Partition-based scheduling on multi-core systems

1st Rubayet Kamal
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
rubayet.kamal@stud.hshl.de

***Abstract*—Systems with components of varying criticality often require stronger verification for high-criticality parts compared to low-criticality ones. In multi-core systems, partitioning reduces the complexity of scheduling by assigning partitions to different cores, allowing isolated execution with optional inter-partition communication. However, partitioned scheduling faces resource utilization challenges similar to the bin-packing problem: tasks may fail to be assigned to any processor even when sufficient total capacity exists. This inefficiency is particularly severe when tasks have high individual utilization, potentially limiting system resource usage to as little as half. Moreover, assigning tasks while maintaining load balance and predictability is an NP-hard problem. This paper explores the fundamentals of partition-based scheduling and examines heuristic allocation and scheduling algorithms. In [1], the authors have used a clustering-based task allocation strategy with Earlest Deadline First (EDF) scheduling algorithm in each core. This strategy or technique is simulated using C and allocation of tasks are visualised using Python script.**

***Index Terms*—multi-core, partition, EDF, styling, insert**

I. Introduction

Real-time systems are often classified based on their timing requirements into soft, firm, and hard real-time systems [2]. Real-time computer system must react to stimuli from its environment within time intervals dictated by its environment. The instant when a result must be produced is called a deadline. If a result has utility even after the deadline has passed, the deadline is classified as soft, otherwise it is firm. If severe consequences could result if a firm deadline is missed, the deadline is called hard [9]. Firm real-time systems are often referred to as mixed-criticality systems due to the presence of some tasks being highly critical than others [16].

The development of mixed-criticality real-time systems presents several challenges, particularly in task scheduling and resource allocation. While power management techniques such as Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) have been explored for energy optimization, these methods were initially designed for uniprocessor systems [2]. With the increasing computational demands and advances in integrated circuit miniaturization, multi-core platforms is a valid solution for providing the necessary computational resources.

Scheduling tasks on multi-core systems introduces two key problems [1]:

- **Allocation problem (spatial organization)** : assigning each task to a core.
- **Scheduling problem (temporal organization)** : determining the execution order of tasks on each core.

According to [19], task scheduling heavily depends on efficient task allocation. A clustering-based scheduling algorithm addresses this by partitioning tasks that frequently communicate onto the same or nearby cores, minimizing communication overhead [1]. As per [16], partitioning isolates different parts of the system to prevent mutual interference. However, partitioned scheduling faces resource utilization issues similar to the bin-packing problem: a task might not fit on any core despite sufficient total system capacity. This inefficiency becomes significant when individual task utilizations are high, and in the worst case, only half of the system's resources may be utilized.

To address this, semi-partitioned scheduling (a hybrid approach) has been proposed. In this method, most tasks are statically assigned to fixed cores, while a small number of tasks are split into subtasks across different cores, enabling limited task migration [1]. Each core must still pass schedulability tests after assignment. Clustering concepts are used to group communicating tasks together, and in [1], the author employed Earliest Deadline First (EDF) algorithm for local scheduling of each core.

EDF is a scheduling algorithm in which jobs with earliest deadlines have higher priority. EDF is an optimal scheduling algorithm on preemp- tive uniprocessors, in the following sense: if a collection of independent jobs can be scheduled (by any algorithm) such that all the jobs complete by their deadlines, EDF will schedule this collection of jobs such that they all complete by their deadlines [2].

Although semi-partitioned scheduling shows theoretical performance improvements, its practical adoption remains limited due to concerns about context switch overhead and migration costs. As discussed in [17], experiments suggest that for task sets with reasonable parameters, semi-partitioned scheduling can indeed outperform traditional partitioned approaches even under realistic overhead conditions.

A clustering-based task allocation strategy that aims to minimize communication costs, balance load across cores, and improve system feasibility for mixed-criticality real-time systems is implemented in this paper, inspired from [1]. Each cores run on Earliest deadline first scheduling algorithm. However, [18] has suggested that EDF can lead to many tasks missing their deadlines on multi-core processors in overload condition for multi-core systems. The purpose of this paper is to evaluate the communication cost between core and ease of allocation of tasks using the clustering-based strategy.

The remainder of this paper is structured as follows: Section II presents the core concepts required for understanding real-time mixed critical systems along with multi-core architectures and multi-core scheduling strategies. Section III defines partition-based scheduling mechanisms, highlighting the differences between static and dynamic partitioning, and discussing the objectives and challenges associated with partitioning. The importance of task partitioning in mixed-criticality systems is also discussed. Section IV outlines the clustering-based scheduling strategy and task allocation methodology, as originally proposed in [1]. Section V describes the implementation of the named algorithm using C code snippets. Thereafter, section **??** shows the outcome from simulation and discusses the key findings. Finally, Section VI concludes the paper and summarizes the main contributions.

II. Background Material

Before exploring the strategy implemented in this paper, it is necessary to discuss the fundamental concepts to ensure that the terms used in later sections are familiar. This section is divided into three subsections: II-A introduces real-time systems and tasks; II-B covers multi-core processor architectures; and II-C explains scheduling approaches on multi-core systems. The objective of this section is to clarify all necessary terminology before implementing and simulating the desired partition-based scheduling algorithm.

A. Tasks and Task Types in Real-Time Systems

A real-time system evolves as a function of physical time [9]. A set of related jobs that execute to support a system function is called a task [11]. Therefore, a real-time task is primarily defined by its temporal properties.

According to [7], a periodic task τ_i is characterized by two parameters:

- **Worst-Case Execution Time (WCET)** W_i: The maximum amount of time a task τ_i could take to execute.
- **Period** T_i: The fixed time interval between successive arrivals of the task.

These properties are assumed to be constant for all task instances in a homogeneous multi-core architecture.

Additional temporal characteristics of real-time tasks, as listed in [1], include:

- **Release Time or Arrival Time** (R_i): The time at which a task τ_i becomes eligible to start execution.
- **Deadline** (D_i): The latest time by which the task must complete its execution.
- **Latency** (L_i): The remaining time before a task either starts execution or its deadline occurs.
- **Laxity or Slack Time** (X_i): The maximum time a task can be delayed after activation and still meet its deadline.

Figure 1 illustrates the temporal properties of a real-time task.

Real-time tasks can be classified as either *periodic* or *aperiodic*:

- **Periodic Tasks**: These tasks consist of an infinite sequence of identical activities, called instances or jobs, that are activated regularly at a constant rate [4]. Periodic tasks are characterized by strict deadlines and form the majority of tasks in a real-time application. According to [12], a periodic task τ_i is typically modeled by four parameters: (R_i, W_i, D_i, P_i) [1].
- **Aperiodic Tasks**: These tasks also consist of a sequence of identical jobs, but their activations are unpredictable and not evenly spaced. Aperiodic tasks are generally modeled by a single parameter, W_i, representing their worst-case execution time [1].

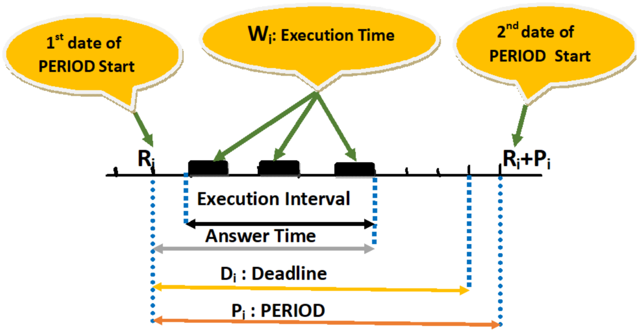

Fig. 1. Temporal properties of a task

A special class of aperiodic tasks is *sporadic tasks* [4]. Sporadic tasks, while aperiodic in execution time, have their execution rate constrained by a minimum inter-arrival time. They are modeled by the parameters $(W_i, D_i, T_{\min})$, where $T_{\min}$ is the minimum time interval between two consecutive activations, and D_i is the critical delay [1].

B. Architecture of Multi-Core Processors

The performance of single-threaded processors has become increasingly limited by power constraints, prompting processor architects to adopt multi-core designs as a means to enhance performance. As Intel analyst Nathan stated, ”multi-core processors are the inevitable product of Moore’s Law.” The growing demands of modern applications and the continued advancement of CPU manufacturing technologies—enabling the integration of a large number of transistors on a single chip—have driven the rapid development of multi-core processors [13].

Although multi-core technology is a hardware innovation, it requires corresponding software support to fully realize its potential. In essence, a multi-core processor integrates multiple processing cores on a single chip, allowing the system to leverage the abundance of transistor resources. This enables the exploitation of various levels of parallelism—such as instruction-level and thread-level parallelism—through concurrent core execution, thereby enhancing overall performance [13].

Increasing the number of cores in a processor not only improves parallel task execution but also enhances energy efficiency. Multi-core systems can deliver higher performance by using several low-frequency cores instead of a single high-frequency core, offering both flexibility and power savings.

Compared to single-core processors, multi-core processors offer several key advantages:

- Improved parallel performance [8],
- Lower communication latency [10],
- Higher bandwidth [15],
- Reduced power consumption.

Multi-core processors can be categorized as either *homogeneous* or *heterogeneous*, depending on how the cores are integrated:

- **Homogeneous Multi-Core Processors**: These integrate multiple identical cores on a single chip. Each core typically performs similar tasks, enabling symmetric and balanced parallel computing.
- **Heterogeneous Multi-Core Processors**: These integrate a mix of cores with varying complexity. Typically, a complex superscalar core (main core) handles general-purpose computation, such as operating system tasks and scheduling, while simpler subordinate cores are optimized for specific tasks. These lightweight cores are often managed by the main core and accelerate computations for specialized applications [13].

C. The scheduling concept

When a single processor has to execute a set of concurrent tasks – that is, tasks that can overlap in time – the CPU has to be assigned to the various tasks according to a predefined criterion, called a scheduling policy. The set of rules that, at any time, determines the order in which tasks are executed is called a scheduling algorithm [4].

In many operating systems that allow dynamic task activation, the running task can be interrupted at any point, so that a more important task that arrives in the system can immediately gain the processor and does not need to wait in the ready queue. In this case, the running task is interrupted and inserted in the ready queue, while the CPU is assigned to the most important ready task that just arrived. The operation of suspending the running task and inserting it into the ready queue is called preemption as shown in fig. 2 [4]. Preemptive scheduling typically allows higher efficiency, in the sense that it al- lows executing a real-time task sets with higher processor utilization [4].

”Among the great variety of algorithms proposed for scheduling real-time tasks, the following main classes can be identified:

- Preemptive: In preemptive algorithms, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy.
- Non-preemtptive: In non-preemptive algorithms, a task, once started, is executed by the pro- cessor until completion. In this case, all scheduling decisions are taken as the task terminates its execution.
- Static: Static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.
- Dynamic: Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system evolution.
- Off-line: A scheduling algorithm is used off line if it is executed on the entire task set before tasks activation. The schedule generated in this way is stored in a table and later executed by a dispatcher.
- Online: A scheduling algorithm is used online if scheduling decisions are taken at runtime every time a new task enters the system or when a running task terminates.
- Optimal: An algorithm is said to be optimal if it minimizes some given cost function defined over the task set. When no cost function is defined and the only concern is to achieve a feasible schedule, then an algorithm is said to be optimal if it is able to find a feasible schedule, if one exists.
- Heuristic: An algorithm is said to be heuristic if it is guided by a heuristic function in taking its scheduling decisions. A heuristic algorithm tends toward the optimal schedule, but does not guarantee finding it” [4].

Multi-core scheduling is a 2 dimensional problem. First, allocation problem (spatial organization) deter- mines for each tasks which core should run on. Second, scheduling problem (temporal organization) de- fines date and order execution of tasks [1].

The complexity of scheduling algorithms is of high relevance in dynamic real-time systems, where scheduling decisions must be taken online during task execution. A polynomial algorithm is one whose time complexity grows as a polynomial function p of the input length n of an instance. The complexity of such algorithms is denoted by $\mathcal{O}(p(n))$. Each algorithm whose complexity function cannot be bounded in that way is called an exponential time algorithm.

In particular, NP is the class of all decision problems that can be solved in polynomial time by a nondeterministic Turing machine. According to [4]:

> ”A problem Q is said to be NP-complete if $Q \in \mathsf{NP}$ and, for every $Q' \in \mathsf{NP}$, Q' is polynomially transformable to Q. A decision problem Q is said to be NP-hard if all problems in NP are polynomially transformable to Q, but we cannot show that $Q \in \mathsf{NP}$.”

An algorithm is said to be heuristic if it is guided by a heuristic function in taking its scheduling decisions. A heuristic algorithm tends toward the optimal schedule, but does not guarantee finding it. For task scheduling, heuristic-based scheduling algorithms are common. These are usually classified into three classes:

- ”Priority-based scheduling: priorities are calculated and assigned to the tasks which are then scheduled on the processors according to their priorities.
- Duplication-based scheduling: while tasks are allocated to a processor, its parent (and predecessor) tasks are duplicated to occupy the idle times of the processor to eliminate the communication delay that occurs when

message is passed from the parent tasks to the allotted task.

- In cluster-based scheduling, some tasks, that need to communicate among themselves, are grouped together to form a cluster. Clusters are then scheduled on to an available processor. The main problem arises when the number of clusters is greater than the number of processors. This leads to programming the communicating clusters on the same processor and which remains in the nearest processor" [1].

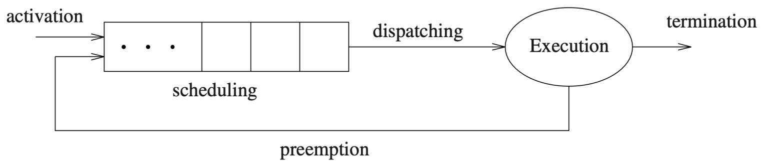

Fig. 2. preemption

III. Partition-based Scheduling

Various algorithms for scheduling recurrent real-time task systems on multiprocessor platforms impose different constraints on where a task's jobs may execute. Two fundamental approaches widely studied for scheduling on multi-core processors are *global scheduling* and *partition-based scheduling*:

- **Global Scheduling:** In global scheduling, a task may execute on any processor core and is allowed to migrate between cores. A single global ready queue holds all ready jobs, and at each scheduling decision point, the highest-priority job is selected for execution. Essentially, a unified scheduling policy is applied across all cores. This concept is illustrated in Fig. 3.
- **Partition-based Scheduling:** In partitioned scheduling, tasks are statically assigned to specific cores. Each processor core schedules only the tasks assigned to it using a local uniprocessor scheduling algorithm. Hence, every task is strictly bound to a single core, and all its jobs must execute on that core. The goal is to assign each task either individually or as part of a subset to a particular core. This enables the application of uniprocessor scheduling techniques independently on each core, each maintaining its own task queue [1]. This approach is shown in Fig. **??**.

Global scheduling allows optimal task distribution due to migration flexibility. However, implementing efficient and predictable global schedulers for real-time systems has proven challenging. Notably, unlike partitioned scheduling—where the synchronous task arrival pattern represents the worst-case scenario—global scheduling lacks any known finite set of worst-case job arrival sequences. This significantly complicates exact schedulability analysis. As demonstrated in [14], there exists no optimal online algorithm for global scheduling of constrained-deadline sporadic task systems on two or more processors.

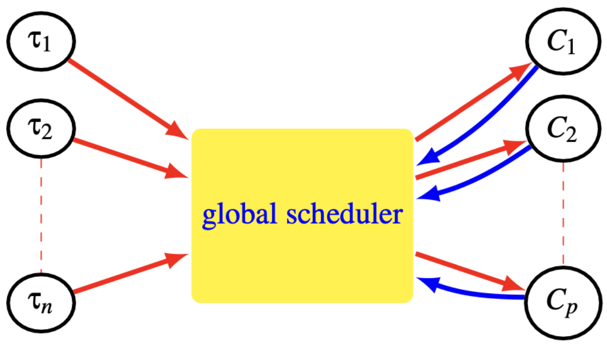

Fig. 3. Global scheduling [1]

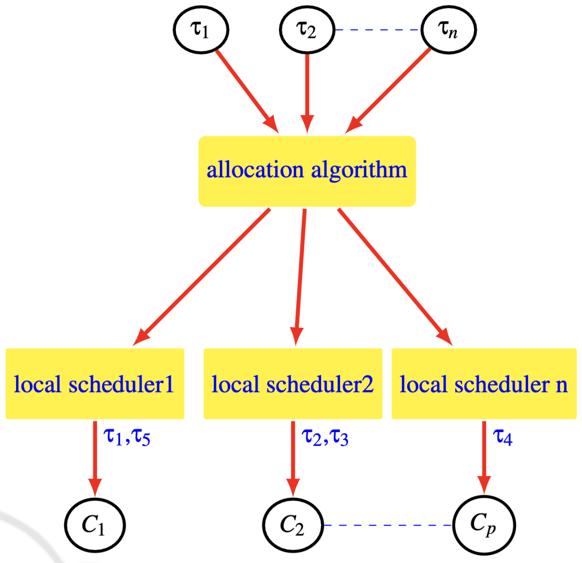

Fig. 4. Partition-based scheduling [1]

The task allocation problem in partition-based scheduling is equivalent to the well-known *bin-packing* problem, where one must assign n items of varying sizes into m identical bins, a problem known to be NP-hard.

Both global and partitioned scheduling are special cases of *clustered scheduling*, as discussed in Section II-C. In clustered scheduling, the processor cores are grouped into clusters, and each recurrent task is statically mapped to one cluster. Migration of a task's jobs is only permitted within the assigned cluster [14].

Recent research suggests that the partitioned approach outperforms others when it comes to scheduling hard real-time systems—both theoretically and practically. However, it is not without drawbacks. Partitioning can lead to significant resource underutilization: even when the system has enough total capacity, a task might not fit on any individual core. This issue is exacerbated when task utilization is high, potentially restricting system utilization to as little as 50% in the worst case.

To mitigate this, a *hybrid* or *semi-partitioned* scheduling approach has been proposed. Here, tasks are first allocated via a partitioning heuristic, but selective task migrations are allowed across cores. Cores may exchange some tasks with others—a mechanism referred to as task migration. This enhances flexibility and helps better utilize processor capacity. A semi-partitioned scheduling algorithm consists of two parts: the partitioning algorithm, which determines how to split and assign each task (or rather each part of it) to a fixed processor, and the scheduling algorithm, which determines how to schedule the tasks assigned to each processor [17].

It is important to note that for global, partitioned, and hybrid approaches, certain schedulability conditions must be satisfied. These will be discussed in Section IV.

In [1], a development strategy based on the hybrid approach is proposed, as it reduces communication overhead and energy consumption. Consequently, this results in a feasible and efficient system with improved quality of service. This clustering-based scheduling strategy is further elaborated in Section IV.

IV. Clustering-Based Scheduling Strategy

In [1], it is assumed that each core schedules tasks locally using the EDF (Earliest Deadline First) algorithm. This section presents the clustering-based task allocation strategy implemented in [1]. It is divided into three subsections: the utilization factor in IV-A, the communication cost between cores due to their distances in IV-B, and the proposed scheduling strategy in IV-C.

A. Utilization Factor

According to Liu and Layland [5], the processor utilization rate by a task τ_i is given by:

$$\upsilon_{\tau_i} = \frac{w_i}{T_i} \tag{1}$$

Here, w_i represents the worst-case execution time of task τ_i, and T_i is its period. The utilization factor for N tasks assigned to core C_k is the sum of the individual utilizations:

$$U_{C_k} = \sum_{i=1}^{N} \upsilon_{\tau_i} = \sum_{i=1}^{N} \frac{w_i}{T_i}, \quad \forall k \in [1 \dots P] \tag{2}$$

A necessary condition for schedulability of tasks on a core is that the total utilization does not exceed 1:

$$U_{C_k} \leq 1 \tag{3}$$

In [1], each core C_k $(k \in [1 \dots P])$ schedules its assigned tasks locally using EDF. Each core must satisfy the schedulability condition defined in Equation 3.

B. Communication Cost

Communication between cores is facilitated by a communication medium such as a bus or a Network-on-Chip (NoC). In NoC architectures, the communication cost between a pair of cores C_k and C_l depends on their relative distance and is denoted by Cost_{C_k,C_l}. Let X_{C_k,C_l} be the cost of transferring one byte of data from core C_k to core C_l. The cost matrix is defined as:

$$\text{Cost}_{C_k,C_l} = \begin{cases} X_{C_k,C_l}, & \text{if } k \neq l, \quad \forall k,l \in [1 \dots P] \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

The total communication cost is obtained by multiplying Cost_{C_k,C_l} with the volume of exchanged data, denoted by $SM_{i,j}$, which represents the size of the message exchanged between tasks τ_i and τ_j in bytes:

$$\text{TotalCost} = \sum_{k=1}^{P} \sum_{l=1}^{P} \sum_{i=1}^{N} \sum_{j=1}^{N} \text{Cost}_{C_k,C_l} \cdot SM_{i,j} \tag{5}$$

C. Proposed Strategy

The clustering-based allocation strategy builds on the concept introduced in Section II-C. The key idea is to group communicating tasks into clusters (CL) and assign them to the same or neighboring cores to reduce communication overhead. Each core schedules its assigned tasks locally using EDF and must satisfy the schedulability condition from Equation 3.

Tasks that frequently communicate are grouped into clusters and allocated to the least loaded core to balance the load and improve system performance. This ensures that the load on each core does not exceed its maximum capacity, maintaining system reliability and real-time performance.

In real-time systems, task dependencies and inter-task communication are critical. When tasks are distributed across different cores, minimizing communication cost becomes essential to maintaining system performance.

The objective of the strategy is to allocate tasks to cores such that total communication cost is minimized while satisfying the utilization constraint. In [1], the proposed strategy is compared with the approaches in [3] and [6]. The system uses a 1-D Mesh network as the communication topology, where cores are connected in a linear array. Each core communicates only with its immediate neighbors.

For example, if Task A resides on Core 0 and needs to communicate with Task B on Core 3, the message must traverse through Core 1 and Core 2—resulting in three hops. To reduce such communication delays, the strategy allocates frequently communicating tasks to the same or nearby cores.

Clusters of tasks are mapped to specific cores that are close to each other in the mesh network. Section V will detail the implementation of this strategy in C and present simulation results obtained using UPPAAL.

V. Implementation of Clustering-based algorithm in C

The complete working code of the implementation can be found in the following GitHub repository: github.com/RubayetKamal/SS2025_Embedded_Engineering. This section is divided into 5 subsections where subsection V-A introduces the algorithm used to implement clustering for task allocation used by [1]. Subsection V-B shows the tasks that are used here as an example and the communication matrix to highlight which task is communicating with which one so that the cluster can be formed. Subsection V-C, not only desribes step-by-step on how communication cost between cores are calculated, but also shows code snippet to present the actual implementation. Subsection V-D lists the terminal-based output that is received when running the code. Finally, V-E shows a UPPAAL simulation of the

A. Clustering

Algorithm 1 Assign Clusters to Cores

```
1: for each cluster in clusters do
2:     Find the least-loaded core that can still take the cluster's
       total utilization
3:     Assign all tasks in the cluster to that core
4:     Update the core's utilization
5: end for
```

B. Task Initilization and Communication Volume

7 tasks are initilized with their core properties, as already discussed: period and worst case execution time. As already states, utilization of core cannot exceed one for a new task to be allocated as shown in equation 3. Table I show the tasks with their corresponding properties.

TABLE I
TASK PARAMETERS

Task No.	Execution Time (C)	Period (T)
0	1	5
1	2	10
2	1	4
3	2	8
4	3	4
5	1	10
6	5	6

Table II shows the communication volume between tasks. It basically shows how much in bytes are being transferred as message from one task to another. The clustering logic checks if a task can be placed on a core that already has a communicating task and whether the resulting utilization stays under the maximum utilization threshold of 1.

The points to notice are the following: Task 0,1,2 and 3 are in constant communication with each other. Therefore if the equation 3 allows, these tasks can be allocated in one core. In this case, the total utilization of these 4 tasks is equal to 0.9 leading to their allocation in Core 0. Task 5 on the other hand only is in communication with Task 3. Therefore the utilization of this task has to comply wth equation 3 for the core. As Task 5 only has 0.1 as utilization, the task was able to be allocated in Core 0 forming the cluster. Task 4 and 6 only communicate among each other. This leads to these two tasks forming a cluster. However, once Task 4 is allocated in Core 1, the utilization constrain does not allow Task 6 to be placed in Core 1. This leads to Task 6 being placed in the neighbouring core of Core 2. The allocation of Task 2 in a core near to Core 1 reduces the communication cost between Core 1 and Core 2. This is shown here in the terminal output:

TABLE II
TASK DEPENDENCY MATRIX

	T0	T1	T2	T3	T4	T5	T6
T0	0	5	3	9	0	0	0
T1	5	0	4	7	0	0	0
T2	3	4	0	0	0	0	0
T3	9	7	0	0	0	8	0
T4	0	0	0	0	0	0	7
T5	0	0	0	8	0	0	0
T6	0	0	0	0	7	0	0

C. Cost calculation

Equations 4 and 5 dicussed in subsection IV are used to calculate the total communication cost between cores.

The function `Calculating_TotalCommunicationCost()` is responsible for computing the total communication cost incurred by the task-to-core mapping in a multi-core system. It begins by initializing a variable `totalCost` to zero. This variable accumulates the overall communication cost resulting from interactions between tasks assigned to different cores.

The function iterates over every possible pair of cores in the system using two nested loops. Let us denote the outer loop index as `firstCore` and the inner loop index as `secondCore`. For each core pair, it proceeds to examine all pairs of tasks, where one task belongs to `firstCore` and the other to `secondCore`. This is achieved through another two nested loops that traverse the list of assigned tasks for each core.

For each task pair, the function identifies two tasks: `firstTask` assigned to `firstCore` and `secondTask` assigned to `secondCore`. It then calculates the communication cost between these tasks by multiplying two quantities:

- `communicationVolume[firstTask][secondTask]`, which represents the volume of data or messages exchanged between the tasks.
- `costMatrix[firstCore][secondCore]`, which captures the communication cost between the two cores themselves (often modeled as distance or latency).

The product of these two terms gives the cost of communication between the selected pair of tasks, considering both the data exchanged and the penalty associated with inter-core communication. This cost is added to the running total `totalCost`.

The function continues this process for all combinations of core pairs and their respective assigned tasks. After all relevant combinations have been evaluated, the function returns `totalCost`, which reflects the complete communication burden introduced by the current allocation of tasks to cores. This metric is critical for assessing the efficiency of clustering algorithms in minimizing communication overhead.

List 1 shows a code snippet of the implementation of calculating total communication cost.

```c
float Calculating_TotalCommunicationCost() {
    float totalCost = 0.0;

    for (int firstCore = 0; firstCore < numberOfCores; firstCore++) {
        for (int secondCore = 0; secondCore < numberOfCores; secondCore++) {

            for (int i = 0; i < cores[firstCore].num_tasks; i++) {
                for (int j = 0; j < cores[secondCore].num_tasks; j++) {

                    int firstTask = cores[firstCore].assigned_tasks[i];
                    int secondTask = cores[secondCore].assigned_tasks[j];

    totalCost += costMatrix[firstCore][secondCore] * communicationVolume[firstTask][secondTask];
                }
            }
        }
    }

    return totalCost;
}
```

Listing 1. Function to calculate total communication cost

D. Output

And finally, the output from the terminal is as follows, as already discussed from the implementation of total communication cost along with the clustering algorithm described in algorithm 1.

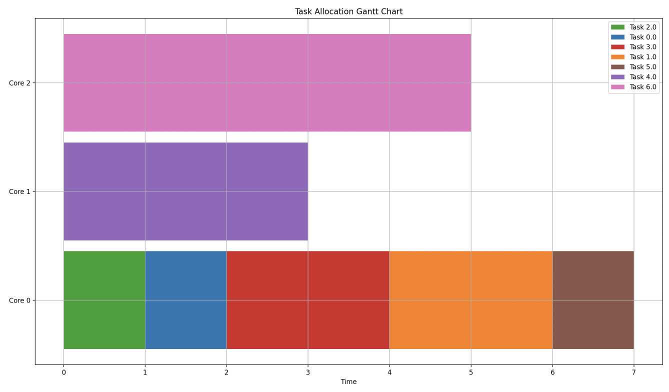

Fig. 5. Allocation of Tasks on cores

E. Simulation

Fig. 6 shows the sequence of how the scheduler and the allocation algorithms are synchronized. EDF is not used in this simulation die to time limitation and level of complication of the system. Only the first core is simulated which waits for cores to be allocated once they pass the utilization test. After that, it is up to the scheduler to execute the task in a particular order.

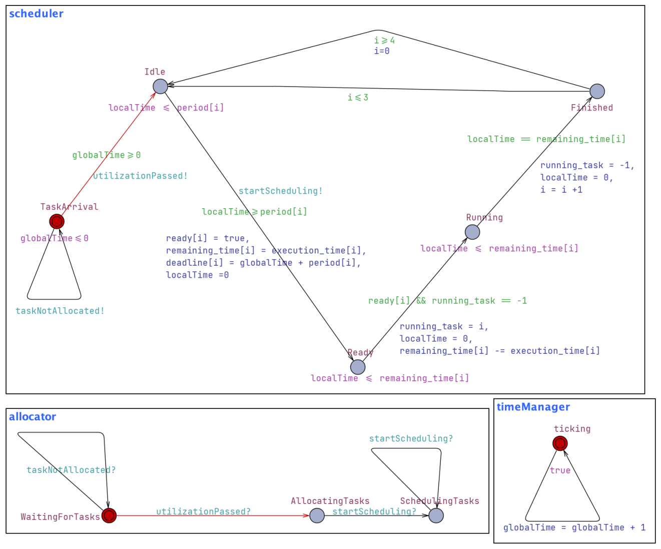

Fig. 6. UPPAAL

VI. Conclusion

In this paper, partition-based scheduling of multi-core real-time systems is explored. The problem of periodic tasks allocation on a homogeneous multi-core architecture using tasks clustering is discussed. The clustering-based scheduling algorithm, inspired from [1] is implemented in C and later simulated in UPPAAL. The output of the proposed solution shows tasks with frequent communication among each other form a cluster and they are allocated on cores close to each other to reduce communication cost, provided that all cores pass the schedulability test.

References

[1] Hayfa Ben Abdallah, Hamza Gharsellaoui, and Sadok Bouamama. “A Novel Partitioning Approach for Real-Time Scheduling of Mixed-Criticality Systems”. In: *Proceedings of the 16th International Conference on Agents and Artificial Intelligence, ICAART 2024, Volume 3, Rome, Italy, February 24-26, 2024*. Ed. by Ana Paula Rocha 0001, Luc Steels, and H. Jaap van den Herik. SCITEPRESS, 2024, pp. 882–889. ISBN: 978-989-758-680-4. DOI: 10.5220/0012411200003636. URL: https://doi.org/10.5220/0012411200003636.

[2] Nadine Abdallah et al. “Partitioned EDF scheduling in multicore systems with quality of service constraints”. In: *2011 18th IEEE International Conference on Electronics, Circuits, and Systems*. 2011, pp. 764–767. DOI: 10.1109/ICECS.2011.6122386.

[3] Poornima Bhardwaj and Vinod Kumar. “An Effective Load Balancing Task Allocation Algorithm using Task Clustering”. In: *International Journal of Computer Applications* 77 (Sept. 2013), pp. 32–39. DOI: 10.5120/13410-1064.

[4] C Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011.

[5] Nathan Fisher and SanjoyK. Baruah. “Rate-Monotonic Scheduling”. In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. New York, NY: Springer New York, 2016, pp. 1784–1788. ISBN: 978-1-4939-2864-4. DOI: 10.1007/978-1-4939-2864-4_334. URL: https://doi.org/10.1007/978-1-4939-2864-4_334.

[6] Dr. Kapil Govil. “A Smart Algorithm for Dynamic Task Allocation for Distributed Processing Environment”. In: *International Journal of Computer Applications* 28 (Aug. 2011). DOI: 10.5120/3362-4641.

[7] Nan Guan et al. “Fixed-Priority Multiprocessor Scheduling with Liu and Layland’s Utilization Bound”. In: *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. 2010, pp. 165–174. DOI: 10.1109/RTAS.2010.39.

[8] L. Hammond et al. “Stanford Hydra CMP”. In: *Micro, IEEE* 20 (Apr. 2000), pp. 71 –84. DOI: 10.1109/40.848474.

[9] H. Kopetz. *Real-time systems: Design principles for distributed embedded applications*. Springer, 2011.

[10] Xiuzhen Lian et al. “Cache Coherence Protocols in Shared-Memory Multiprocessors”. In: Jan. 2015. DOI: 10.2991/iccse-15.2015.52.

[11] Jane W.S Liu. *Real-time Systems*. Pearson, 2006.

[12] M.Alabau and T. Dechaize. *Temps-réel par échéance*. Ordonnancement, 1991.

[13] Yang Pan et al. “The research of multi-core parallel technology”. In: *2012 8th International Conference on Natural Computation*. 2012, pp. 1056–1059. DOI: 10.1109/ICNC.2012.6234619.

[14] Giorgio Buttazzo Sanjoy Baruah Marko Bertogna. *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

[15] M.B. Taylor et al. “The Raw microprocessor: a computational fabric for software circuits and general-purpose programs”. In: *IEEE Micro* 22.2 (2002), pp. 25–35. DOI: 10.1109/MM.2002.997877.

[16] Juan Zamorano and Juan Antonio de la Puente. “On real-time partitioned multicore systems”. In: *ACM SIGAda Ada Letters* 33 (Nov. 2013), pp. 33–39. DOI: 10.1145/2552999.2553003.

[17] Yi Zhang et al. “Implementation and empirical comparison of partitioning-based multi-core scheduling”. In: *2011 6th IEEE International Symposium on Industrial and Embedded Systems*. 2011, pp. 248–255. DOI: 10.1109/SIES.2011.5953668.

[18] Benhai Zhou, Jianzhong Qiao, and Shukuan Lin. “Research on synthesis parameter real-time scheduling algorithm on multi-core architecture”. In: *2009 Chinese Control and Decision Conference*. 2009, pp. 5116–5120. DOI: 10.1109/CCDC.2009.5194980.

[19] Kai Zhu and Yun Ding. “Research on Low Power Scheduling of Heterogeneous Multi Core Mission Based on Genetic Algorithm”. In: *2017 9th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*. 2017, pp. 219–223. DOI: 10.1109/ICMTMA.2017.0059.