# HiPART: High-Performance Technology for Advanced Real-Time Systems

**Sara Royuela**
Barcelona Supercomputing Center, Spain

**Adrian Munera**
Barcelona Supercomputing Center, Spain

**Chenle Yu**
Barcelona Supercomputing Center, Spain

**Josep Pinot**
Barcelona Supercomputing Center, Spain

## Abstract

Cyber-physical systems (CPS) attempt to meet real-time and safety requirements by using hypervisors that provide isolation via virtualisation and Real-Time Operating Systems that manage the concurrency of system tasks. However, the operating system's (OS) decisions may hinder the efficiency of tasks because it needs more awareness of their specific intricacies. Hence, one critical limitation to efficiently developing CPSs is the lack of tailored parallel programming models that can harness the capabilities of advanced heterogeneous architectures while meeting the requirements integral to CPSs, such as real-time behaviour and safety requirements. While conventional HPC languages, like OpenMP and CUDA, cannot accommodate critical non-functional properties, safety languages, like Rust and Ada, are limited in their capabilities to exploit complex systems efficiently. On top of that, accessibility to the programming task is essential to making the system usable to different domain experts. HiPART tackles these challenges by developing a comprehensive framework holistically addressing efficiency, interoperability, reliability, and sustainability. The HiPART framework, based on OpenMP, provides tailored support for (1) real-time behaviour and safety requirements and (2) the efficient exploitation of advanced parallel and heterogeneous processor architectures. This support is exposed to users through extensions to the OpenMP specification and its implementation in the LLVM framework, including the compiler and the OpenMP runtime library. With this framework, HiPART will contribute to realising more capable and reliable autonomous systems across various domains, from autonomous mobility to space exploration.

**2012 ACM Subject Classification** General and reference → Cross-computing tools and techniques; Computing methodologies → Parallel programming languages; Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Parallel architectures

**Keywords and phrases** Cyber-physical systems, OpenMP, Parallel and heterogeneous architectures, Efficiency, Adaptability, Interoperability, Real-time, Resilience, Reliability

**Digital Object Identifier** 10.4230/OASIcs.PARMA-DITAM.2025.6

**Supplementary Material** *Software (GuardianOMP Tool for task-based replication with OpenMP)*: `https://anonymous.4open.science/r/GuardianOMP-4233/README.md` [18]
*Software (LLVM for OpenMP event-based synchronization in taskgraphs)*: `https://gitlab.bsc.es/ppc-bsc/software/llvm-taskgraph/-/tree/cudaGraph_RTevent` [4]
archived at `swh:1:dir:aa51e2f008a05c550f7f0f9b2359a13165b8094a`
*Software (LLVM for adaptive parallelism)*: `https://gitlab.bsc.es/ppc-bsc/research/c3po2024-dynamicvariants` [16]
archived at `swh:1:dir:a43d82b3d564a56f1a25347ff9a0beec81def2d5`

**Funding** This work is part of the HiPART project, with reference PID2023-148117NA-I00, financed by MICIU/AEI /10.13039/501100011033 and FEDER, UE.

16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025).
Editors: Daniele Cattaneo, Maria Fazio, Leonidas Kosmidis, and Gabriele Morabito; Article No. 6; pp. 6:1–6:15
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Acknowledgements** The authors want to thank the outstanding advisory board of HiPART, which includes Dr. Raúl de la Cruz (Collins Aerospace), Mr. Marc Blanch (Technica Engineering), Mr. Franck Wartel (Airbus Defence and Space), Dr. Michael Klemm (OpenMP ARB), Mr. Roland Weigan (European Space Agency), and Dr. Dirk Ziegenbein (Robert Bosch GmbH).

## 1 Introduction

The demands of our rapidly evolving society and the ever-expanding scope of industrial applications urge a substantial leap forward in the autonomy and intelligence of complex Cyber-Physical Systems (CPSs), like those used in autonomous mobility and space exploration. The increasing need for High-Performance Computing (HPC) capabilities coupled with the requirements regarding Real-Time (RT) behaviour exacerbates two critical challenges in CPS' development: (1) the coordination of a potentially extensive set of tasks that often require real-time execution and significant computational resources, and (2) the complexities entailed by the parallel and heterogeneous platforms upon which CPS are commonly deployed.

Nowadays, CPSs attempt to meet real-time and safety requirements through hypervisors [13] that provide isolation via virtualisation and Real-Time Operating Systems (RTOS) [11] that manage the concurrency of tasks that increasingly encompass dynamic and resource-intensive computations (e.g., AI flows). However, current software development environments fail to enable a comprehensive analysis of the entire system, impeding the efficient utilisation of highly parallel and heterogeneous architectures.

There is an urgent need in CPS development for parallel programming models tailored to harness the parallel capabilities of advanced processors efficiently while fulfilling the non-functional requirements (NFR) that are integral to CPSs. Unfortunately, conventional programming models, like OpenMP and CUDA, do not support essential NFRs, such as real-time behaviour (e.g., task deadlines and periods, predictability, and event-based execution) and safety requirements (e.g., functional correctness and resilience).

HiPART, depicted in Figure 1, originates in the limitations of current programming systems to provide an unified computing framework equipped with mechanisms for developing, deploying, and executing complex CPSs on parallel and heterogeneous architectures. HiPART considers a holistic approach that integrates primary requirements in CPS [10]: (1) *efficiency*, optimising the amount of resources (e.g., energy and time) required to deliver the expected functionalities; (2) *interoperability*, ensuring seamless compatibility and scalability, and support heterogeneity to compose various components into a cohesive system; (3) *reliability*, operating as expected, even under challenging conditions, providing robustness, availability, and predictability; and (4) *sustainability*, enabling adaptability, resilience, and reconfigurability. The HiPART framework is based on OpenMP and its implementation in the LLVM compilation framework, integrating extensions at the levels of the programming model, the compiler and the runtime system to meet real-time constraints, event-based execution, resilience, efficiency and adaptability.

## 2 State of the art (SoA)

HiPART builds upon three main pillars: parallel programming models for critical real-time computation, mechanisms to boost the performance and adaptability of the evolving CPS, and mechanisms to meet reliability and resilience expectations.

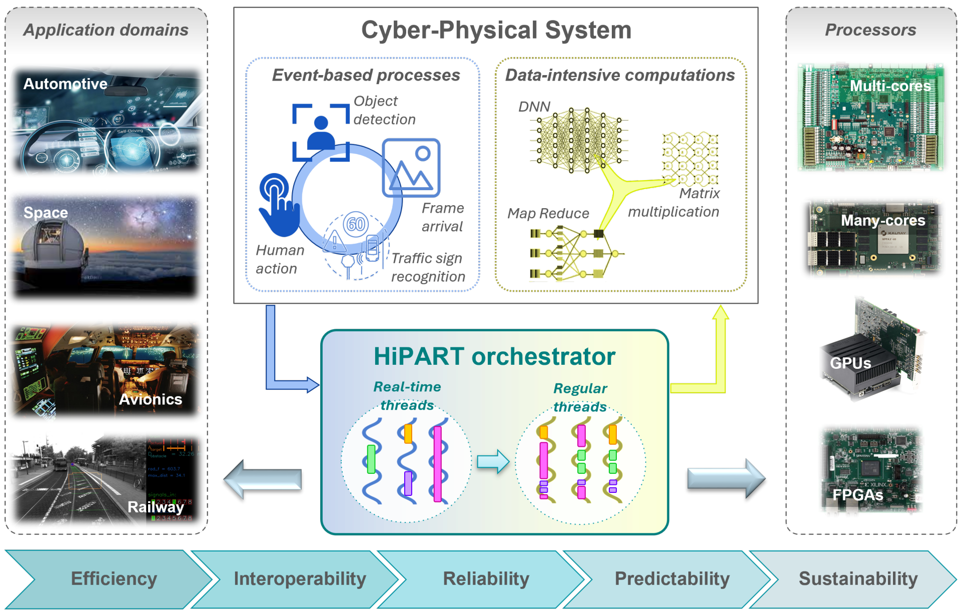


**Figure 1** HiPART's overview and intended application.

**Parallel programming models and real-time.** A programming language is selected based on factors like the system's requirements, performance constraints, and the desired level of control. Languages like Ada prioritise safety, with Ada Ravenscar for determinism and SPARK for formal verification. Despite their merits, these languages, except for the young Rust, fail to achieve popularity. Conversely, widely used languages like C/C++, with fine-grained control for performance-critical applications, and Python, offering a high level of abstraction that simplifies task development, are embraced for their versatility. Despite their popularity and support for parallelism, these languages are either inaccessible to non-expert programmers or cannot provide the required efficiency. Contrarily, models like OpenMP and SYCL are well-adopted for their efficiency but exhibit shortcomings in supporting NFR. Nonetheless, aspects such as real-time behaviour [28, 26, 5, 14] and correctness are already considered [33, 23, 22] for extensions to the OpenMP tasking model (see Section 3).

**Efficiency and adaptability in heterogeneous systems.** Over the past decade, GPUs have become popular in fields like scientific computing and machine learning, where parallel tasks are frequent. Although parallel programming models like OpenMP offer high-level interfaces with competitive productivity in heterogeneous platforms [7], they may fall short in performance compared to hand-tuned applications using low-level models like CUDA [3]. Challenges arise in achieving performance portability, scalability, and adaptability. New techniques, like highly dynamic task-based parallelism and asynchronous programming, aim to streamline the development process. However, the overhead introduced by the parallel orchestrator [9, 12, 34] and the lack of features to describe adaptability opportunities may impede optimal execution in evolving environments. This is critical in rapidly evolving heterogeneous architectures where specific devices might not be available or even present failures, and the overall conditions of the system constantly vary.

**Reliability and resiliency.** The development of trustworthy CPSs is challenged by the intricate interaction between the computational and physical realms. Schedulability [27] and fault-tolerance are integral in CPSs, given that missing deadlines and processing errors can potentially lead to catastrophic consequences. Techniques for space redundancy, like N-modular redundancy [30] and task replication [6], and time redundancy, like application-level checkpointing [31] may enhance fault-tolerance but also compromise the schedulability of the system [1] if tasks miss their deadlines due to increased computing requirements. Furthermore, these mechanisms need to be made aware of the structure of the applications and either require error-prone processes for manually determining the checkpointed data [29] or increase overheads and memory footprint due to poorly decided checkpoints.

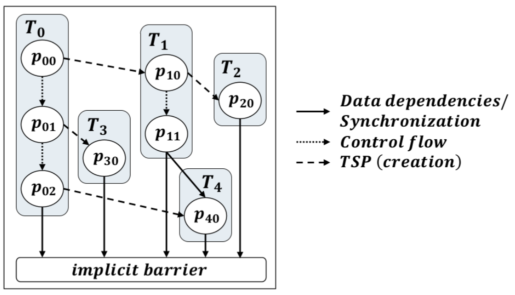
Considering the limitations mentioned above, HiPART leverages proposals and know-how from previous projects that have worked towards converging the HPC and the critical real-time domains using OpenMP, like AMPERE [19], RESPECT [21], HP4S [32], and the yet to finish RisingSTARS [24] and LIONESS [25] projects to revolutionize the landscape of complex CPS. Through extensions to OpenMP and an open-access implementation based on LLVM, HiPART will facilitate the design, deployment, and execution of real-time HPC CPS on advanced parallel and heterogeneous architectures, holistically addressing efficiency, interoperability, reliability, and sustainability.

## 3 The OpenMP programming model

HiPART builds on OpenMP, the de facto standard for programming shared-memory systems within the HPC community. The model defines an application programming interface (API) with compiler directives to annotate C/C++ and Fortran applications. Figure 2 shows an OpenMP task-based example, with a code snippet in Figure 2a and an extended task dependency graph (TDG) in Figure 2b describing the execution constraints among the different tasks. This example illustrates the most relevant features of OpenMP utilized in the HiPART project in the following paragraphs.

```
1  #pragma omp parallel num_threads(N)
2  #pragma omp single
3  #pragma omp task                        // T0
4  {
5    p00();
6    #pragma omp task depend(out:x)        // T1
7    {
8      p10();
9      #pragma omp task                    // T2
10     { p20(); }
11     p11();
12   }
13   p01();
14   #pragma omp task                      // T3
15   { p30(); }
16   #pragma omp task depend(inout:x)      // T4
17   { p40(); }
18 } // Implicit barrier
```

**(a)** Sample code.



**(b)** Task dependency graph (TDG).

**Figure 2** OpenMP tasking example.

OpenMP implements fork-join parallelism, i.e., a program starts sequentially until it reaches a `parallel` construct (line 1) and creates a team of threads associated with the parallel region where different mechanisms can distribute work. The *thread model* defines an abstraction of user-level threads that exposes low-level architectural details for exploiting loop-intensive applications. The *tasking model* provides a high-level abstraction to define independent regions of work, namely *tasks*. The *accelerator model* leverages the tasking model to offload tasks to accelerators and the thread model to exploit parallelism within the accelerator. Given its programmability and productivity [20] and the extensions proposed to adapt it to real-time systems (see Section 2), this work builds on the tasking model.

An OpenMP *task* (lines 3, 6, 9, 14, and 16) is an independent work unit with a block of executable code and its data environment. Tasks can be synchronized through memory fences, like the `taskwait` and `barrier` constructs, including the implicit barriers like those at the end of the single and the parallel regions (line 18), or the data-flow synchronizations defined by the `depend` clause coupled with the `in` (line 6), `out`, and `inout` modifiers (line 16). Tasks can also be nested, where each nesting level entails an isolated domain of synchronization. Task-based program commonly use the `single` construct to allow only one thread in the single region to execute the sequential code. Meanwhile, the rest of the threads wait in the implicit barrier at the end of the region until there is work to do (i.e., tasks are instantiated).

When a thread encounters a `task` construct at run-time, it creates a task instance that becomes ready when all its input dependencies are satisfied. In mainstream implementations, e.g., GCC and LLVM, ready tasks are dynamically scheduled by the runtime system to the available OpenMP threads. Furthermore, threads use work-stealing when they do not have work enqueued but other threads in the team still have work in their queues.

## 4 The HiPART framework: Extended OpenMP for real-time HPC

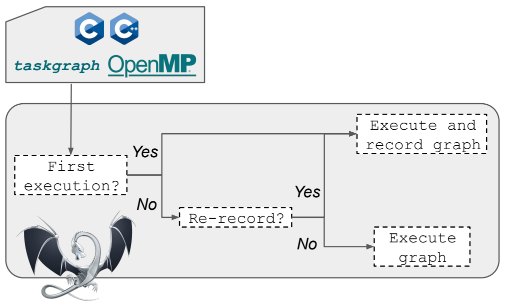
Building on the OpenMP tasking model (introduced in Section 3) and considering the limitations described in Section 2, the HiPART framework leverages and introduces extensions to the OpenMP specification and its implementation in LLVM, including the Clang frontend for processing new directives and clauses, the LLVM compiler for static analysis and code generation, and the OpenMP runtime library for runtime support, to address efficiency, interoperability, reliability, and sustainability. The reminder of this section introduces the extensions leveraged from previous projects and planned to be refined during the project (see Sections 4.1, 4.2, and 4.3) and the extensions already prototyped and preliminary tested since the beginning of the project in September 2024 (see Sections 4.4 and 4.5).

### 4.1 Graph-based execution

The recently released OpenMP6.0 [2] incorporates the *taskgraph* directive. This functionality implements reusable graphs of tasks to reduce the overhead of task orchestration (e.g., task creation) and minimise contention on shared resources (e.g., task ready queues). The functioning of taskgraphs is illustrated in Figure 3, with a sample code showing a `taskgraph` directive in Figure 3a and the execution flow depicted in Figure 3b. Overall, the `taskgraph` construct encloses a region of code that can be captured as a TDG. Hence, it includes task-generating constructs (e.g., `task` and `taskloop`) that are executed whenever the region is reached and other statements (e.g., control-flow statements) that will only be executed when the region is recorded. When a taskgraph region is encountered at runtime, if a graph of tasks already exists for that region, it is played. Otherwise, or if the user explicitly asks for regenerating the graph through the additional `graph_reset` clause (e.g., when the condition within the clause in line 5 resolves to true), then the system generates the TDG of the region. How the TDG is generated and executed is implementation-defined, so it can either be generated statically, at compile-time, or dynamically, at run-time. In the latter case, it can be created while executing the region or in a preprocessing step to later execute the graph.

```
#pragma omp parallel
#pragma omp single
{
  for (int i = 0; i < N; ++i) {
    #pragma omp taskgraph graph_reset(i%2==0)
    {
      #pragma omp task depend (out:a[i])
      produce_element (a[i]);
      #pragma omp task depend (in:a[i])
      consume_element (a[i]);
    }
  }
}
```

**(a)** Sample code.



**(b)** Workflow.

**Figure 3** OpenMP *taskgraph* example.

The taskgraph framework has already been tested using several HPC benchmarks including `task` and `taskloop` constructs. The results show speedups of up to 6x compared to the LLVM native implementation of tasking [34]. While upstream LLVM alreay implements a *record and replay* mechanism relying on runtime routines, a prototype implementation of the `taskgraph` construct including static (compile-time) and dynamic (run-time) recording capabilities is publicly available in `https://gitlab.bsc.es/ppc-bsc/software/llvm-taskgraph`.

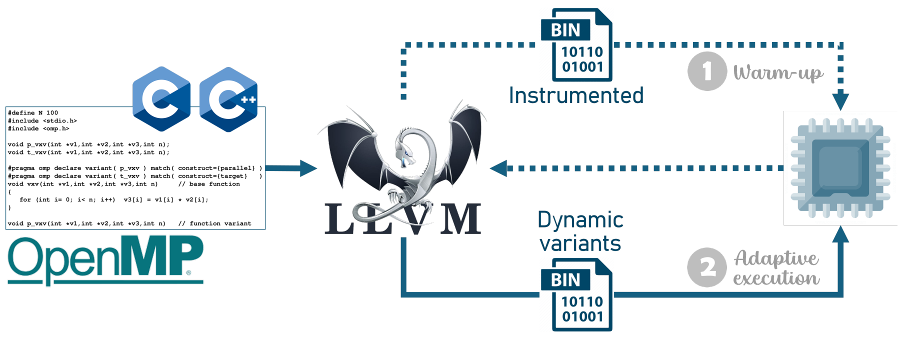
Graph-based computation has recently become popular as it allows reducing the overheads of task orchestration in CPUs [34] and enhancing the performance of CPU-GPU heterogeneous systems using CUDA graphs (for NVIDIA devices) or HIP graphs (for AMD devices). Leveraging the similarities between taskgraphs and CUDA and HIP graphs, an extension of the OpenMP taskgraph framework has been proposed to deploy OpenMP taskgraphs in GPUs using the GPU native graphs [35], i.e., CUDA or HIP graphs. The results show similar or better performance compared to original *target tasks* exploiting the target threading model due to the reduction of kernel offloads and an optimised orchestration of the tasks. Furthermore, the framework also shows better scalability with the number of processors.

## 4.2 Adaptability through function variants

Adaptability and performance portability in complex and changing heterogeneous systems cannot only be accomplished through compiler optimisations or runtime mechanisms. OpenMP includes features to define function specialisations. In the standard, different user functions can be linked together through the `declare variant` directive, which establishes different functions to implement a unique functionality and the condition that the compiler must check to statically decide which implementation is used in each function call. Although this presents a step forward, adaptability can only be obtained at compilation time. Therefore, runtime changes (e.g., a permanent failure in a device) cannot be considered.

HiPART relies on an extended interpretation of OpenMP variants that gathers metrics at run-time to dynamically decide among the set of function specialisations provided by the user [15]. This procedure allows for considering the system's dynamic conditions, like workload and energy consumption. To that end, the compiler follows the flow shown in Figure 4, producing two distinct binaries: (1) an instrumented version equipped with runtime calls that collect metrics about resource usage and (2) a version that interprets the metrics gathered by the instrumented version to guide variant selection. The second binary is generated only after the instrumented binary runs and collects the metrics.

The metrics captured during warm-up include average and peak CPU usage (%), average and peak GPU usage (%), thread stack memory consumption (%), heap memory consumption (%), GPU memory consumption (%), and execution time (ms). During the execution of



```
#define N 100
#include <stdio.h>
#include <omp.h>

void p_vxv(int *v1,int *v2,int *v3,int n);
void t_vxv(int *v1,int *v2,int *v3,int n);

#pragma omp declare variant( p_vxv ) match( construct={parallel} )
#pragma omp declare variant( t_vxv ) match( construct={target}   )
void vxv(int *v1,int *v2,int *v3,int n)     // base function
{
   for (int i= 0; i< n; i++)  v3[i] = v1[i] * v2[i];
}

void p_vxv(int *v1,int *v2,int *v3,int n)   // function variant
```

**Figure 4** Workflow of the extended LLVM to support dynamically selected function variants.

the final binary, users can provide a list of metrics to guide variant selection, represented as a set of comma separated triplets of the form of `metric:threshold:weight`, where `metric` is *cpu*, *gpu*, *mem*, *stack* or *heap*, `threashold` indicates a percentage that, when reached, forces the runtime to select the variant that stresses less the corresponding metric, and `weight` is an optional parameter that indicates the weight of the metric.

The evaluation of this proposal [15] shows the capability of the system to tune the optimal number of threads for each parallel region, prevent errors or slowdowns in systems with limited memory, and effectively swap between CPU and GPU implementations when one resource is overloaded. However, the method might introduce unbearable overheads in systems with rapid fluctuations, and decisions might soon be outdated. HiPART plans to mitigate these effects with mechanisms like splitting functions. Furthermore, other planned extensions include more refined predictive models that can foresee system state changes and new metrics like power consumption.

## 4.3 Predictable execution via task-to-thread mapping heuristics

The dynamic nature of the OpenMP task scheduler and the use of work-stealing to balanc the workload of all threads introduce uncertainty in the execution and, although good-enough in general terms, entail certain overheads. Recent works have proposed the use of temporal conditions to derive a more efficient task-to-thread mapping able to reduce system response time and running time and providing better predictability [26]. The scheduling mechanism proposed is depicted in Figure 5. It is split into two phases: (1) *allocation*, assigning each task to a thread, and (2) *dispatching*, selecting a task from the ready task queue. A series of heuristics are proposed for each of these phases, leveraging information about the number of tasks in a thread's ready queue and their execution time, among other aspects.

An evaluation of the proposed heuristics compares to common implementations in OpenMP runtime, including breadth-first scheduler (BFS) and work-first scheduler (WFS), regarding response time and running time. The results show that the response time produced by some heuristics is lower than the default LLVM scheduler in most cases, and the variability in the results given by the heuristics is lower than that of the default scheduler.

HiPART plans to extend this work by providing a schedulability analysis of state-of-the-art mapping strategies and the suggested heuristics in relevant applications. Furthermore, the project considers extending these mechanisms to heterogeneous systems with multiple GPUs.

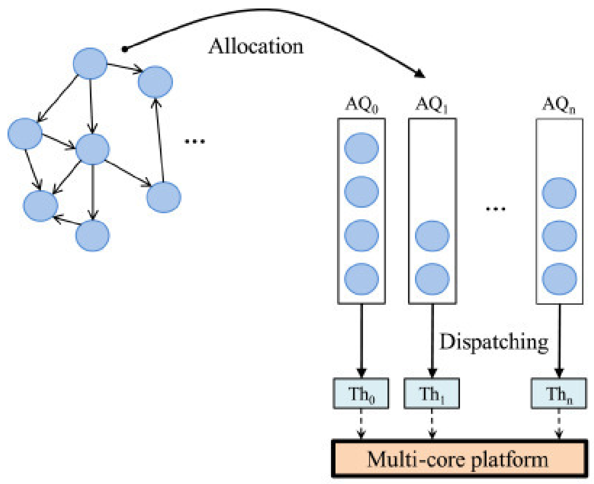


**Figure 5** Task-to-thread mapping workflow.

## 4.4 Fault tolerance

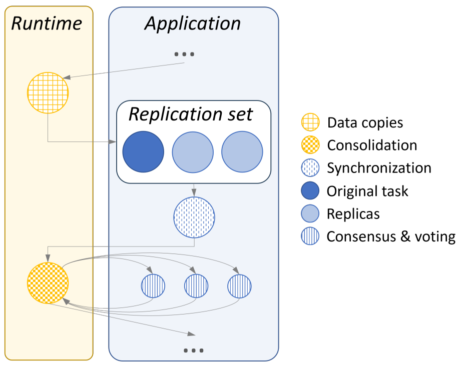
Several CPSs are sensible to transient faults due to the harm these may cause to the safety of people, the environment and the system itself. Unfortunately, models like OpenMP lack the mechanisms to provide fail-operational behaviour. Accordingly, HiPART is developing fault-tolerance techniques to enable software-based task-level replication and user-directed fault detection to mitigate the impact of transient faults.

The proposed fault-tolerance mechanism is based on an extension to OpenMP to define task replication. Figure 6 shows an example of such an extension. The sample code in Figure 6a illustrates the syntax proposed (lines 9 to 11) [17], where:

- the clause `replicated` specifies the number of replicas to create through an integer expression $> 1$ (3 in the example) that indicates the total number of tasks to be generated (including the original instance and the replicated instances), and the replication strategy through the `spatial`, `temporal`, or `spatial_temporal` optional keywords. *Spatial replication* forces each task to be executed on a different resource, such as a processor core or an architecture, allowing them to run in parallel; *temporal replication* ensures that tasks execute one after the other, enforcing sequential execution; and *spatial_temporal replication* combines both approaches, requiring tasks to run on different resources while also executing sequentially. If no replication type is specified, the default behaviour allows the tasks to run in parallel without restrictions, thus favouring performance.
- the clause `replica_private` defines variables that will be replicated for each task replica, i.e., each task will have its own copy of the variable. By making data private to replicas, this clause prevents data races and ensures each replica operates independently without causing inconsistencies in shared data.
- the clause `replica_firstprivate` specifies a list of variables that must be firstprivate to each replica (i.e., each task will allocate a new space and initialize its value to the value of the original variable at the time the task is instantiated). The compiler performs a shallow copy unless a shaping expression (e.g., $[size]a$, an array $a$ of $size$ elements) is defined, in which case it performs a deep copy considering the corresponding size. This clause ensures that the replicas start with a consistent state, improving fault tolerance while maintaining independent execution for each replica.

```
 1  void foo() {
 2     int v = 2;
 3     int size = sizeof(int)*10;
 4     int *a = (int *) malloc (size);
 5     #pragma omp parallel
 6     #pragma omp single
 7     {
 8         #pragma omp task \
 9             replicated(3, spatial) \
10             replica_private([size]a) \
11             replica_firstprivate(v)
12         {
13             for (int i=0; i<10; i++)
14                 a[i] = v;
15         }
16     }
17  }
```

**(a)** Sample code.



**(b)** Execution flow.

**Figure 6** OpenMP task replication proposal example.

The execution flow in Figure 6b illustrates the behaviour of the replication mechanism. The thread that encounters the task replicated (line 8) creates a *replication set* with three tasks: the original and two replicas. Since the spatial constraint is specified, the OpenMP runtime system prevents threads from executing multiple tasks within the same replication set. Threads can further be bound to cores through the `OMP_PROC_BIND` environment variable for the whole execution or the `proc_bind` clause, attached to a `parallel` construct.

Figure 7 shows preliminary results of the impact of using task replication in the Barcelona OpenMP Task Suite (BOTS) [8], a benchmark suite with eleven benchmarks exposing different memory profiles and CPU consumption when randomly inserting a single bitflip per execution in either memory or registers. The result of a bitflip can be a *benign fault*, when the application completes with the correct result without detecting any error, an *output error* when the application finishes normally but the output is incorrect, a *crash*, when the application terminates unexpectedly due to an internal error, or *timeout*, when the application hangs and does not complete. The results compare a bitflip in the sequential vanilla version, namely *vanilla*, with a bitflip in the replicated version, namely *GuOMP*. Considering that bitflips typically occur in the stack in applications with minimal memory usage, examples like *Floorplan*, *Alignment*, *Fib*, *Nqueens*, and *Knapsack* exhibit high tolerance to faults because the majority of the stack is not actively used. Oppositely, benchmarks like *UTS*, which uses about 1.5MB of stack memory, are more prone to crashes. On the other hand, applications with higher use of dynamic memory see more significant effects from bitflips. Still, this behaviour depends on the specific algorithms and their memory access patterns. For example, a single bitflip in the *Sort* integer array disrupts the output, as sorting algorithms depend on precise memory operations.

HiPART plans to extend the proposal for fault-tolerance in OpenMP with a user-defined consensus and voting mechanism that will provide better accuracy when comparing results from different replicas. Furthermore, replicas will be extended to exploit function variants to boost resilience in heterogeneous systems. Finally, to provide fault-recovery capabilities, HiPART plans to enable communication between the runtime system and the application through runtime routines and OpenMP data structures and offer a checkpointing mechanism to store selected memory objects. These methods combined will enable users to handle errors in the most adequate way for each application.

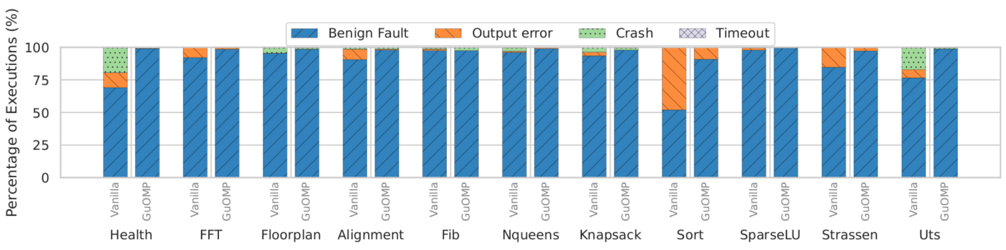
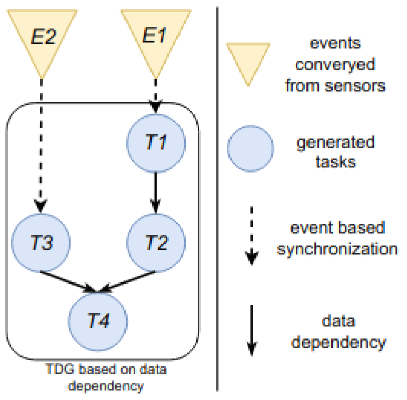


**Figure 7** Comparison of the consequences of memory bitflips in BOTS with sequential vanilla and task-replicated versions.

```
void tdg_computation() {
    #pragma omp target nowait \
        depend(out:dep1) attach(event1)
    {...} // T1
    #pragma omp target nowait \
        depend(in:dep1) depend(out:dep2)
    {...} // T2
    #pragma omp target nowait \
        depend(out:dep3) attach(event2)
    {...} // T3
    #pragma omp target nowait \
        depend(in:dep2,dep3)
    {...} // T4
}
void notify() {
    #pragma fulfill_event gpu_notify(event1)
    #pragma fulfill_event gpu_notify(event2)
}
```

**(a)** Sample code.


E2
E1
T1
T3
T2
T4
TDG based on data dependency
events converyed from sensors
generated tasks
event based synchronization
data dependency

**(b)** TDG.

**Figure 8** OpenMP event-based synchronization proposal example.

## 4.5 Real-time event-based execution

The throughput-centric design of GPUs poses challenges when integrating them into time-sensitive CPS. Modern systems recently evolved to minimize overheads and interference along the critical path through advanced mechanisms, such as CUDA graphs in NVIDIA devices and HIP graphs in AMD devices. However, GPU vendors provide ecosystems specific to their products, preventing code portability. HiPART has integrated event-based synchronizations into OpenMP and extended the support for OpenMP taskgraph to CUDA/HIP graphs to notably reduce interference and overheads in time-sensitive applications.

Figure 8 shows an example of the proposal for event-based synchronization in OpenMP. Figure 8a depicts an example of generating four interdependent target tasks, two synchronized with events. The corresponding TDG is shown in Figure 8b. Upon the encountering of an `attach` clause, the implementation creates a new *allow-launch* event and connects it to the beginning of the execution of the associated task region. The generated task can only start executing its associated structured block when the *allow-launch* event is fulfilled. This will happen when another thread encounters the `fulfill_event` directive with either the `cpu_notify` or `gpu_notify` clauses taking the same event as the argument.

Preliminary experiments measure the response time and time variability of the Adaptive Optics (AO) real-time controller illustrated in Figure 9. The application combines two functionalities: (1) a pixel processing method that takes raw data from wavefront sensor cameras and processes the pixels with a series of arithmetic kernels, and (2) a series of matrix-vector multiplications that produce a command as a vector sent to the deformable mirror actuators of the physical component, typically a telescope.

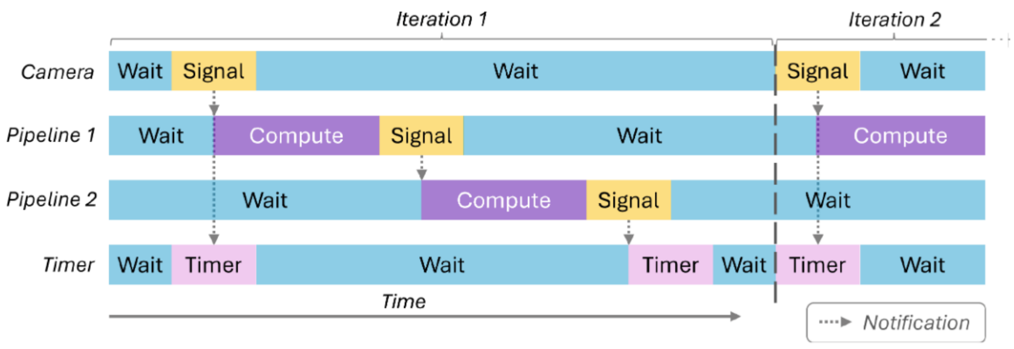


**Figure 9** AO pipeline.

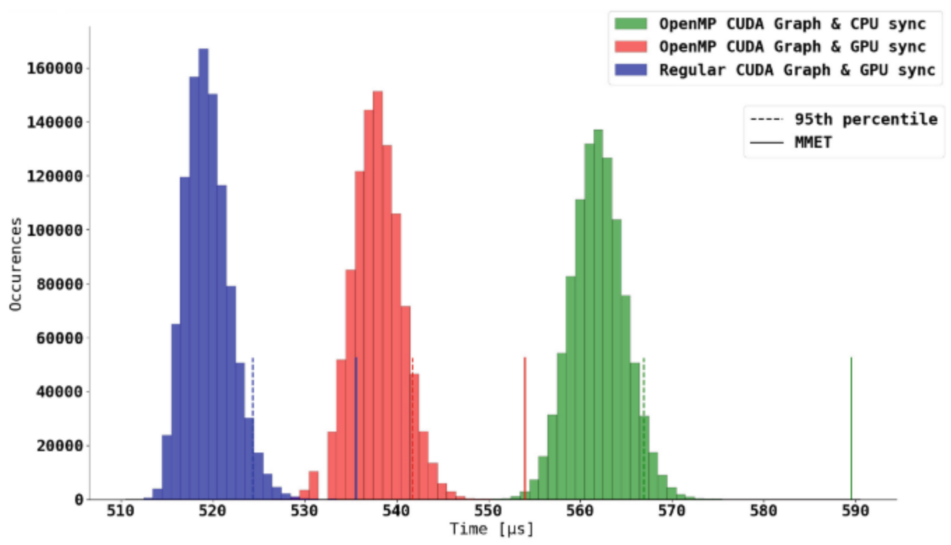
Figure 10 shows the response time of the AO application using CUDA (in Figure 10a) and HIP (in Figure 10b). In NVIDIA devices, our implementation (*OpenMP CUDA Graph & GPU sync*) achieves 16$\mu s$ max. jitter, with 538$\mu s$ mean execution time (MET) and 554$\mu s$ maximum measured execution time (MMET). Although the response time is slightly higher in our OpenMP CUDA version than the native solution (*Regular CUDA Graph & GPU sync*), the max jitter is identical, showing comparable predictability. In AMD devices, however, the OpenMP HIP version (*OpenMP HIP Graph & GPU sync*) achieves a slightly higher max. jitter of 21$\mu s$ (443$\mu s$ MET and 464$\mu s$ MMET). As expected, the third method using the CPU synchronization strategy showed the largest execution times and a max jitter of 28$\mu s$ (562$\mu s$ MET and 590$\mu s$ MMET) with CUDA and 33$\mu s$ (478$\mu s$ MET and 511$\mu s$ MMET) with HIP. All in all, our proposal delivers MET and jitter comparable to the native CUDA/HIP implementations while maintaining a simple, directive-based programming style.

## 5 Project plan and next steps

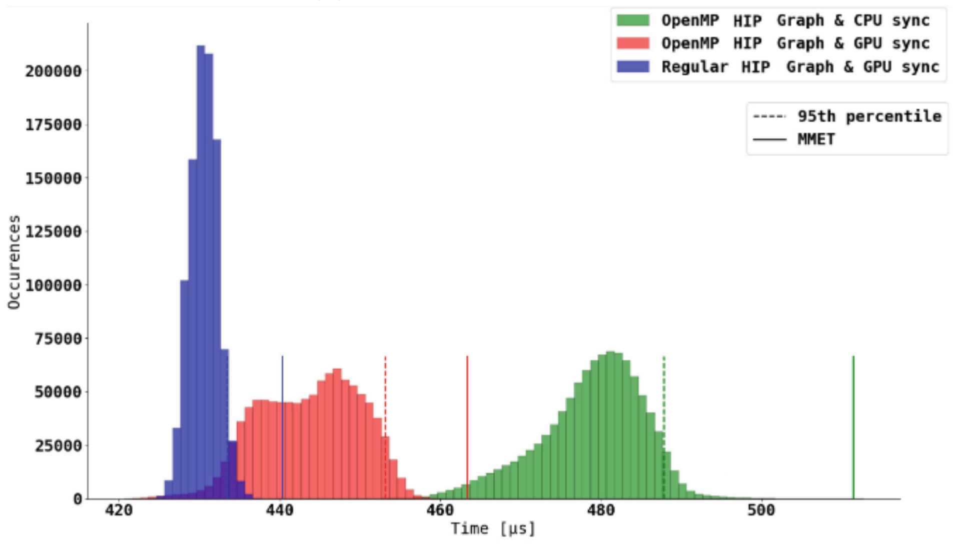
HiPART is organized into three technical Work Packages: (1) *WP1* is dedicated to developing and validating real-time high-performance systems through relevant use cases, (2) *WP2* is dedicated to developing extensions for high-performance, including performance and adaptability, and (3) *WP3* is dedicated to extensions for critical real-time systems, including resilience and predictability. Additionally, there is a dedicated WP for impact maximization and project management, ensuring continuous dissemination of key findings[1] and the smooth development of HiPART. All WPs span the 4 phases of the project, including (1) *phase 1* to define the use cases and the initial design, (2) *phase 2* for the development and isolated testing of software components, ensuring their individual functionality and compatibility with the targeted platforms, (3) *phase 3* for the integration and optimization of the parallel framework, and (4) *phase 4* for the validation and demonstration.

HiPART started in September 2024 and will expand three years of work. It is now in its initial phase, where use cases and system requirements are being defined. This paper presents the project and its intended solution for real-time high-performance systems. The project is developing a unique framework for efficiently deploying advanced CPS in parallel and heterogeneous processor architectures, holistically addressing real-time and HPC requirements. The project leverages and extends OpenMP to accommodate requirements from the critical

[1] Follow HiPART in the LinkedIn profile: `https://www.linkedin.com/company/105114452`.



**(a)** CUDA on NVIDIA.



**(b)** HIP on AMD.

**Figure 10** Histogram of response time.

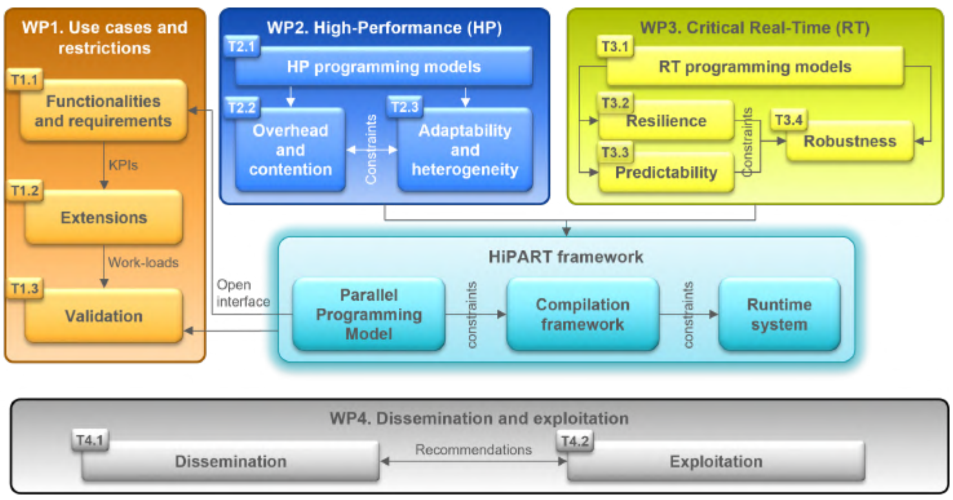


**Figure 11** Timeplan of the HiPART project.

real-time domain, including event-based execution, time predictability and resilience, and the HPC domain, including performance portability and adaptability. The project has preliminary results for task-based replication, showing enhancements in detecting and bypassing faults, and event-based execution exploiting heterogeneous OpenMP taskgraphs, showing competitive performance, equivalent jitter and much better programmability than native solutions. Future work includes further extensions for fault-recovery based on consensus-and-voting, N-version programming and checkpointing and scheduling mechanisms to enhance the predictability of heterogeneous systems, among others.

## References

1 Jaemin Baek, Jeonghyun Baek, Jeeheon Yoo, and Hyeongboo Baek. An N-modular redundancy framework incorporating response-time analysis on multiprocessor platforms. *Symmetry*, 11(8):960, 2019. doi:10.3390/SYM11080960.

2 OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 6.0, 2024. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf.

3 Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. A comparison of SYCL, OpenCL, CUDA, and OpenMP for massively parallel support vector machine classification on multi-vendor hardware. In *10th International Workshop on OpenCL*, pages 1–12, 2022. doi:10.1145/3529538.3529980.

4 Cyril Cetre, Chenle Yu, and Sara Royuela. LLVM for OpenMP event-based synchronization in taskgraphs. Software, version 1.0. swhId: swh:1:dir:aa51e2f008a05c550f7f0f9b2359a13165b8094a (visited on 2025-02-14). URL: https://gitlab.bsc.es/ppc-bsc/software/llvm-taskgraph/-/tree/cudaGraph_RTevent, doi:10.4230/artifacts.22923.

5 Cyril Cetre, Chenle Yu, Sara Royuela, Rémi Barrere, Eduardo Quiñones, and Damien Gratadour. Event-Based OpenMP Tasks for Time-Sensitive GPU-Accelerated Systems. In *International Workshop on OpenMP*, pages 31–45. Springer, 2024. doi:10.1007/978-3-031-72567-8_3.

6 Jian-Jia Chen, Chuan-Yue Yang, Tei-Wei Kuo, and Shau-Yin Tseng. Real-time task replication for fault tolerance in identical multiprocessor systems. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 249–258. IEEE, 2007. doi:10.1109/RTAS.2007.30.

7 Jose Monsalve Diaz, Swaroop Pophale, Kyle Friedline, Oscar Hernandez, David E Bernholdt, and Sunita Chandrasekaran. Evaluating support for OpenMP offload features. In *47th International Conference on Parallel Processing*, pages 1–10, 2018. doi:10.1145/3229710.3229717.

8 Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *International Conference on Parallel Processing*, pages 124–131. IEEE, 2009. doi:10.1109/ICPP.2009.64.

9 Thierry Gautier, Christian Pérez, and Jérôme Richard. On the impact of OpenMP task granularity. In *Evolving OpenMP for Evolving Architectures: 14th International Workshop on OpenMP (IWOMP)*, pages 205–221. Springer, 2018. doi:10.1007/978-3-319-98521-3_14.

10 Volkan Gunes, Steffen Peter, Tony Givargis, and Frank Vahid. A survey on concepts, applications, and challenges in cyber-physical systems. *KSII Transactions on Internet and Information Systems (TIIS)*, 8(12):4242–4268, 2014. doi:10.3837/TIIS.2014.12.001.

11 Prasanna Hambarde, Rachit Varma, and Shivani Jha. The survey of real time operating system: RTOS. In *International Conference on Electronic Systems, Signal Processing and Computing Technologies*, pages 34–39. IEEE, 2014.

12 Dian-Lun Lin and Tsung-Wei Huang. Efficient GPU computation using task graph parallelism. In *27th International Conference on Parallel and Distributed Computing*. Springer, 2021.

13 Santiago Lozano, Tamara Lugo, and Jesús Carretero. A Comprehensive Survey on the Use of Hypervisors in Safety-Critical Systems. *IEEE Access*, 11:36244–36263, 2023. doi:10.1109/ACCESS.2023.3264825.

14 Brayden McDonald and Frank Mueller. OpenMP-RT: Native Pragma Support for Real-Time Tasks and Synchronization with LLVM under Linux. In *25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 119–130, 2024. doi:10.1145/3652032.3657574.

15 Adrian Munera, Guerau Dasca, Eduardo Quiñones, and Sara Royuela. Adaptive parallelism in OpenMP through Dynamic Variants. In *High Performance Computing: ISC High Performance 2024 International Workshops*, 2024.

16 Adrian Munera, Guerau Dasca, and Sara Royuela. LLVM for adaptive parallelism with dynamic variants. Software, version 1.0. swhId: swh:1:dir:a43d82b3d564a56f1a25347ff9a0beec81def2d5 (visited on 2025-02-14). URL: https://gitlab.bsc.es/ppc-bsc/research/c3po2024-dynamicvariants, doi:10.4230/artifacts.22924.

17 Adrian Munera, Eduardo Quiñones, and Sara Royuela. GuardianOMP: A framework for highly productive fault tolerance via OpenMP task-level replication. In *International Parallel & Distributed Processing Symposium*, 2025.

18 Adrian Munera and Sara Royuela. GuardianOMP. Software, version 1.0. (visited on 2025-02-14). URL: https://anonymous.4open.science/r/GuardianOMP-4233/README.md, doi:10.4230/artifacts.22922.

19 Eduardo Quiñones, Sara Royuela, Claudio Scordino, Paolo Gai, Luís Miguel Pinho, Luís Nogueira, Jan Rollo, Tommaso Cucinotta, Alessandro Biondi, Arne Hamann, et al. The AMPERE Project:: A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimization. In *IEEE 23rd International Symposium on Real-Time Distributed Computing*, pages 201–206. IEEE, 2020.

20 Alejandro Rico, Isaac Sánchez Barrera, Jose A Joao, Joshua Randall, Marc Casas, and Miquel Moretó. On the benefits of tasking with OpenMP. In *15th International Workshop on OpenMP*, pages 217–230. Springer, 2019.

21 Sara Royuela. RESPECT - Reliable Heterogeneous Parallelism for Embedded Critical Systems, 2024. URL: https://sroyuela.github.io/respect-project/.

22 Sara Royuela, Alejandro Duran, Maria A Serrano, Eduardo Quiñones, and Xavier Martorell. A functional safety OpenMP for critical real-time embedded systems. In *13th International Workshop on OpenMP*, pages 231–245. Springer, 2017. doi:10.1007/978-3-319-65578-9_16.

23 Sara Royuela, Roger Ferrer, Diego Caballero, and Xavier Martorell. Compiler analysis for OpenMP tasks correctness. In *ACM International Conference on Computing Frontiers*, 2015.

24 Sara Royuela, Bartomeu Pou, Cyril Cetre, Rémi Barrère, Eduardo Quiñones, and Damien Gratadour. RisingSTARS, RISE International Network for Solutions Technologies and Applications of Real-time Systems. In *ISC High Performance*, 2023.

25 Sara Royuela, Franck Wartel, Sylvain Tiberio, Eric Jenn, Hubert Guérard, and Guy Bois. LIONESS – Improving and leveraging OpenMP for the efficient and safe use of new high-performance hardware platforms. *Ada User Journal*, 45(4), 2024.

26 Mohammad Samadi, Sara Royuela, Luis Miguel Pinho, Tiago Carvalho, and Eduardo Quiñones. Time-predictable task-to-thread mapping in multi-core processors. *Journal of Systems Architecture*, 148:103068, 2024. doi:10.1016/J.SYSARC.2024.103068.

27 Maria A Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quiñones. Timing characterization of OpenMP4 tasking model. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE, 2015.

28 Maria A Serrano, Sara Royuela, and Eduardo Quiñones. Towards an OpenMP specification for critical real-time systems. In *14th International Workshop on OpenMP*. Springer, 2018.

29 Faisal Shahzad, Jonas Thies, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein. CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):501–514, 2018. doi:10.1109/TPDS.2018.2866794.

30 Aleksandar Simevski, Rolf Kraemer, and Milos Krstic. Investigating core-level N-modular redundancy in multiprocessors. In *IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*, pages 175–180. IEEE, 2014. doi:10.1109/MCSOC.2014.33.

31 John Paul Walters and Vipin Chaudhary. Application-level checkpointing techniques for parallel programs. In *International Conference on Distributed Computing and Internet Technology*, pages 221–234. Springer, 2006. doi:10.1007/11951957_21.

32 Franck Wartel and Certain Antoine. HP4S: High Performance Parallel Payload Processing for Space. In *European Workshop on On-board Data Processing*, 2021. URL: https://zenodo.org/records/5639503/files/05.01_OBDP2021_Certain.pdf?download=1.

33 Michael Wong, Michael Klemm, Alejandro Duran, Tim Mattson, Grant Haab, Bronis R de Supinski, and Andrey Churbanov. Towards an error model for OpenMP. In *International Workshop on OpenMP*, pages 70–82. Springer, 2010. doi:10.1007/978-3-642-13217-9_6.

34 Chenle Yu, Sara Royuela, and Eduardo Quiñones. Taskgraph: A low contention OpenMP tasking framework. *Transactions on Parallel and Distributed Systems*, 34(8):2325–2336, 2023. doi:10.1109/TPDS.2023.3284219.

35 Chenle Yu, Sara Royuela, and Eduardo Quiñones. Enhancing heterogeneous computing through OpenMP and GPU graph. In *53rd International Conference on Parallel Processing*, pages 534–543, 2024. doi:10.1145/3673038.3673050.