference increases. This is due to a caching effect that affects the execution with HPX with 1 core at 9,000 sample points, and the execution as a sequential program only just at 13,000 sample points. After sample point 13,000, the curve

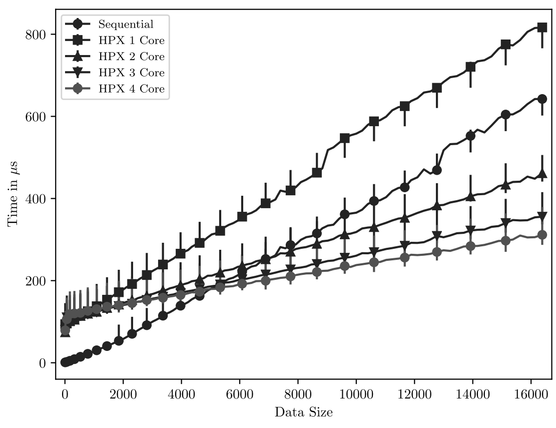

Fig. 5. Measurement results of the stencil operation

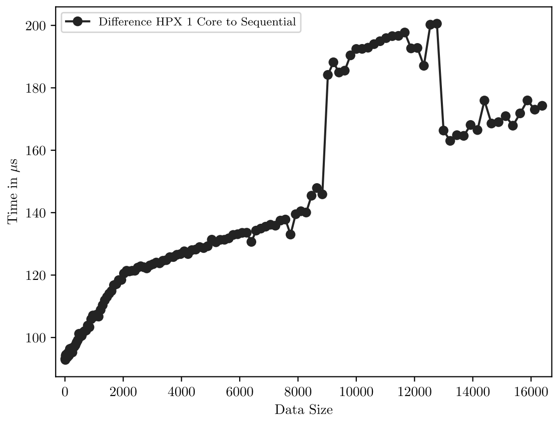

Fig. 6. Difference between HPX with one core and sequential execution, that is the overhead spent

goes back to the previous trend.

VII. Conclusion and Future Work

In this work we presented our results from using HPX on an embedded system. HPX and therefore C++ is a good solution to parallelize an algorithm because of the possibility to describe parallel sections elegantly inside C++ source code.

However, when used with the small data sets typical for many domains in the automotive industry, HPX has too much overhead resulting in an increased execution time compared to the pure sequential execution. For a useful application in these domains, the overhead required to schedule the tasks needs to be lowered in order to not slow down an algorithm by parallelizing it.

In future work we will investigate ways to minimize the overhead introduced by HPX to make it feasible for the usage on embedded devices with small data sets.

Acknowledgment

This research is part of the FORMUS^{3}IC Project (www.formus3ic.de) and has been funded by the Bayerische Forschungsstiftung.

The authors would like to thank Thomas Heller for his help with the setup and usage of HPX.

Special thanks for proofreading goes to Christoph Hartmann and Niklas Hehenkamp.

References

[1] H. Alkhatib, P. Faraboschi, E. Frachtenberg, *et al.*, “IEEE CS,” *IEEE Computer*, vol. 2014, Feb. 2014. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.500.211&rep=rep1&type=pdf (visited on 12/16/2016).

[2] “Working Draft, Standard for Programming Language C++,” The C++ Standards Committee, Mar. 21, 2017. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf (visited on 06/27/2017).

[3] J. Hoberock, “Programming Languages — Technical Specification for C++ Extensions for Parallelism,” The C++ Standards Committee, May 5, 2015. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507 (visited on 11/21/2016).

[4] Artur Laksberg, “Technical Specification for C++ Extensions for Concurrency, Working Draft,” Apr. 10, 2015. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4399.html (visited on 11/22/2016).

[5] H. Kaiser, B. Adelstein-Lelbach, T. Heller, *et al.*, *HPX: A General Purpose C++ Runtime System for Parallel and Distributed Applications of Any Scale (Commit af8e2717c3)*, Dec. 8, 2016. [Online]. Available: https://github.com/STEllAR-GROUP/hpx/tree/af8e2717c38a697d0b03f90a01641656bebdef5c.

[6] H. Kaiser, M. Brodowicz, and T. Sterling, “Parallex an advanced parallel execution model for scaling-impaired applications,” in *2009 International Conference on Parallel Processing Workshops*, IEEE, 2009, pp. 394–401. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5364511 (visited on 12/02/2016).

[7] N. Hehenkamp, R. v. Wagensveld, D. Schoenwetter, C. Facchi, U. Margull, D. Fey, and R. Mader, “How to Speed up Embedded Multi-core Systems Using Locality Conscious Array Distribution for Loop Parallelization,” in *ARCS 2016; 29th International Conference on Architecture of Computing Systems*, Apr. 2016, pp. 1–5.

[8] The AUTOSAR Foundation. (2016). AUTOSAR: Home, [Online]. Available: https://www.autosar.org/ (visited on 03/16/2017).

[9] ——, (2016). AUTOSAR: Adaptive Platform, [Online]. Available: https://www.autosar.org/standards/adaptive-platform/ (visited on 03/14/2017).

[10] The OpenMP Architecture Review Board. (2017). OpenMP, [Online]. Available: http://www.openmp.org/ (visited on 03/02/2017).

[11] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, “Implementing OpenMP on a high performance embedded multicore MPSoC,” in *2009 IEEE International Symposium on Parallel Distributed Processing*,

May 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5161107.

[12] Intel Corporation. (2017). Threading Building Blocks, [Online]. Available: https://www.threadingbuildingblocks.org/ (visited on 03/02/2017).

[13] ——, (2017). Intel TBB FAQs, [Online]. Available: https://www.threadingbuildingblocks.org/faq (visited on 03/02/2017).

[14] Mike Blome, Colin Robertson, Gordon Hogenson, and Saisang Cai. (Nov. 4, 2016). Parallel Patterns Library (PPL), [Online]. Available: https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl (visited on 01/28/2017).

[15] ——, (Nov. 4, 2016). Concurrency Runtime, [Online]. Available: https://docs.microsoft.com/en-us/cpp/parallel/concrt/concurrency-runtime (visited on 01/28/2017).

[16] Lois Goldthwaite, "Technical Report on C++ Performance," Jul. 15, 2004.

[17] B. Stroustrup, "Abstraction and the C++ Machine Model," in *Embedded Software and Systems*, Springer, Berlin, Heidelberg, Dec. 9, 2004, pp. 1–13. DOI: 10.1007/11535409_1. [Online]. Available: https://link.springer.com/chapter/10.1007/11535409_1 (visited on 03/02/2017).

[18] D. L. Dvorak and W. K. Reinholtz, "Hard Real-time: C++ Versus RTSJ," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '04, New York, NY, USA: ACM, 2004, pp. 268–274, ISBN: 978-1-58113-833-7. DOI: 10.1145/1028664.1028770. [Online]. Available: http://doi.acm.org/10.1145/1028664.1028770 (visited on 03/02/2017).

[19] Hartmut Kaiser, Bryce Adelstein-Lelbach, Adrian Serio, and Thomas Heller. (Dec. 2, 2016). HPX - The STEllAR Group, [Online]. Available: http://stellar.cct.lsu.edu/projects/hpx/ (visited on 12/02/2016).

[20] Christoph Hartmann, Ralph Mader, Lothar Michel, Christos Ebert, and Ulrich Margull, "Massive Parallelization of Real-World Automotive Real-Time Software by GPGPU," in *ARCS 2017; 30th International Conference on Architecture of Computing Systems*, Vienna, Apr. 4, 2017.

[21] The Wandboard Community. (2016). Wandboard - NXP i.MX6 ARM Cortex-A9 Community Development Board - Wandboard i.MX6, [Online]. Available: http://www.wandboard.org/index.php/details/wandboard (visited on 11/16/2016).

[22] Philippe Gerum. (2017). Xenomai – Real-time framework for Linux, [Online]. Available: https://xenomai.org/ (visited on 03/15/2017).