QUBX: RUST LIBRARY FOR QUEUE-BASED MULTITHREADED REAL-TIME PARALLEL AUDIO STREAMS PROCESSING AND MANAGEMENT

Pasquale Mainolfi
Conservatorio Statale di Musica "G. Martucci"
Salerno, IT
pasquale.mainolfi@consalerno.it

ABSTRACT

The concurrent management of real-time audio streams pose an increasingly complex technical challenge within the realm of the digital audio signals processing, necessitating efficient and intuitive solutions. Qubx endeavors to lead in tackling this obstacle with an architecture grounded in dynamic circular queues, tailored to optimize and synchronize the processing of parallel audio streams. It is a library written in Rust, a modern and powerful ecosystem with a still limited range of tools for digital signal processing and management. Additionally, Rust's inherent security features and expressive type system bolster the resilience and stability of the proposed tool.

1. INTRODUCTION

Effective management of parallel audio streams is a critical aspect within the domain of real-time digital signal processing (DSP). Managing concurrent audio streams involves processing them simultaneously and independently from one another. This necessitates careful consideration of various aspects of the problem, including latency, computational load, temporal coherence, and resource management, all of which directly impact the quality, performance, and robustness of the entire system.

The management and control of these aspects can significantly vary depending on the programming language utilized. Implementing a particular approach in a specific language entail providing a tangible illustration of how that theoretical approach translates into action, with all the limitations and advantages inherited from the implementation context.

Although specialized environments exist, such as C-family languages like JUCE[1], Cmajor[2] or Faust[3], special attention has been given to Rust[4] in the choice of programming language. Rust is emerging as an increasingly popular choice in the field of DSP, thanks to its combination of high performance and system security.

In the field of Digital Signal Processing (DSP), the programming languages C and C++ have long been considered the reference point, owing to their ability to operate at hardware-proximate abstraction levels. However, the memory-unsafe management in these languages is well known and such vulnerability can be introduced into applications without developers' awareness [18].

In this context, Rust emerges as a memory-safe language, ensuring secure memory handling while maintaining high execution performance [19, 20]. Its distinctive ownership and borrowing system prevent common bugs such as null pointer dereferencing or invalid memory release. This characteristic significantly reduces the possibility of unexpected behaviors and enhances the stability and reliability of applications.

Examples of this are studies conducted by Rooney and Matthews [21], comparing the efficiency of FFT algorithm implementations in Rust and C on two different Raspberry Pi devices, demonstrating the superiority of Rust implementation. Fougeray [22] examined the efficiency of DSP algorithm implementations in Rust compared to CMSIS-DSP, noting a performance increase of approximately 1.8x. Additionally, Namavari [23] leveraged Rust as a foundation to create a Domain Specific Language (DSL) for controlling and managing specific musical structures, capitalizing on Rust's type system.

Furthermore, Cargo[5], the package manager for Rust, is intuitive to use and designed to simplify the development and distribution process. Thanks to its integration with Rust's build system, Cargo automatically manages dependencies and package versions, ensuring consistency and compatibility among various libraries.

Therefore, the primary goal is to actively contribute to the development of tools for managing and controlling digital audio signals in a new language that offers a programming paradigm and unique features for developing highly performant and scalable solutions. Nevertheless, creating an environment for real-time parallel data stream DSP requires a deep understanding of signal processing concepts and the peculiarities of the involved hardware. This complexity is further accentuated by the need to manage data in real-time, ensuring synchronization and coherence of operations, and requires precise management of latency and execution priorities. Additionally, challenges related to low-level programming and performance optimization may arise, as such contexts often demand the implementation of efficient algorithms operating on large amounts of real-time data. This can make the development and debugging process extremely challenging. Consequently, there is a need to devise a tool whose primary objective is to drastically simplify the implementation of complex contexts, providing an intuitive interface that minimizes the time required to configure and use the system. This would allow users to focus directly on their goals without encountering obstacles or wasting time associated with implementation.

In this context, the Qubx library emerges as an intuitive and flexible tool, with an architecture based on shared queue lists

[1] https://juce.com/
[2] https://cmajor.dev/
[3] https://faust.grame.fr/
[4] https://www.rust-lang.org/
[5] https://doc.rust-lang.org/cargo/

aimed at creating and managing advanced and high-performance audio processing systems.

2. APPROACH ARCHITECTURE

Qubx enables efficient management of concurrent audio streams through an architecture that leverages queue lists. This design facilitates secure and sequential access to audio data. A key feature of Qubx is its use of meticulously crafted structural logic to mitigate the complexities inherent in multi-threading, such as synchronization challenges and recurrence conflicts. Consequently, Qubx promotes a more deterministic execution flow, thereby enhancing the predictability and reliability of the data management process.

2.1. The Queue as a Key Component

In the domain of audio processing, the queue emerges as a pivotal tool for efficiently handling and manipulating incoming data. The sequential storage of elements provides quick and orderly access to the data in the order of their insertion. This attribute proves especially advantageous for processing tasks that require sequential frame-by-frame processing, facilitating a streamlined and optimized workflow.

2.1.1. Queues (Abstract Data Type)

In computer science, a queue [9] stands as an abstract data type (ADT) that delineates an ordered collection of elements. Operations involving access and removal adhere to the FIFO (First-In-First-Out) principle [1], signifying that the element inserted earliest is the first to be removed [1, 2].

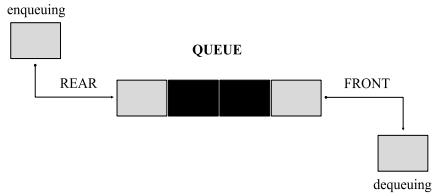

Figure 1: *Main operations on the queue (enqueuing and dequeuing)*

Queues can be categorized as linear or circular, bounded or unbounded. The key distinction between a linear and circular queue lies in managing vacant spaces within the data structure.

A linear queue adheres to a linear structure while maintaining the FIFO principle. In a bounded linear queue, a limit is imposed on the maximum number of elements it can contain. Upon reaching full capacity, attempting to insert a new element triggers an overflow condition [4, 5].

Conversely, a circular queue employs a circular buffer for element storage. In a bounded circular queue, reaching maximum capacity involves relocating the insertion pointer to the buffer's inception, potentially overwriting previous elements if the queue is full. In the unbounded variant, memory allocation dynamically adjusts as needed. Nevertheless, overflow conditions can arise in both scenarios. Similarly, both bounded and unbounded cases may encounter underflow when attempting to remove an element from an empty queue [6].

Key operations supported by a queue include inserting elements (enqueue), removing elements from the queue's outset (dequeue), peeking at the initial element without removing it (peek), assessing whether the queue is empty (empty), and determining the queue's size (size) [3, 6].

The ADT concept an abstract depiction of data organization and permissible operations, abstaining from specifying internal implementations. In practical application, queue implementation utilizes concrete data types such as arrays or linked lists. Although the complexity term may vary slightly depending on the specific implementation and the number of the elements present, in general, the main operations tend to have a costant *O(1)* [6] or linear time complexity relative to the number of elements, while the space complexity is usually linear *O(n)* in the worst case, where *n* represents the number of elements in the queue.

2.1.2. Qubx Queue Model

The queue model employed in Qubx for managing audio frames adopts the unbounded linear type, realized through arrays, structured as depicted in the following pseudocode.

```
CLASS Queue
  VARIABLE q = ARRAY[]
  VARIABLE front = 0
  VARIABLE rear = -1

  FUNCTION empty
    RETURN TRUE IF front > rear

  FUNCTION enqueue(data)
    INCREMENT rear by 1
    q[rear] = data

  FUNCTION dequeue
    IF NOT empty
      ELEMENT = q[front]
      INCREMENT front by 1
      RETURN ELEMENT
    ELSE
      RETURN NONE

  FUNCTION peek
    IF NOT empty RETURN q[front]
```

Utilizing an unbounded queue proves particularly advantageous in this context as it can dynamically adapt to fluctuations in data volume, maintaining constant space allocation without risking data loss. Opting for an array rather than a linked list [9] yields minimal differences apart from simplifying code development and maintenance, thereby mitigating the risk of errors. However, it is imperative to note that unlike arrays, linked list necessitate additional memory for pointers (memory overhead), resulting in heightened memory resource consumption compared to arrays. Moreover, arrays offer superior cache performance since elements occupy contiguous memory locations. Conversely, elements within a linked list may be dispersed non-contiguously in memory, heightening the probability of cache misses and compromising data access performance [7, 8].

2.2. Benefits of Three-Way Multithreading Architecture

In a computer science, a thread denotes an autonomous execution path within a process. A process embodies an instance of a

[6] The Big-*O* notation in computer science is a way to describe the asymptotic behavior of a function, especially in terms of the time complexity or space used by an algorithm. It is useful for understanding how efficient an algorithm is.

running program and has the capability to accommodate multiple threads (multithreading), each executing instructions concurrently while sharing the same execution context, including memory and system resources [9].

Typically, processes execute sequentially, one after another, thereby potentially blocking hardware for operations of a certain duration. Consequently, if another process requires execution, it must await the readiness of the hardware. In the context of multithreading (Figure 2), multiple threads operate relatively simultaneously [11,12].

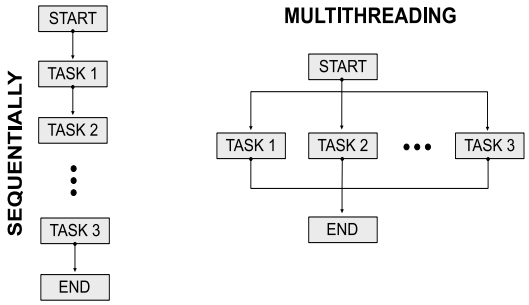

Figure 2: *Sequentially and multithreading process*

In many instances [13, 14, 15], the parallel management of audio streams entails the utilization of multiple threads, each entrusted with processing its designated data stream autonomously. During execution, each thread retrieves data from its stream, conducts necessary processing, and subsequently transmits the processed data to the output. This iterative process continues as long as data persists within the input streams. However, employing a designated number of threads inevitably engenders challenges in resource management and precise synchronization, typically addressed through the implementation of mutexes. A mutex serves as a synchronization mechanism, akin to a semaphore, enabling a thread to temporarily restrict access to a shared resource until it completes its processing.

Starting from version 0.42.0, for instance, the `pd~` object [24] enables the integration of PureData[7] patches executed on separate threads within a dedicated main process (master patch) for I/O management. DSP computations in individual child processes occur independently from those executed in the parent process, while communication between the various processes and the main one occurs through shared FIFO queues.

Faust [25] utilizes OpenMP[8] or WSS (Work Stealing Scheduler, from version 0.9.10 onwards) to automate parallelization. In the former case (OpenMP), it employs a *fork-join* model where each main thread distributes the workload to a pool of child processes. In the letter case the approach is dynamic. A series of threads (created and destroyed dynamically during program execution) actively seeks tasks to execute. Specifically, each thread has its own task queue to execute. When the queue is empty and there are no tasks to execute locally, the thread can "steal" tasks from another thread's queue.

Other parallelization techniques used in Open Sound World (OSW) are illustrated in [14].

While this mechanism safeguards data integrity and forestalls conflicts among threads vying for access to the same shared resource, it also introduces complexities into real-time programming for DSP, rendering it arduous and less accessible.

Nevertheless, alternative implementations featuring non-blocking structures [18] and other proposed solutions for this issue exist [16, 17].

Recently, the *Crill* library (Cross Platform I/O and Low Latency) [28] for C++, developed by Doumler [17], proposes a *lock and wait-free* approach by adapting the Read-Copy-Update (RCU) mechanism [26, 27] to the context of real-time audio.

However, these approaches may be less straightforward and intricate to implement. Therefore, despite the availability of solutions, substantial investments of time and effort are requisite for their effective and intuitive utilization.

The architecture of Qubx is fashioned as a system compartmentalized into three principal streams: one exclusively dedicated to outgoing audio data, another for signal processing and individual signal encoding, and finally, a third for monitoring active processes. This tripartite approach, distinct from conventional multithreading, yields significant technical advantages. Operational components organized in this manner maintain continual interaction, affording precise control over each phase of the process and heightened flexibility in adapting to specific requisites. Additionally, this approach adeptly manages real-time processes, judiciously leveraging available resources, minimizing complexity, and mitigating issues and risks associated with race conditions and deadlocks. It notably diminishes potential read-write conflicts and simplifies issue localization and resolution, as each stream can be independently monitored and analyzed, thereby streamlining the debugging and maintenance processes for the system as a whole.

Each process dedicated to real-time data processing (*QubxDspProcess* in *Qubx*, see sec. 3) has also been designed to maximize efficiency under conditions of excessive computational load through data parallelization. This strategy allows each process to divide the data into manageable portions and perform operations on them simultaneously, thereby ensuring optimal computational load balancing (6)(7). However, it should be noted that the implementation of data parallelism has been designed to be turned off. This is because, when the computational load is light, the additional overhead introduced by this strategy does not improve performance but rather makes the overall process slower (see Table 2).

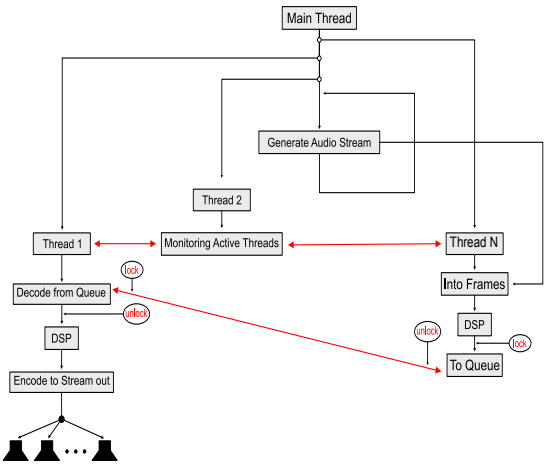

Figure 3: *Operative diagram of Qubx model*

[7] https://puredata.info/
[8] https://www.openmp.org/

2.3. Managing Parallel Audio Data in Qubx

The operational principle of the entire Qubx system (Figure 3) follows a relatively straightforward approach. It employs a dynamic array (unbounded) to manage a variable number of queues, with the quantity of queues being directly proportional to the number of active parallel audio streams.

Each audio stream is partitioned into frames of predefined length, processed sequentially, and inserted into the respective queue.

The array of queues serves as a shared resource between the main thread and the thread dedicated to outgoing audio streaming. The latter continually traverses the array of queues, extracting frames with identical temporal positions from each queue and subjecting them to a summation process. This process yields a single aggregated frame, which undergoes processing and is promptly directed to the output.

Consider a scenario where we have N audio streams, represented as A, each of which is segmented into M frames, denoted as f. Every frame undergoes sequential processing individually (1).

$$A_n \rightarrow \{DSP(f_0), DSP(f_1), DSP(f_2), \dots, DSP(f_M)\} \tag{1}$$

Where A_n represents the n-th audio stream, and let the array of queues be denoted by Q, where Q_i denoting the i-th queue q.

$$Q = \{q_1, q_2, q_3, \dots, q_N\} \tag{2}$$

Each queue q contains a series of frames, denoted as F_{ij}, where j indicates the index of the processed frame within the respective queue.

The resulting frame R_j (3) is processed and put out.

$$R_j = \sum_{i=0}^{N} F_{ij} \qquad DSP(R_j) \rightarrow out \tag{3}$$

Where N is the total number of queues present in Q.

As depicted in the operational diagram in Figure 3, N independent streams imply T_n independent processes (4).

$$T: \{A_1, A_2, A_3, \dots, A_N\} = \{T_1, T_2, T_3, \dots, T_N\} \tag{4}$$

Here, T_n represents the n-th thread associated with the n-th stream A_n.

The state S_n of each $T(A_n)$ is stored in a shared dictionary D between the main thread and the thread dedicated to monitoring the state of each $T(A_n)$ (Thread 2 in Figure 3) (5).

$$D: \{(T_{id}(A_1), S_1), (T_{id}(A_2), S_2), , \dots, (T_{id}(A_N), S_N)\} \tag{5}$$

Where *id* is the unique identifier of the n-th process $T(A_n)$.

When $T(A_n)$ completes its task, the state S_n in $D(T_{id}(A_n), S_n)$ will be updated from busy to free, and subsequently terminated and removed from D.

The monitoring process assumes a pivotal role in system management by upholding responsiveness, ensuring seamless continuation of main execution even in the event of secondary thread suspension. Thread suspension, representing a blocking operation, may necessitate waiting for its completion before proceeding with other tasks. Moreover, terminating processes upon task completion optimizes memory and CPU resource utilization. Essentially, process termination involves releasing previously allocated memory and CPU resources, fostering theoretically and practically more efficient system utilization.

Under conditions of excessive computational load, data processing (DSP in (1)) is performed by exploiting data parallelization. This procedure involves dividing a dataset into n subsets, with each subset being processed by one of the n computing units.

Starting from (1), if we consider that the *DSP* operation processes the entire set of frames A_n sequentially, then we could represent a parallelization context as in (6), where each f_i is processed by a separate computing unit, independently and simultaneously.

$$DSP(A_n) = \bigcup_{i=0}^{M} DSP(f_i) \tag{6}$$

The total execution time T_{max} (7) will depend on the maximum execution time among all n computing units.

$$T_{max} = \max(T_i) \qquad i \in \{0, 1, 2, \dots, M\} \tag{7}$$

3. IMPLEMENTATION AND USAGE OVERVIEW

3.1. Implementation Overview

The implementation of this approach in Rust differs significantly from approaches in other languages, particularly concerning memory management. Rust is specifically designed to prioritize safety and performance, employing a system centered around data ownership and advanced types. This design allows for the assurance, at compile time, of the absence of common memory management errors such as dangling pointers, buffer overflows, and race conditions. While this emphasis on memory control results in a more robust implementation, it may potentially lead to code that is less readable and fluid.

The source code for this implementation is available at the following url: *https://github.com/PasqualeMainolfi/Qubx.git.*

In Rust, handling operational flows involving concurrent blocks can be efficiently and safely managed using the standard module `std::thread`[9]. To guarantee exclusive access to shared data among threads, Rust provides specific types and methods, including `Arc`, `Mutex`, `lock()`, and `drop()`, which are utilized in the developed of the proposed approach.

`Arc` (Atomic Reference Counting) is a data type defined in `std::sync::Arc` that provides atomic reference counting, enabling concurrent data sharing while preserving integrity and adhering to ownership priciples.

[9] https://doc.rust-lang.org/std/thread/

`Mutex` (Mutual Exclusion), defined in `std::sync::Mutex`, ensure exclusive access to shared data among processes. When a thread acquires a mutex lock (see 2.2), no other thread can access the data until the lock is released.

```
impl QList {
    pub fn new() -> Self {
        let q: Vec<ConcurrentQueue<Vec<f32>>> = Vec::new();
        Self {
            qlist: q,
            length: 0,
            index: 0,
        }
    }

    pub fn initialize(&mut self) {
        self.qlist.push(ConcurrentQueue::<Vec<f32>>::unbounded());
        self.length += 1;
    }

    pub fn put_frame(&mut self, frame: Vec<f32>) {
        self.qlist[self.index].push(frame).unwrap()
    }

    pub fn get_frame(&mut self, index: usize) -> Vec<f32> {
        self.qlist[index].pop().unwrap()
    }

    pub fn is_empty_at_index(&self, index: usize) -> bool {
        self.qlist[index].is_empty()
    }

    pub fn get_next_empty_queue(&mut self) {
        let mut counter = 0;
        while counter < self.length {
            if self.is_empty_at_index(self.index) {
                break;
            }

            self.index += 1;
            self.index %= self.length as usize;
            counter += 1;

            if counter >= self.length {
                let q = ConcurrentQueue::<Vec<f32>>::unbounded();
                self.qlist.push(q);
                self.length += 1;
                self.index = (self.length - 1) as usize;
            }
        }
    }

    pub fn is_all_empty(&self) -> bool {
        self.qlist.iter().all(|x| x.is_empty())
    }
}
```

Figure 4: *This code snippets defines the QList structure, which is designed to efficiently manage a list of concurrent queues. It highlights the structure definition and its main methods.*

This lock is obtained using the `lock()` method on a mutex data which returns a `std::sync::MutexGuard` representing a locked reference to the data protected by the mutex. The `drop()` method enables manual release of the lock, ensuring proper and efficient management of shared resources when they are no longer needed.

Data parallelization is implemented using *rayon*[10] (`par_iter_mut()` method in Qubx), a specific library that enables parallel operations on data collections by automatically leveraging the processor's multiple cores.

The structure for managing parallel audio streams (Figure 4) and the set of shared queues (referred to as QList in Qubx) are designed to optimize resource utilization, minimizing waste and maximizing efficiency in processing data across multiple information streams concurrently.

When QList is tasked with storing a new audio data stream, it verifies the availability of empty queues to accommodate it before creating a new one, thereby dynamically and responsively allocating resources based on contextual demands. QList utilizes the `ConcurrentQueue` module from the *concurrent_queue*[11] library to create and manage concurrent FIFOs.

Continuous monitoring of thread states is undertaken by an additional process solely dedicated to terminating threads that are no longer operational.

```
pub fn start_monitoring_active_processes(&mut self) {
    let monitor_clone = Arc::clone(&self.processes_monitor_ptr);
    let local_run = Arc::clone(&self.run);

    let t = thread::spawn(move || {
        while local_run.load(Ordering::Acquire) {
            let mut m = monitor_clone.lock().unwrap();
            m.remove_inactive_processes();
            drop(m);

            thread::sleep(std::time::Duration::from_secs(1));
        }
    });

    println!("[INFO] Start monitoring process...");
    let mclone = Arc::clone(&self.processes_monitor_ptr);
    let mut pm = mclone.lock().unwrap();
    pm.add_process(Process::new(
        t,
        String::from("MONITOR ACTIVE PROCESSES"),
        ProcessState::On,
    ));
}
```

Figure 5: *Code snippets showcases the implementation of a `start_monitoring_active_processes()` function within the Qubx library. Its role is to initiate a thread for monitoring and managing active processes.*

The concept of a thread's state refers to its operational condition within a process, with an operating state indicating active execution of instructions and a non-operating state indicating completion of computation or idleness.

The monitoring and termination process contributes to additional resource optimization, particularly crucial in resource-

[10] https://docs.rs/rayon/latest/rayon/
[11] https://docs.rs/concurrent-queue/latest/concurrent_queue/

intensive contexts like real-time audio signal processing. Terminating an inactive thread releases all associated resources, including memory, and reduces the system's workload, eliminating the need to monitor or schedule the inactive process for execution.

Importantly, the safe termination of a thread is a blocking action, requiring the parent process to await the completion of the child thread's execution before proceeding further. Implementing an independent process for monitoring and eventual termination allows the main process to continue without the risk of being blocked.

3.2. Usage Overview

Version 0.1.0 of the Qubx system introduces the following functionalities:

- Creation and management of an indefinitely large number of independent processes for controlling outgoing master audio streams (in Qubx, a data of type *QubxMasterProcess*), in parallel, each potentially associated with a different device.
- Creation and management of a potentially indefinitely large number of independent duplex stream processes (*in→dsp→out*, of type *QubxDuplexProcess*), which can be associated with different devices, for the real-time management and manipulation of signals captured in real time.
- Possibility to create an indefinitely large number of processes (of type *QubxDspProcess*) for real-time management and manipulation of sampled audio signals, in parallel.

For the first point, creating a process for controlling an outgoing master audio stream involves allocating an independent queue-based space shared with one or more QubxDspProcess, communicating with a specific output device. Each QubxMasterProcess can be assigned a function for manipulating outgoing data (Figure 6).

```
let mut master_out = q.create_master_streamout(String::from("M1"), stream_params);
master_out.start(|frame| {
    frame.iter_mut().for_each(|sample| { *sample *= 0.7 }) });
```

Figure 6: *Example of defining and starting a QubxMasterProcess.*

The type QubxDuplexProcess is dedicated to capturing sound material in real time through a specific input device, processing these signals in real time, and reproducing them through a specific output device (Figure 7).

```
let mut duplex = q.create_duplex_dsp_process(stream_params);
duplex.start(|frame| frame.to_vec());
```

Figure 7: *Example of defining and starting a QubxDuplexProcess.*

Lastly, a QubxDspProcess is dedicated to encoding, processing, and inserting an audio stream into the respective shared queue for playback. Upon creation, each QubxDspProcess must be associated with a specific QubxMasterProcess. During startup, a specific function can be assigned for processing audio data, and potentially, a single process could generate an indefinite number of parallel streams.

```
let mut dsp_process1 = q.create_parallel_dsp_process(String::from("M1"), true);
let mut dsp_process2 = q.create_parallel_dsp_process(String::from("M1"), true);

loop {

    // do something
    // .
    // .
    // .
    // generates audio_data1 and audio_data2...

    dsp_process1.start(audio_data1, |_audio_data| {
        let y = _audio_data.iter().map(|sample| sample * 0.7).collect();
        y
    });

    dsp_process2.start(audio_data2, |_audio_data| {
       let y = _audio_data.iter().map(|sample| sample * 0.7).collect();
       y
    });

    if !run {
        break;
    }

    let delay = rng.gen_range(0.5..1.0);
    thread::sleep(Duration::from_secs_f32(delay));
}
```

Figure 8: *Example of defining and starting a QubxDspProcess*

As depicted in Figure 8, the creation of a QubxDspProcess requires the unique identifier of a QubxMasterStreamoutProcess (e.g., "M1" in the example in Figure 8) and specifying whether the process will use the data parallelization or not (boolean type, e.g. `true` in Figure 8).

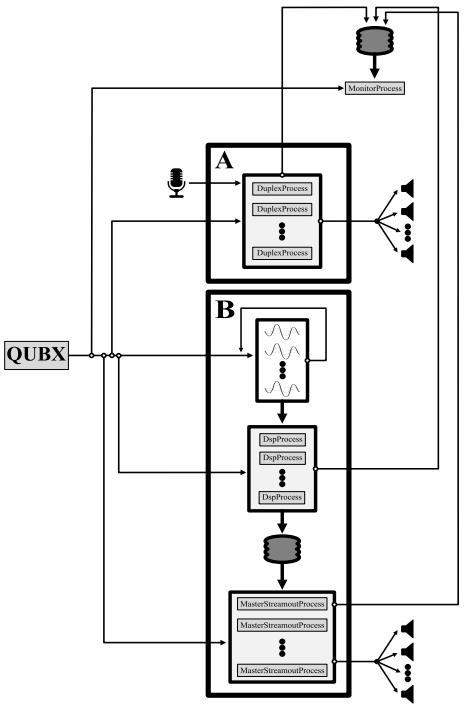

Figure 9: *A more detailed representation of the operational diagram show in Figure 3. Section "A" is dedicated to the QubxDuplexProcess, and in section "B" the QubxMaster-Streamout and DspProcess pair is illustrated.*

The comprehensive documentation can be accessed by compiling the library and generating the documentation using Cargo (see *https://github.com/PasqualeMainolfi/Qubx.git*).

Additionally, an example of Qubx usage implemented for an advanced granular synthesis process is available at: *https://github.com/PasqualeMainolfi/QubxGCT.git.* This example serves to demonstrate its usage and functionality of Qubx.

4. CONCLUSIONS

Qubx emerges as a Rust library, bridging a crucial gap in the ecosystem by furnishing tools dedicated to real-time audio management and processing. Despite the capabilities of Rust, the availability of specific and user-friendly solutions for real-time audio management and manipulation remains limited (see: https://rust.audio/).

The examples provided, including a granulator constructed using the proposed architecture, underscore the efficiency and adaptability of the method across various working scenarios. These tests encompass the utilization and processing of sampled audio files, synthetically generated events, and events captured in real-time.

The capability to manage frame-by-frame processing significantly alleviates computational load, resulting in an estimated average latency of about 6 – 15 ms, encompassing both processing and generation aspects (as observed in the provided examples. By setting *verbose* to *true* during the instantiation of Qubx structure - such as in the following example: `let qubx = Qubx::new(true);` - detailed output, including latency amount for each individual process, is attainable). The term *latency* denotes the duration required for a particular sequence of operations to transpire. Latency, as assessed within a MasterStreamout, encompasses the time necessary to retrieve frames from queues, consolidate, process, and subsequently write them to the output. In a DuplexProcess, it signifies the duration essential for processing a frame and delivering it to the output. Finally, within a DspProcess, it represents the time essential for segmentation, frame-by-frame processing, and the enqueuing of frames into their respective queues. Furthermore, it is imperative to consider the physiological latency linked with the specific input and output devices (refer to Table 1).

Table 1: *Average latency values in Qubx in a non-parallel manner*

	Process Data Type	Device Latency		Iterations Num.	Started Processes Num.	Latency Average (ms)
		Input (ms)	Output (ms)			
audio in	DspProcess	32.698	19.092	149	1	0.116
sampled	MasterStreamout	---	19.092	117	1	0.039
	DspProcess	---	---	---	184	0.012
synthetic	MasterStreamout	---	19.092	117	1	0.024
	DspProcess	---	---	---	179	0.007

In Table 1, the average latency values corresponding to a particular operational context are presented (gathered using a MacBook Pro M3 Max with 32 GB of RAM). Specifically, these contexts include real-time captured audio (input from a microphone), sampled audio events, and synthetically generated audio signals (synthetic), with data parallelization disabled.

However, in scenarios where real-time captured data undergoes processing, the limitation of buffer size becomes apparent. For instance, when granulating data transmitted from any input device on a frame-by-frame basis and employing a phase tracking mechanism to handle grain lengths surpassing that of the buffer - such as through the application of a segmented envelope - input buffers smaller than 1024 samples may encounter operational challenges.

In Table 2, the results obtained by comparing execution with data parallelization activated versus sequential execution are represented.

Table 2: *Average latency values in Qubx in parallel (**PAR.**) and sequential mode (**SEQ.**)*

Process Type	Chunk (samples)	Data Length (samples)	Latency Average (ms)	
			SEQ.	PAR.
MStreamout	4096	$[4.41 \cdot 10^4, 1.3 \cdot 10^5)$	0.023	0.022
DspProcess			0.086	0.331
MStreamout	4096	$[5.2 \cdot 10^6, 13.2 \cdot 10^6)$	0.163	0.157
DspProcess			11.730	8.246
MStreamout	1024	$[13.2 \cdot 10^6, 18.5 \cdot 10^6)$	0.063	0.053
DspProcess			20.569	16.027

The results clearly show that activating data parallelization (PAR. in Table 2) under conditions of relatively light computational load significantly worsens performance compared to sequential processing (SEQ. in Table 2). Conversely, under conditions of excessive load, execution becomes more efficient (~1.3x faster): as the load increases, the difference in execution time between data parallelization and sequential processing becomes increasingly significant.

5. FUTURE WORKS

While Qubx holds promise as a potent and dynamic tool in its domain, there exist numerous opportunities for further expansion and enhancement of its usability and adaptability.

Exploring new methodologies to optimize system performance is crucial. This entails identifying and rectifying potential inefficiencies in the existing codebase while also experimenting with novel approaches to resource sharing and synchronization logic.

To augment the versatility and utility of the library, it would be beneficial to contemplate integrating new features. For instance, incorporating tools for audio data synthesis, analysis, and processing could greatly enhance its functionality and appeal to a broader user base. Additionally, considering the involvement of a community of contributors is essential. Such a community brings with it a diverse range of perspectives, skills, and ideas, ensuring continuous support and maintenance for the library while fostering innovation and growth. Collaboration with a community can lead to accelerated development and the implementation of valuable enhancements, ultimately driving the evolution of Qubx as a leading solution in its field.

6. REFERENCES

[1] R. L. Kruse, “Data Structures & Program Design, Englewood Cliffs”, New Jersey: Prentice Hall, 1987.

[2] J. Bullinaria, “Data Structures and Algorithms”, Lecture notes, University of Birmingham, 2019, Available at https://www.cs.bham.ac.uk/~jxb/DSA/dsa.pdf, Accessed February 16, 2024.

[3] B.V. Zanden, "Queues", Lecture notes CS140 *Data Structures*, University of Tennessee, 2010, Available at https://web.eecs.utk.edu/~bvanderz/teaching/cs140Fa10/notes/Queues/, Accessed February 16, 2024.

[4] S. Mohapatra, "Data Structure Using C", CET, Bhudaneswar, 2020, Available at https://www.cet.edu.in/noticefiles/280_DS%20Complete.pdf, Accessed February 16, 2024.

[5] R. Thrareja, "Data Structures Using C", Oxford University Press, Second Edition, 2014.

[6] IIT Kharagpur, "Stacks and queues", CS13002 Programming and Data Structures, IIT Kharagpur, 2006, Available at https://cse.iitkgp.ac.in/pds/notes/stackqueue.html, Accessed February 16 2024.

[7] B. Sonntag, and D. Colnet, "RIP Linked List", arXiv preprint arXiv:2306.06942, 2023.

[8] D. Hoang, "Performance of array vs. linked-list on modern computers", Available at https://dzone.com/articles/performance-of-array-vs-linked-list-on-modern-comp, Accessed February 20, 2024.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms Third Edition", MIT Press, 2009.

[10] L. Spracklen, and S. G. Abraham, "Chip multithreading: opportunities and challenges", *In 11th International Symposium on High-Performance Computer Architecture*, San Francisco, CA, USA, 2005, pp. 248-252.

[11] P. Ancilotti and M. Boari, "Programmazione concorrente e distribuita", McGraw-Hill, 2007.

[12] C. Filira, G. Filira, F. Filira, and M. Moro, "Sistemi operativi: Architettura e Programmazione concorrente, 2a edizione Vol. 1, Edizioni Libreria Progetto, 2006.

[13] M. A. Kiefer, K. Molitorisz, J. Bieler and W. F. Tichy, "Parallelizing a Real-Time Audio Application -- A Case Study in Multithreaded Software Engineering," *IEEE International Parallel and Distributed Processing Symposium Workshop*, Hyderabad, India, 2015, pp. 405-414, doi: 10.1109/IPDPSW.2015.32.

[14] A. Chaudhary, A. Freed, and D. Wessel, "Exploiting Parallelism in Real-Time Music and Audio Applications, Computing in Object-Oriented Parallel Environments," *Third International Symposium*, ISCOPE 99, San Francisco, CA, USA, 1999, doi: 10.1007/10704054_5.

[15] J. Jiménez-Sauma, "Real-Time Multi-Track Mixing For Live Performance," Zenodo, 2019.

[16] R. Angelov, and A. Ezequíel Viso, "Implementing Real-Time Parallel Audio DSP on GPUs," Audio Developer Conference, 2022.

[17] T. Doumler, "Thread synchronisation in real-time audio processing with RCU (Read-Copy-Update)," Audio Developer Conference, 2022.

[18] M. Noseda, F. Frei, A. Rust, and S. Kunzli, "Rust for secure iot applications: why c is getting rusty," *In Embedded World Conference 2022*, Nuremberg, 21-23 June 2022. WEKA, 2022.

[19] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamaric, and L. Ryzhyk, "System programming in rust: Beyond safety," *In Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pp. 156–161, 2017.

[20] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, 2021.

[21] M. P. Rooney, and S. J. Matthews, "Evaluating FFT performance of the C and Rust Languages on Raspberry Pi platforms," *In 2023 57th annual Conference on Information Sciences and Systems (CISS)*, (pp. 1-6). IEEE, 2023.

[22] C. Fougeray, "Rust for Low Power Digital Signal Processing," Available at https://interrupt.memfault.com/blog/rust-for-digital-signal-processing#fn:4, Accessed May 3, 2024.

[23] A. Namavari, "DAWPL: A Simple Rust Based DSL For Algorithmic Composition and Music Production", Standford University USA, 2017.

[24] M. Puckette, "Multiprocessing for pd," *in Proc. of the 3rd Int'l Pure Data Convention (PDCON09)*, 2009.

[25] S. Letz, Y. Orlay, D. Fober, "Work Stealing Scheduler for Automatic Parallelization in Faust," *In Linux Audio Conference*, 2010.

[26] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni, "Read-copy update," *In AUUG Conference Proceedings*, (Vol. 175), AUUG, Inc., 2001.

[27] P. E. McKenney, and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," *In Parallel and Distributed Computing and Systems*, (Vol. 509518, pp. 509-518), 1998.

[28] Crill, Cross-Platform Real-Time I/O and Low-Latency Library, Available at https://github.com/crill-dev/crill, Accessed May 15, 2024.