Synchronisation de données inter-processus dans les applications audio temps réel: qu'est-ce qui débloque?

Thibaut Carpentier

SYNCHRONISATION DE DONNÉES INTER-PROCESSUS DANS LES APPLICATIONS AUDIO TEMPS RÉEL : QU'EST-CE QUI DÉBLOQUE ?

Thibaut Carpentier
CNRS – Ircam – Sorbonne Université – Ministère de la Culture
Sciences et Technologies de la Musique et du Son (UMR 9912 STMS)
1, place Igor Stravinsky, 75004 Paris
thibaut.carpentier@ircam.fr

RÉSUMÉ

Cet article expose des considérations pragmatiques pour le développement d'applications audio temps réel. Il passe en revue plusieurs concepts fondamentaux qui doivent être pris en compte lors de l'élaboration de telles applications. En particulier, les applications audio sont intrinsèquement *multi-thread* et asynchrones, et la synchronisation de ressources partagées entre plusieurs processus concurrents doit donc faire l'objet d'une attention minutieuse. Les concepts exposés ici ne sont pas nouveaux, toutefois leur mise en pratique, dans un contexte concret de production, demeure un défi majeur pour tous les développeurs audio.

Enfin, la thèse soutenue dans cette étude est que les mécanismes de synchronisation dits *lock-free* ne sont généralement pas indispensables et doivent être évités autant que faire se peut. Une approche par *mutex* non-bloquant est proposée comme substitut simple et satisfaisant.

1. INTRODUCTION

1.1. Anatomie d'une application audio temps réel

L'expression "application audio temps réel" se réfère, dans son acception la plus répandue, à des logiciels [1] qui génèrent, analysent, ou transforment des signaux audio-numériques, en un temps suffisamment court pour que la latence induite soit auditivement imperceptible (de l'ordre de quelques millisecondes). Ces logiciels sont majoritairement mis en œuvre sur des systèmes d'exploitation "conventionnels" tels que macOS, Linux, ou Windows. Il est à noter que lesdits systèmes d'exploitation (*Operating Systems*, OS) ne sont pas des systèmes temps réel au sens strict (*hard real-time*) : le temps maximum entre un stimulus d'entrée et une réponse de sortie ne peut être garanti de façon déterministe. Lorsque ce temps excède les contraintes du temps réel, la "qualité du service" se dégrade, mais le système reste opérationnel et continue de fonctionner ; on parle dans ce cas de temps réel souple (*soft real-time*).

1 . Les hardwares dédiés (DSP, FPGA, etc.) sortent du cadre de cet article.

Les logiciels audio se présentent généralement sous forme d'applications autonomes (*standalones*) ou de *plugins* insérables dans des environnements hôtes, tels que des stations de travail audionumériques (*Digital Audio Workstations*, DAW) ou des *frameworks* multimédia interactifs tels que Max [48], PureData [49], SuperCollider [37], Unity3D, etc. Pour des raisons de performances et d'inter-opérabilité avec leurs environnements hôtes, la vaste majorité des applications audio temps réel est codée en langage `C` ou `C++`. Ce dernier servira d'exemple pour le présent article ; toutefois la plupart des concepts exposés sont transposables à d'autres langages.

1.2. Un contexte intrinsèquement concurrent

Les OS et les applications hôtes considérés ici s'appuient fortement sur des paradigmes de programmation concurrente : pour offrir une interaction riche avec l'environnement externe, et pour tirer profit des systèmes multi-cœurs, plusieurs processus (ou *threads*) sont exécutés en parallèle. Les applications audio doivent donc s'intégrer dans de tels contextes intrinsèquement concurrents. Typiquement plusieurs *threads* sont exécutés "simultanément", afin de gérer le traitement des échantillons audio, les flux de données entrantes (instruments MIDI, capteurs, vidéo, etc.), ou des interactions de l'utilisateur (clavier, souris, etc.). Évidemment, la concurrence ici en jeu est de nature *coopérative*, c'est-à-dire que les différents processus ne sont pas isolés : pour le bon fonctionnement de l'application, ils doivent interagir, notamment en s'échangeant des ressources.

Une conséquence induite par la programmation concurrente est le phénomène dit "d'un indéterminisme d'exécution" : l'ordre d'exécution des instructions n'est plus séquentiel, mais il est soumis à une politique d'ordonnancement (*scheduling*) qui est déterminée par le noyau du système d'exploitation et par les règles de l'environnement hôte. Il en résulte que l'accès aux ressources partagées entre les processus doit être protégé : le développeur doit assurer leur intégrité, typiquement en mettant en œuvre des mécanismes de synchronisation (qui seront discutés plus loin).

Thibaut Carpentier, *Synchronisation de données inter-processus dans les applications audio temps réel : qu'est-ce qui débloque ?*, Journées d'Informatique Musicale (JIM 2018), Amiens, France, éd. L. Bigo, M. Giraud, R. Groult, F. Levé, pp. 35–44, 2018.

1.3. Ordonnancement

Nous considérons le comportement de l'environnement temps réel Max comme un exemple canonique. Il s'agit certes d'un cas particulier, mais celui-ci tout en étant simple s'avère représentatif de la majorité des environnements audio. Dans Max, les processeurs audio peuvent être pilotés par des événements (messages, clavier MIDI, clic souris, etc.). Comparativement à la cadence audio (*audio-rate*, typiquement 48000 échantillons par seconde), ces événements surviennent de façon sporadique ; ils sont dits *control-rate*. Les événements peuvent être à haute priorité (i.e. nécessitant une grande précision temporelle, par exemple métronome, messages MIDI) ou à basse priorité (interaction avec souris/clavier, ou plus généralement avec des interfaces graphiques). Les événements à basse priorité sont toujours traités dans le *thread* principal (*main thread*) de l'application ; selon les réglages de Max (paramètre *overdrive*) un second *thread* peut être activé pour gérer les événements à haute priorité (si l'*overdrive* est désactivé, tous les événements transitent dans le *thread* principal). Les événements à haute priorité peuvent donc potentiellement interrompre le *thread* principal, et s'exécuter avant les événements à basse priorité, selon un ordre non déterministe. De la même manière, dans une DAW, un *plugin* pourra recevoir des événements depuis le *thread* principal (interface graphique) et depuis un *thread* séparé pour l'automation [2].

Le *scheduler* de Max peut gérer la priorité des événements et les affecter au *thread* haute ou basse priorité (voire à d'autres *threads*). Tous les *threads* restent toutefois tributaires de l'ordonnancement du noyau de l'OS qui régule l'accès des processus au CPU. Enfin, les échantillons audio sont traités dans un *thread* spécifique : le *thread* audio.

1.4. Le *thread* audio

Une application audionumérique délivre un flux d'échantillons audio vers le convertisseur numérique/analogique (*digital to analog converter*, DAC) de l'interface audio (carte son). Les échantillons sont produits à une cadence fixe, imposée par la fréquence d'échantillonnage de l'interface. Sur les OS conventionnels, les échantillons ne sont pas délivrés un par un, mais par bloc (*buffer*) typiquement de 32 à 512 échantillons. Les *buffers* sont convoyés vers le driver audio par l'intermédiaire d'une couche de l'OS, e.g. CoreAudio sous macOS, ALSA sous Linux, ASIO sous Windows. Pour ce faire, l'application audio reçoit périodiquement un appel (*callback*) de la couche audio. Ce *callback* survient toutes les N millisecondes (N=1000×buffersize/samplerate) ; il fournit le(s) *buffer(s)* des échantillons d'entrée, et l'application audio doit remplir les *buffers* de sortie. Ce mécanisme de traitement par blocs introduit une certaine latence d'entrée/sortie, proportionnelle à la taille de bloc, et généralement ajustable par l'utilisateur final (dans l'application hôte).

Le *callback* survient dans un *thread* spécifique, appelé *thread* audio et parfois qualifié de "temps réel". Sur les OS traditionnels, le *thread* audio est en fait un *thread* à "temps borné" (*time constraint policy*), généralement préemptable, et qui doit, pour garantir un fonctionnement correct, exécuter sa tâche avant une échéance donnée (*deadline*). Typiquement, un tel *thread* indique à l'OS sa périodicité nominale N, sa durée nominale d'exécution, et sa contrainte maximale (durée minimale d'exécution). Ces données sont comme une promesse faite à l'OS, et sur cette base, le *scheduler* a la responsabilité d'allouer des ressources et d'ordonnancer les différents processus de sorte à servir au mieux les différents *threads* courants.

Si l'échéance du *thread* audio n'est pas satisfaite, des artefacts (*clicks, glitches*) sont audibles. Plusieurs phénomènes peuvent en être la cause : *a*) un défaut du *scheduler* de l'OS (rare), *b*) le code de traitement audio n'est pas assez rapide pour s'exécuter dans le temps imparti (ne "tient pas" le temps réel), ou *c*) le code utilisé n'a pas un temps d'exécution déterministe ou borné. Le problème *b*) est lié à l'efficacité du code et/ou des algorithmes employés ; des techniques d'optimisation des performances peuvent alors être mises en jeu, mais ceci sort du cadre de cet article. Pour éviter le phénomène *c*), on trouve [3] un certain nombre de règles de bonne conduite à respecter dans le *thread* audio :

- éviter les opérations (potentiellement) bloquantes telles que l'acquisition de verrous (*locks*), la lecture de données sur disque ou sur une *socket* réseau, etc. ;
- proscrire les allocations mémoire (`std::malloc`, `std::free`, opérateurs `new` ou `delete`, etc.) car elles peuvent être bloquantes ou, selon leur implémentation, basée sur des algorithmes à temps non déterministe. La mémoire utilisée dans le *thread* audio doit être pré-allouée à l'avance dans des *threads* non-critiques. Aussi, la réservation de zones mémoire (*memory pool*) est fréquemment recommandée ;
- désactiver si besoin le ramasse-miette (*garbage collector*) ; ceci ne concerne pas `C++`, mais peut affecter d'autres langages tels que `Java` ou `C#` ;
- ne pas appeler de code tiers si son comportement (temps d'exécution) n'est pas connu et prédictible ; ceci concerne également tout appel à des fonctions de l'OS qui, sauf preuve du contraire, doivent être considérées comme bloquantes. De même, tout appel à `Objective-C` ou `Swift` est interdit puisque la résolution des messages durant l'exécution (*dynamic dispatch*) peut déclencher des verrous. Signalons ici qu'il existe des outils d'analyse statique de code permettant de détecter les appels à des fonctions non fiables (voir par exemple [38]) ;
- dans le choix des algorithmes employés, ne pas tenir compte de leur complexité en moyenne (ni leur complexité amortie), mais de leur complexité dans le cas le plus défavorable (*worst-case complexity*). De façon générale, lisser la charge de travail (sur plusieurs *callbacks*), et éviter tout algorithme pouvant manifester des pics de calcul (CPU *spikes*) ;

2 . Les mécanismes peuvent varier sensiblement d'une DAW à l'autre.

3 . essentiellement sur des sites web dédiés au développement `C++`, mais on pourra aussi se référer à [19, 12, 47, 51].

1.5. Problématique

Ainsi, les processeurs audio temps réel doivent s'inscrire dans des contextes *multi-thread*, et échanger de façon asynchrone des données avec le *callback* audio qui est soumis à d'importantes contraintes pour s'exécuter en un temps borné. Comment garantir alors un accès *thread-safe* aux ressources partagées par le *thread* audio ? Cette problématique n'est pas nouvelle : les paradigmes de programmation concurrente et les mécanismes de synchronisation inter-processus sont, d'un point de vue théorique, bien maîtrisés et largement discutés dans la littérature [50, 46, 4]. Toutefois leur mise en œuvre pratique, notamment dans un contexte de développement d'applications audio temps réel, demeure un défi et une source de questionnement pour les développeurs.

2. MÉCANISMES DE SYNCHRONISATION DE DONNÉES ENTRE *THREADS*

2.1. Le modèle de mémoire en C++

C et C++ sont spécifiés comme des langages séquentiels à *thread* unique (*single-threaded languages*), la gestion des *threads* étant reléguée dans des bibliothèques de l'OS (par exemple `pthread` sur les systèmes Posix), ou des APIs dédiées (par exemple `OpenMP` [4]). En conséquence, les compilateurs n'ont pour l'essentiel pas connaissance des *threads*, et ils sont autorisés à effectuer un certain nombre de transformations du code, en particulier ré-ordonner l'affectation de variables indépendantes tant que ces transformations préservent la consistance du programme séquentiel *single-thread*. Dans des contextes *mulit-thread*, l'accès aux données peut engendrer des problèmes de condition de concurrence (*race condition* ou *data race*). Le modèle de mémoire (*memory model*, voir [10, 1, 11, 8, 57]) du langage décrit la sémantique des variables partagées, i.e. qu'il spécifie la ou les valeurs qu'elles peuvent retourner dans un programme *mulit-thread* [5]. Dans le cas de C++, le *memory model* stipule que deux actions (*threads*) sont en conflit si elles accèdent au même emplacement mémoire (e.g. à la même variable) et que l'une au moins des actions est une écriture. Par exemple lire une variable x depuis un *thread* A, alors qu'un *thread* B écrit dans x constitue une *data race*. C++ garantit l'exécution séquentielle correcte (*sequential consistency* [33, 1]) des programmes sans *race condition* ; en présence de *data race*, la sémantique et le comportement ne sont pas définis (*undefined behaviour* : tout et n'importe quoi peut alors se produire, éventuellement un crash du programme [9]). Il est à noter que le *memory model* de C++ est sensiblement différent, par exemple, de Java ou C# [36, 44, 1]. Par ailleurs, signalons qu'il existe également un *memory model* au niveau *hardware* car les processeurs, à l'instar des compilateurs, peuvent ré-ordonner les instructions. Évidemment les modèles *software* et *hardware* se doivent d'être compatibles.

Pour permettre l'accès concurrent à des variables normales conformément au *memory model*, c'est-à-dire en évitant les *data races*, plusieurs primitives de synchronisation sont disponibles.

2.2. Verrous de protection

2.2.1. Sections critiques

Le mécanisme le plus communément usité pour éviter les *race conditions* est d'employer des verrous de protection (*locks*) qui permettent de garantir un accès mutuellement exclusif à une donnée (ou structure de données). Les primitives de synchronisation sont appelées *mutex* (*mutual exclusion*). Un *thread* peut verrouiller le *mutex* pendant qu'il accède à la donnée partagée ; on dira qu'il *possède* le *mutex*. Tant qu'il possède le *mutex*, tous les autres *threads* sont bloqués (i.e. mis en attente) s'ils essayent d'accéder à la ressource. La zone de code protégée par le *mutex* est désignée "section critique". Ce mécanisme garantit que tous les *threads* voient toujours une version consistante de la donnée.

Depuis `C++11`, la bibliothèque standard (*Standard Template Library*, STL) fournit plusieurs variantes de *mutex* (`std::mutex`, `std::recursive_mutex`, etc.) ainsi que d'utiles wrappers (par exemple `std::lock_guard`) qui facilitent leur mise en œuvre selon un idiome RAII [6]. Avec `C++14` et `C++17`, la STL s'enrichit encore avec les ajouts de `std::shared_mutex`, `std::shared_lock`, `std::scoped_lock`, etc. L'accès à ces fonctionnalités dans la STL simplifie grandement la tâche des développeurs, les affranchissant des différences d'implémentation entre OS [7].

2.2.2. Inconvénients

L'utilisation de *mutex* s'accompagne toutefois d'un certain nombre de dangers. En particulier, ils peuvent conduire à une situation d'inter-blocage (*deadlock*) lorsque deux processus concurrents s'attendent mutuellement, essayant d'acquérir deux *mutex* dans un ordre différent [8]. Il existe des recommandations et des algorithmes dédiés pour se prémunir de ce phénomène.

Dans un contexte audio temps réel, d'autres inconvénients des *mutex* sont encore à prendre en considération :

— Si une ressource est bloquée par un *mutex*, les autres *threads* ne peuvent savoir pendant combien de temps elle sera bloquée. En outre, l'acquisition d'un *mutex* est tributaire du *scheduling* opéré par le noyau de l'OS. Le temps d'acquisition d'un *mutex* n'est donc pas prédictible de façon déterministe.

4 . `www.openmp.org`

5 . Il est surprenant de noter que le *memory model multi-thread* n'apparaît qu'à partir de la révision `C++11`.

6 . Resource acquisition is initialization (RAII) : par exemple la classe `std::lock_guard` verrouille le *mutex* dès son constructeur, et le débloque dans son déstructeur donc dès qu'il disparaît du *scope* courant.

7 . Il existe en pratique plusieurs stratégies pour implémenter les *mutex* (e.g. *binary semaphores*, *spinlocks*, *ticket locks*) mais cela sort du cadre de cet article.

8 . Exemple : un *thread* `t1` acquiert un *mutex* `m1`, un *thread* `t2` acquiert un *mutex* `m2`, `t1` attend pour acquérir `m2`, `t2` attend pour acquérir `m1`.

— Les *mutex* peuvent provoquer un scénario d'inversion de priorité (*priority inversion*) : lorsqu'une tâche H à haute priorité attend un *mutex* détenu par une tâche B à basse priorité, B peut se retrouver préemptée par une tierce tâche M de priorité intermédiaire, inversant l'ordonnancement prévu ($M < H$) et bloquant l'exécution de H.
— Enfin les *mutex* peuvent générer un problème de performance dit *lock convoy* lorsque des tâches de même priorité tentent d'acquérir de façon répétée un *mutex* ; les *threads* ne sont pas bloqués, mais à chaque tentative avortée d'acquisition du *mutex*, le *thread* doit renoncer à son quantum auprès du *scheduler*, et il provoque une commutation de contexte (*context switch*). Ceci contribue à dégrader les performances du système.

Notons qu'il existe d'autres problèmes à prendre à compte lors de l'usage de *mutex* (e.g. statut d'un *thread* tué tandis qu'il détient un *mutex*) mais ceux-ci ne sont pas directement pertinents pour notre étude.

2.3. Les approches non-bloquantes

Pour pallier les inconvénients des verrous, des techniques de programmation non-bloquantes, généralement dites *lock-free*, ont été proposées. La notion de *lock-free* fait référence à l'accès *thread-safe* (i.e. exempt de *race conditions*) à des données partagées, sans le recours à des primitives telles que les *mutex*. On recense trois niveaux de stratégies non-bloquantes [25] :

— Programmation *wait-free* [31] : il s'agit de la garantie "la plus forte". Un système est dit sans attente (*wait-free*) si tous les processus sont garantis de toujours progresser ; aucun *thread* n'est entravé par un autre. Il est donc assuré que toute opération pourra se réaliser un nombre fini d'instructions. En pratique, l'implémentation d'algorithmes *wait-free* est très difficile et assez rare. Ils sont toutefois employés dans des contextes temps réel strict (*hard real-time*) [21].
— Programmation *lock-free* : les algorithmes *lock-free* garantissent qu'au moins l'un des *threads* pourra toujours progresser. En particulier, si l'un des *threads* est préemptés tandis qu'il travaille sur une ressource *lock-free*, il ne bloque pas les autres *threads* qui peuvent accéder à cette ressource et donc converger vers un résultat en un nombre fini d'instructions. Des trois approches non-bloquantes, le paradigme *lock-free* est incontestablement le plus répandu. Notons que tous les algorithmes *wait-free* sont nécessairement *lock-free*.
— Programmation *obstruction-free* [30] : il s'agit de la garantie "la plus faible". Dans ce paradigme, une opération concurrente peut être annulée et ré-essayée ultérieurement, au profit des autres *threads* en compétition. Un algorithme est donc *obstruction-free* si, à un instant donné, un *thread* "isolé" (tous les autres *threads* en concurrence étant suspendus) peut accomplir ses opérations en un nombre fini d'instructions. Cette approche permet d'éviter les situations d'inter-blocage et d'inversion de priorité, mais peut causer des phénomènes de famine (*livelock*) lorsque deux opérations mutuellement en compétition s'annulent l'une l'autre. Notons enfin que tous les algorithmes *lock-free* sont nécessairement *obstruction-free*.

2.4. Implémentation pratique des structures *lock-free*

La conception d'algorithmes *lock-free* se révèle très difficile, même d'un point de vue formel. En pratique, on développe plutôt des structures de données *lock-free* (voir paragraphe 2.4.2). Celles-ci sont fréquemment usitées ou recommandées dans le domaine des applications audio temps réel [23, 24, 35, 19, 54, 53, 6, 7, 18, 55]. L'implémentation de telles structures *lock-free* s'appuie sur des variables dites de synchronisation (par opposition aux variables normales) et qui permettent l'exécution d'opération atomiques.

2.4.1. Opérations atomiques

On dit d'une opération qu'elle est atomique si elle est indivisible, i.e. qu'elle ne peut être interrompue. Soit l'opération s'exécute en entier, soit elle ne s'exécute pas du tout, et il ne peut y avoir d'état intermédiaire.

Depuis `C++11`, des types atomiques `std::atomic` sont disponibles dans la STL. Toutes les opérations qu'ils permettent sont atomiques, et ce sont les seules opérations strictement atomiques offertes par le langage [9]. `std::atomic` offre un nombre restreint de méthodes telles que `load`, `store`, `fetch_add` et `compare_exchange`. Cette dernière primitive est fondamentale et elle est à la base de toutes les implémentations de structures *lock-free*. Comme son nom l'indique, elle réalise un *compare-and-swap* (aussi appelé *compare-and-exchange*) : cette méthode compare la valeur courante x avec une valeur attendue y ; en cas d'égalité, remplace la valeur courante par une nouvelle valeur désirée z (opération *read-modify-write*) ; sinon, la valeur courante x est chargée (*load*) dans y. L'atomicité de cet échange est rendue possible par des propriétés du processeur (*hardware feature*) et non de l'OS. Notons que ceci introduit nécessairement une synchronisation (barrière) au niveau CPU ; le traitement de variables atomiques est donc significativement plus lent que celui de variables normales [10].

9 . Les fonctionnalités de `std::atomic` sont peu ou prou équivalentes à la notion de `volatile` en langage `Java`. Notons en revanche que le mot-clé `volatile` en `C++` n'a aucun rapport ! En `C++`, `volatile` ne garantit pas l'atomicité, ne permet pas la synchronisation entre *threads* et n'empêche pas le ré-ordonnancement des instructions (ni au niveau du compilateur, ni au niveau hardware).

10 . Par souci d'exhaustivité, indiquons que les opérations sur `std::atomic` peuvent être paramétrées plus finement en spécifiant le mode d'ordonnancement mémoire (*memory ordering*). En effet, le memory model `C++` offre ici plusieurs alternatives (*relaxed memory ordering*, *acquire/release model*, *consume/release model*) qui permettent de contrôler au bas-niveau les barrières mémoires (*memory fences*) employées par le hardware [5]. On parle de *low-level atomics*. Ce contrôle de bas-niveau peut, en théorie, accroître les performances. Toutefois, ces modes de fonctionnement impliquent un assouplissement du memory model, et en particulier la consistance séquentielle du programme (voir paragraphe 2.1) n'est plus garantie. Sans consistance séquentielle, le code devient extrêmement difficile à écrire et pire encore à déboguer ; c'est pourquoi il est généralement déconseillé de s'y risquer, et nous ne nous étendrons pas

`std::atomic<T>` est une classe template dont le paramètre `T` peut prendre un type scalaire (`bool`, `char`, `int`, etc.) à l'exception des flottants (`float` ou `double`). Les scalaires englobant les pointeurs, `T` peut également être une adresse : `std::atomic<T*>`. Ici, il est important de noter que `T` ne peut être un object (`class`) ou un composé (`struct`, `union`, `array`) ; ceux-ci ne peuvent qu'être passés par leur adresse (i.e. par pointeur).

Enfin, signalons que `std::atomic` n'implique pas nécessairement une implémentation *lock-free*. Sur la plupart des processeurs "courants", `std::atomic<T>` est *lock-free* au moins pour les types `T` de 8 bits ; pour des types plus longs, il est possible que le processeur ne dispose pas du jeu d'instruction nécessaire et, de fait, `std::atomic<T>` sera implémenté de façon bloquante (au niveau du système) [11].

2.4.2. De nombreuses structures existantes

Les variables atomiques rendent possible le développement de structures de données *lock-free*, notamment des queues (FIFO), des piles (LIFO), ou des listes chaînées (*singly* ou *doubly linked lists*), etc. Les exemples d'implémentations ne manquent pas dans la littérature [56, 42, 23, 24, 50, 57, 25, 43, 2, 3, 30, 26]. Chacune de ces structures de données a des caractéristiques spécifiques, et le choix d'un *container* approprié dépend de l'application visée. Dans le cas de structures *lock-free*, ce choix est rendu plus difficile par l'existence de nombreuses variantes, fonction du contexte *multi-thread* : il faut en effet distinguer les *threads* qui écrivent dans le *container* (*producers*) de ceux qui y lisent (*consumers*). On trouve ainsi des déclinaisons *single producer-single consumer* (SPSC : cas le plus simple où un *thread* écrit et un autre lit), *multiple producers-multiple consumers* (MPMC : plusieurs *threads* concurrents peuvent écrire tandis que plusieurs *threads* peuvent lire), et les combinaisons SPMC et MPSC. On comprend aisément qu'une structure SPSC requiert moins d'efforts de synchronisation qu'une MPMC ; on peut donc supputer que cette dernière sera moins performante.

Les containers *lock-free* sont le plus souvent implémentés sous forme de *buffers* circulaires de taille fixe, de sorte à pré-allouer l'espace de stockage (statiquement ou durant l'initialisation) et à éviter les allocations dynamiques durant l'exécution.

2.5. Avantages et inconvénients des structures non-bloquantes

La principale motivation et l'avantage putatif de la programmation *lock-free* est la scalabilité, c'est-à-dire la capacité d'exploiter au mieux le parallélisme en augmentant le nombre d'opérations sur une structure de données tout en minimisant le temps d'attente de chacun des *threads*, et en évitant les problèmes de deadlocks et d'inversion de priorité.

Cependant cela s'accompagne d'un certain nombre de difficultés, voire d'inconvénients par rapport aux techniques bloquantes :

— La conception et le développement de structures *lock-free* est considérablement plus difficile que la programmation à base de *mutex*. Il est communément admis que cela doit être strictement réservé aux développeurs les plus aguerris [12]. Bjarne Stroustrup et Herb Sutter ont récemment publié un ensemble de recommandations pour l'usage pertinent et efficace du langage `C++` (Core Guidelines [13]) ; la première règle (CP.100) de la section *lock-free* stipule : "Don't use lock-free programming unless you absolutely have to. Reason : It's error-prone and requires expert level knowledge of language features, machine architecture, and data structures."

— En corollaire, les programmes *lock-free* sont très difficiles à maintenir et à débogguer, d'autant qu'il est compliqué sinon impossible de prouver formellement leur exactitude [34]. Par ailleurs, il est délicat d'évaluer les performances de la programmation non-bloquante et de démontrer ses bénéfices.

— Un autre inconvénient crucial des structures *lock-free* concerne la gestion de la mémoire et de la durée de vie des objets : comme expliqué au paragraphe 2.4.1, les containers *lock-free* de données "complexes" (composées) nécessitent l'usage de `std::atomic<T*>`. L'utilisation d'un pointeur nu `T*` contrevient à toutes les bonnes règles de codage en `C++` moderne, et délègue à l'appelant la gestion mémoire (allocation/libération) des éléments du container. Ceci augmente considérablement la complexité du code et le risque d'erreurs [14]. Pour "pallier" cela, les structures *lock-free* doivent généralement être accompagnées d'un dispositif ad-hoc de gestion de la mémoire, ce qui constitue en soi un défi considérable. Ont été proposés à cet effet : des *pools* d'objets [32], des ramasse-miettes avec compteur de références [22, 29, 27], des *hazard pointers* [40, 3], des approches d'accès optimistes [17], etc.

— Elles peuvent être soumises au problème dit d'échange d'états ABA [20, 21].

— Elles ne constituent pas un remède au problème de contention.

— Certaines implémentations font l'objet de brevets [52, 45].

sur ce sujet.

11 . La méthode `std::atomic<T>::is_lock_free()` permet de tester si l'implémentation est *lock-free* pour le type `T` et l'architecture courante. (Bizarrement, cette méthode est une fonction membre et non une fonction statique de classe.)

12 . Herb Sutter, éminent spécialiste du langage et président du comité de standardisation ISO `C++`, a donné de nombreux keynotes et tenu une colonne sur `www.drdobbs.com` au sujet de la programmation concurrente. Il explique en substance que la programmation *lock-free* est réellement maîtrisée par environ "fifty people in the world (and that's a generous idea)", car "writing lock-free code can confound anyone – even expert programmers" (Lock-Free Code : A False Sense of Security – publié en septembre 2008).

13 . `https://github.com/isocpp/CppCoreGuidelines`

14 . Tempérons un peu en signalant que des propositions concernant `std::atomic_shared_ptr` font l'objet de discussions pour `C++20`. Ceci pourrait grandement simplifier l'usage de structure de données atomiques.

3. UNE APPROCHE SOUS-ESTIMÉE : LES *MUTEX* NON-BLOQUANTS

Revenons à la problématique des applications audio temps réel. Nous l'avons dit dans la section 1, il est généralement inévitable de se confronter à la concurrence *multi-thread*, car le *callback* audio a besoin de communiquer avec un *thread* graphique, des *threads* haute priorité (interfaces MIDI, sockets réseau, etc.) ou basse priorité (accès de fichiers sur disque, etc.). Dans ce contexte concurrent, une queue d'événements est requise. Deux cas d'usage typiques se présentent, et ils couvrent une vaste majorité des applications audio :

— Queue MPSC : différents *threads* (GUI et/ou haute priorité) sont employés pour piloter un moteur audio vers qui ils envoient des messages. Le *callback* audio est dans ce cas le seul et unique *consumer*, et plusieurs *threads* sont *producers* (généralement en nombre assez restreint – deux ou guère plus).
— Queue SPMC : le *thread* audio analyse les signaux et renvoie des valeurs par exemple vers une interface graphique (e.g. spectroscope, mesure de niveaux RMS et vu-mètres, etc.). Dans ce cas le *callback* audio est l'unique *producer*, et les autres *threads* sont *consumers*.

Dans la plupart des usages courants, il est en outre légitime de considérer que le timing des événements n'est pas critique. Typiquement, une réactivité de l'ordre de quelques millisecondes (supérieur à la taille de bloc audio) est satisfaisante (e.g. pour changer les paramètres d'un processeur audio ou rafraîchir la visualisation d'une analyse).

Sous ces hypothèses, nous proposons d'utiliser une technique de synchronisation inter-processus basée sur des *mutex* non-bloquants. Cette technique n'est pas nouvelle, mais elle semble largement sous-estimée et inexploitée. Elle repose simplement sur la méthode `try_lock()` de `std::mutex` qui, comme son nom l'indique, tente d'acquérir le *mutex*. En cas d'échec (si le *mutex* est déjà verrouillé par ailleurs), la méthode retourne immédiatement, sans bloquer. En cas de succès, elle obtient la lock et les autres *threads* ne pourront entrer dans la section critique. La méthode est susceptible de retourner des faux négatifs.

Par exemple, dans le scénario de queue MPSC évoqué, l'écriture des événements par les *threads producers* est protégée par `std::mutex::lock()`, tandis que la lecture de la queue dans le *thread* audio se contente d'un `try_lock`. Si le `try_lock` échoue, l'opération sera re-tentée ultérieurement, par exemple au prochain bloc audio [15]. Le scénario de queue SPMC peut être traité de façon analogue.

Il est en théorie possible que le `try_lock` échoue systématiquement et que les paramètres dans le *thread* audio ne soient donc jamais mis à jour. Nous spéculons que ce phénomène n'a, en pratique, que peu de risque de se produire : le nombre de *threads* concurrents est très restreint, les événements ne surviennent que sporadiquement, et les sections critiques sont très "courtes" (il s'agit de lire ou écrire un élément – e.g. message MIDI ou OSC – dans la queue, ce qui ne requiert que quelques instructions). Pour les mêmes raisons, on peut espérer que les *threads producers* pâtiront peu des blocages de *mutex*, de sorte que le programme demeure réactif.

La technique de *mutex* non-bloquants ici proposée permet donc une synchronisation *thread-safe* qui tire profit de la grande simplicité des *mutex* (l'effort de codage est minimal), sans risquer d'inversion de priorité (le *callback* audio n'étant jamais bloqué). Elle garantit la progression *obstruction-free* du *thread* audio [41].

3.1. Évaluation des performances

La notion de "progression garantie" est indépendante des performances du sytème. Il convient donc d'évaluer l'efficacité de la solution `try_lock`. Les mérites des structures non-bloquantes s'évaluent essentiellement selon deux critères [39] : leur capacité de traitement (*throughput*), c'est-à-dire la quantité de données pouvant être produites–consommées par la structure par unité de temps, et leur scalabilité, autrement dit leur capacité à "fournir plus de travail" lorsque plus de *threads* sont mobilisés. La scalabilité constitue un facteur crucial pour les applications massivement parallèles, mais, comme évoqué précédemment, ce critère est assez peu pertinent pour les programmes audio où le nombre de *threads* concurrents est très faible. Nous nous focaliserons donc sur le *throughput*.

3.2. Protocole

Nous réalisons un test comparatif entre la technique de *mutex* non-bloquants et une approche *lock-free*. Nous considérons le cas d'une queue de capacité fixe (de taille arbitraire $2^{15} = 32768$ éléments). Afin de s'affranchir des problèmes de gestion mémoire (voir paragraphe 2.5), nous testons une queue d'éléments entiers, c'est-à-dire que la queue *lock-free* s'appuie sur `std::atomic<int>`. Pour l'implémentation *lock-free*, nous utilisons une bibliothèque développée par Erik Rigtorp [16]. Celle-ci est simple d'usage (fichiers d'entête seulement), écrite en utilisant des idiomes `C++` modernes, et optimisée (utilisation d'atomiques de bas-niveau, alignement de cache, etc.). Elle propose une queue SPSC et une queue MPMC (il est plus rare de trouver des implémentations MSPC ou SPMC). Les tests sont conduits dans l'environnement Max (7.3.5 en 64 bit), sous macOS (10.13.4), processeur Intel Core i7 (4 Mo de cache L3). Nous évaluons le *throughput* des différentes implémentations en mesurant le nombre d'événements qui peuvent être écrits (*push*) et lus (*pop*) dans la queue pendant une unité de temps. Les événements sont simplement des messages Max convoyant un nombre entier tiré aléatoirement. Le *thread* audio est le seul et unique

15 . Il est également possible d'obtenir une granularité plus fine en re-découpant le bloc audio en sous-blocs de taille plus petite.

16 . `https://github.com/rigtorp/SPSCQueue` `https://github.com/rigtorp/MPMCQueue`

consumer de ces messages. À chaque entrée dans le *callback* audio, l'approche par *mutex* opère un `try_lock` ; s'il est concluant, tous les événements de la queue sont dépilés un par un. Sinon l'exécution continue et un nouvel essai sera réalisé au bloc suivant. Les tests sont menés pour différentes tailles de bloc audio. La répétabilité n'étant pas garantie, les résultats sont moyennés sur trois réalisations de dix secondes chacune.

Dans le test #1, un seul *thread producer* (à haute priorité) est considéré, les messages étant générés par un métronome (objet `metro`). Le test #2 est identique, sinon que la contention globale du programme est accrue, en déclenchant un nombre considérable d'événements concurrents (*stress test*) dans les *threads* haute priorité, basse priorité, et d'autres *threads* de priorité normale. Le test #3 est similaire à #1, mais deux *threads producers* sont mobilisés, en utilisant à la fois `metro` et `qmetro`. Le test #4 est similaire à #3, mais la contention est accrue selon un protocole similaire à #2. Le test #4 représente la situation la plus réaliste d'un processeur audio temps réel en situation de performance. L'implémentation SPSC n'est pas valide lorsque plusieurs *threads* sont *producers*, aussi n'est-elle évaluée que dans les tests #1 et #2.

3.3. Analyse des résultats

Les résultats sont présentés dans la figure 1. Les données absolues n'ayant guère d'utilité ici, nous nous limiterons à des analyses en relatif.

Test #1 : les performances des queues *lock-free* SPSC et MPMC sont sensiblement identiques. Elles se dégradent notablement pour des tailles de bloc de 1024 et 2048. Nous émettons l'hypothèse que ce phénomène pourrait être lié à des problèmes de cache CPU ou d'erreur de page (*cache miss* ou *page fault*). Les performances de l'implémentation `try_lock` ne dépendent (quasiment) pas de la taille de *buffer*. Pour des *buffers* de taille ≤ 512, les queues *lock-free* surpassent nettement le *mutex*, et la variante SPSC est marginalement meilleure que MPMC (comme conjecturé paragraphe 2.4.2).

Test #2 : En présence de contention, les performances des trois implémentations se dégradent nettement. Toutefois la méthode *mutex* (avec une diminution de throughput d'environ 20%) semble plus robuste que les approches *lock-free* (perte d'environ 35%). Le comportement "anormal" des structures *lock-free* pour un bloc de 2048 (et 1024) est similaire au test #1 et demeure inexpliqué.

Test #3 : En présence de *threads producers* concurrents, les performances générales diminuent encore. L'avantage de la queue MPMC sur le `try_lock` devient plus marginal.

Test #4 : Le throughput des implémentations testées est globalement moindre que dans #3. Le gain de MPMC par rapport au *mutex* non-bloquant est approximativement de 10%.

En définitive, les implémentations *lock-free* sont globalement plus performantes que l'approche par *mutex* `try_lock`, toutefois elles sont plus sensibles à la contention. Dans le cas le plus réaliste (test #4), le gain de performances de MPMC est minime, d'autant que ces benchmarks ont été réalisés sur des queues d'entier donc ne tenant pas compte du possible surcoût de gestion dynamique de la mémoire. Les *mutex* font partie des primitives de base de tous les OS courants ; il est donc permis de croire qu'ils sont extrêmement optimisés pour tous les usages "courants", justifiant ainsi leur bon rendement ici observé.

Ces remarques sont proches des conclusions dressées dans [13], dans lequel l'auteur évalue les mérites respectifs des queues *lock-free* (implémentation de la bibliothèque Boost) et à base de lock : "[...] both the lock-based and lock-free queue perform just as well. This is contrary to [19], which suggests that audio programs should use lock-free data structures. These patterns, however, were based on the concurrency primitives that were available to programmers in 2005. The assumption now is that the newer `C++` concurrency primitives perform much better than their older counterparts [17] , meaning that for some applications, the implementation overheads of a lock-free queue may not warrant its use."

4. CONCLUSION

Le développement d'applications audio temps réel nécessite la synchronisation de données partagées entre différents processus et le *thread* audio. En raison des contraintes spécifiques auxquelles est soumis ce dernier, on recommande fréquemment la mise en œuvre de mécanismes *lock-free*, afin d'éviter les phénomènes d'inter-blocage ou d'inversion de priorité. La programmation de structures *lock-free*, basée sur des variables atomiques, est particulièrement ardue et propice aux erreurs car elle requiert une connaissance experte du langage, du *memory model* et de l'architecture *hardware*. Elle complexifie en outre la gestion dynamique des ressources mémoire.

Dans cet article, nous avançons que les approches *lock-free*, si elles sont parfois les plus performantes, ne sont généralement pas indispensables dans un contexte d'audio temps réel, et elles devraient être évitées au profit de l'hygiène du code. Comme substitut simple et pragmatique, nous proposons un paradigme *thread-safe* non-bloquant s'appuyant sur le `try_lock` des *mutex*. Cette méthode garantit une progression *obstruction-free* du *thread* audio. Nous montrons que les performances obtenues sont proches des structures *lock-free*, et viables pour une exploitation en production [18] .

17 . N. Pipenbrinck. Online discussion on lock-free data structures. `http://stackoverflow.com/questions/27738660`

18 . La technique `try_lock` est effectivement mise en œuvre dans plusieurs applications temps réel développées par l'auteur [16, 15, 28, 14].

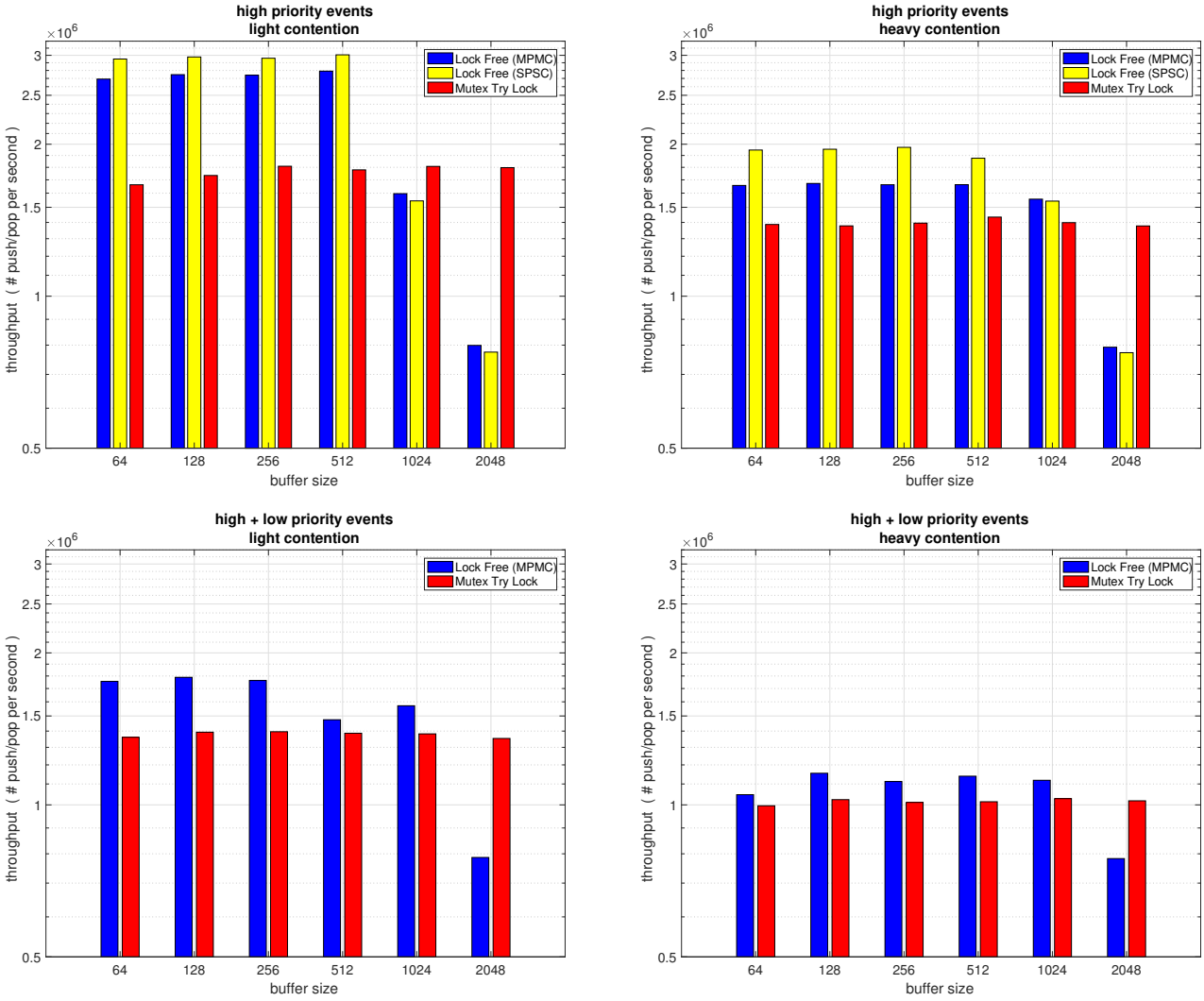

Figure 1. En haut à gauche : test #1. En haut à droite : test #2. En bas à gauche : test #3. En bas à droite : test #4.

5. REFERENCES

[1] Adve, S.V., Boehm, H.J., "Memory Models : A Case for Rethinking Parallel Languages and Hardware", *Communications of the ACM*, volume 53(8), pages 90 – 101, Aug 2010.

[2] Alexandrescu, A., "Lock-Free Data Structures", *C/C++ Users Journal*, Oct 2004.

[3] Alexandrescu, A., Michael, M.M., "Lock-Free Data Structures with Hazard Pointers", *C/C++ Users Journal*, Dec 2004.

[4] Andrews, G.R., *Foundations of Multithreaded, Parallel, and Distributed Programming*, Pearson, 1999.

[5] Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T., "Mathematizing C++ Concurrency", *Proceedings of the 38th annual Symposium on Principles of programming languages (ACM SIGPLAN-SIGACT)*, pages 55 – 66, Austin, TX, USA, Jan 2011.

[6] Bencina, R., "Interfacing real-time audio and file I/O", *Proc. of the Australasian Computer Music Conference (ACMC)*, pages 21 – 28, Melbourne, July 2014.

[7] Blechmann, T., "Supernova – A scalable parallel audio synthesis server for SuperCollider", *Proc. International Computer Music Conference (ICMC)*, Huddersfield, UK, August 2011.

[8] Boehm, H.J., "Threads Cannot Be Implemented As a Library", *Proc of the Conference on Programming Language Design and Implementation (ACM SIGPLAN)*, volume 40, pages 261 – 268, Chicago, IL, USA, June 2005.

[9] Boehm, H.J., "Position Paper : Nondeterminism is unavoidable, but data races are pure evil", *Proc of the ACM workshop on Relaxing synchronization for multicore and manycore scalability*, pages 9 – 14, Tucson, AZ, USA, Oct 2012.

[10] Boehm, H.J., Adve, S.V., "Foundations of the C++ Concurrency Memory Model", *Proc. of the Conference on Programming Language Design and Imple-*

mentation (ACM SIGPLAN), volume 43, pages 68 – 78, Tucson, AZ, USA, June 2008.

[11] Boehm, H.J., Adve, S.V., "You Don't Know Jack about Shared Variables or Memory Models", *ACM Queue*, volume 9(12), Dec 2011.

[12] Boulanger, R., Lazzarini, V., *The Audio Programming Book*, MIT Press, 2011.

[13] Cameron, E., *Parallelizing the ALSA modular audio synthesizer*, Master's thesis, Concordia University, Montreal, Canada, 2015.

[14] Carpentier, T., "ToscA : An OSC Communication Plugin for Object-Oriented Spatialization Authoring", *Proc. of the 41st International Computer Music Conference*, pages 368 – 371, Denton, TX, USA, Sept. 2015.

[15] Carpentier, T., "Panoramix : 3D mixing and post-production workstation", *Proc. 42nd International Computer Music Conference (ICMC)*, pages 122 – 127, Utrecht, Netherlands, Sept 2016.

[16] Carpentier, T., Noisternig, M., Warusfel, O., "Twenty Years of Ircam Spat : Looking Back, Looking Forward", *Proc. of the 41st International Computer Music Conference*, pages 270 – 277, Denton, TX, USA, Sept. 2015.

[17] Cohen, N., Petrank, E., "Efficient Memory Management for Lock-Free Data Structures with Optimistic Access", *Proc. of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 254 – 263, Portland, OR, USA, June 2015.

[18] Colmenares, J.A., Saxton, I., Battenberg, E., Avizienis, R., Peters, N., Asanovic, K., Kubiatowicz, J.D., Wessel, D., "Real-time Musical Applications on an Experimental Operating System for Multi-Core Processors", *Proc. International Computer Music Conference (ICMC)*, Huddersfield, UK, August 2011.

[19] Dannenberg, R.B., Bencina, R., "Design Patterns for Real-Time Computer Music Systems", *Workshop on Real Time Systems Concepts for Computer Music (ICMC)*, Sept 2005.

[20] Dechev, D., Pirkelbauer, P., Stroustrup, B., "Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs", *Proc. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 185– 192, Seville, Spain, May 2010.

[21] Dechev, D., Stroustrup, B., "Scalable Nonblocking Concurrent Objects for Mission Critical Code", *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, Orlando, FL, USA, Oct 2009.

[22] Detlefs, D.L., Martin, P.A., Moir, M., Jr., G.L.S., "Lock-free reference counting", *Distributed Computing*, volume 15(4), pages 255 — 271, 2002.

[23] Fober, D., Orlarey, Y., Letz, S., "Optimised Lock-Free FIFO Queue", Technical report, GRAME - Computer Music Research Lab, Lyon, France, Jan 2001.

[24] Fober, D., Orlarey, Y., Letz, S., "Lock-Free Techniques for Concurrent Access to Shared Objects", *Actes des Journées d'Informatique musicale (JIM)*, pages 143 – 150, Marseille, France, 2002.

[25] Fraser, K., "Practical lock-freedom", Technical report, University of Cambridge Computer Laboratory, Feb 2004.

[26] Fraser, K., Harris, T., "Concurrent Programming Without Locks", *ACM Transactions on Computer Systems (TOCS)*, volume 25(2), May 2007.

[27] Gao, H., Groote, J., Hesselink, W., "Lock-free parallel and concurrent garbage collection by mark & sweep", *Science of Computer Programming*, volume 64, pages 341 — 374, 2007.

[28] Geier, M., Carpentier, T., Noisternig, M., Warusfel, O., "Software tools for object-based audio production using the Audio Definition Model", *Proc. of the 4th International Conference on Spatial Audio (ICSA)*, Graz, Austria, Sept 2017.

[29] Gidenstam, A., Papatriantafilou, M., Sundell, H., Tsigas, P., "Practical and Efficient Lock-Free Garbage Collection Based on Reference Counting", Technical Report no. 2005-04, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, 2005.

[30] Herlihy, M., Luchangco, V., Moir, M., "Obstruction-free synchronization : double-ended queues as an example", *Proc. of the 23rd International Conference on Distributed Computing Systems*, pages 522 – 529, May 2003.

[31] Herlihy, M.P., "Impossibility and universality results for wait-free synchronization", *Proc. of the seventh annual ACM Symposium on Principles of distributed computing (POD)*, pages 276 – 290, Toronto, Canada, August 1988.

[32] Herlihy, M.P., "A methodology for implementing highly concurrent data objects", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 15(5), pages 745 — 770, Nov 1993.

[33] Lamport, L., "How to make a multi-processor computer that correctly executes multiprocess programs", *IEEE Transactions on Computers*, volume C-28(9), pages 690 – 691, Sept 1979.

[34] Lamport, L., "The temporal logic of actions", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 16(3), pages 872 – 923, May 1994.

[35] Letz, S., Fober, D., Orlarey, Y., "Jack audio server for multi-processor machines", *Proc. of the International Computer Music Conference (ICMC)*, pages 1 – 4, Barcelona, Spain, Sept 2005.

[36] Manson, J., Pugh, W., Adve, S.V., "The Java Memory Model", *Proc of the 32nd Symposium on Principles of programming languages (ACM SIGPLAN-SIGACT)*, volume 40, pages 378 – 391, Long Beach, CA, USA, Jan 2005.

[37] McCartney, J., "Rethinking the Computer Music Language : SuperCollider", *Computer Music Journal*, volume 26(4), pages 61 – 68, Winter 2002.

[38] McCurry, M., "STatic (LLVM) Object Analysis Tool : Stoat", *Proc. of the Linux Audio Conference (LAC)*, Saint-Etienne, France, May 2017.

[39] Meneghin, M., Pasetto, D., Franke, H., Petrini, F., Xenidis, J., "Performance evaluation of inter-thread communication mechanisms on multicore/multithreaded architectures", *Proc. of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2012.

[40] Michael, M., "Hazard pointers : Safe memory reclamation for lock-free objects", *IEEE Transactions on Parallel and Distributed Systems*, volume 15(6), pages 491 — 504, 2004.

[41] Michael, M.M., "The Balancing Act of Choosing Nonblocking Features", *Communications of the ACM*, volume 56(9), pages 46 – 53, Sept 2013.

[42] Michael, M.M., Scott, M.L., "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms", *Proc. of the fifteenth annual ACM symposium on Principles of distributed computing (PODC)*, pages 267 – 275, 1996.

[43] Michael, M.M., Scott, M.L., "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors", *Journal of Parallel and Distributed Computing*, volume 51(1), pages 1 – 26, May 1998.

[44] Ostrovsky, I., "The C# Memory Model in Theory and Practice", Dec 2012.

[45] Otenko, O., "System and method for efficient concurrent queue implementation – US Patent 8,607,249 B2", Technical report, Oracle International Corporation, Dec 2013.

[46] Pacheco, P., *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011.

[47] Pirkle, W., *Designing Audio Effect Plug-Ins in C++ : With Digital Audio Signal Processing Theory*, Focal Press, 2012.

[48] Puckette, M., "The Patcher", *Proc. International Computer Music Conference (ICMC)*, pages 420 – 429, San Francisco, CA, USA, 1988.

[49] Puckette, M., "Pure Data", *Proc. of the International Computer Music Conference (ICMC)*, pages 224 – 227, Thessaloniki, Greece, 1997.

[50] Raynal, M., *Concurrent Programming : Algorithms, Principles, and Foundations*, Springer, 2013.

[51] Robinson, M., *Getting Started with JUCE*, Packt Publishing, 2013.

[52] Rushworth, T.B., Telfer, A.R., "Multi-reader, multi-writer lock-free ring buffer", Technical report, Inetco Systems Limited, 2012.

[53] Schmeder, A., Freed, A., Wessel, D., "Best Practices for Open Sound Control", *Proc. of the Linux Audio Conference (LAC)*, Utrecht, Netherlands, May 2010.

[54] Schnell, N., Roebel, A., Schwarz, D., Peeters, G., Borghesi, R., "MuBu and friends–assembling tools for content based real-time interactive audio processing in Max/MSP", *Proc. International Computer Music Conference (ICMC)*, pages 423 – 426, Montreal, Canada, Aug 2009.

[55] Shelton, R.J., *A Lock-Free Environment for Computer Music : Concurrent Components for Computer Supported Cooperative Work*, Ph.D. thesis, University of Melbourne, 2011.

[56] Valois, J.D., "Lock-Free Linked Lists Using Compare-and Swap", *Proc. of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214 – 222, 1995.

[57] Williams, A., *C++ Concurrency in Action Practical Multithreading*, Manning Publications, 2010.