UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Analisi e Implementazione di Code MPMC lock-free in linguaggio C++

Relatore:
Prof. Massimo Torquati

Candidato:
Mattia Piras

Anno Accademico 2024/2025

Indice

Introduzione

Nell'era delle architetture multi-core, lo sviluppo di algoritmi e strutture dati *concorrenti* affidabili e scalabili che possano operare in maniera efficiente su sistemi con numerosi core è diventata una priorità per l'ingegneria del software. Tali algoritmi, capaci di gestire risorse condivise senza compromettere l'integrità dei dati, rappresentano la chiave per costruire applicazioni performanti e robuste, in grado di rispondere alle crescenti esigenze di elaborazione e scalabilità imposte dalle tecnologie hardware/software emergenti.

Con il termine di *algoritmi concorrenti* si identifica l'insieme di tecniche e strategie che consentono a più thread o processi di interagire e operare su risorse condivise in maniera efficiente, senza compromettere l'integrità dei dati. Tuttavia, questa modalità di interazione comporta sfide notevolmente più complesse rispetto alla progettazione di algoritmi sequenziali. La gestione della concorrenza richiede un'attenta valutazione dei meccanismi di sincronizzazione per mantenere la consistenza dei dati durante gli accessi simultanei e l'adozione di tecniche specifiche per evitare situazioni di stallo, in cui i processi si bloccano reciprocamente. In questo contesto, l'approccio tradizionale basato su meccanismi di sincronizzazione bloccanti (quali lock, condition variables e semafori) si rivela particolarmente inefficace, soprattutto in scenari ad alta concorrenza [1].

Gli approcci che utilizzano meccanismi di sincronizzazione *non bloccanti* basati su istruzioni atomiche offrono una valida alternativa per diminuire gli overhead tipici dei meccanismi di sincronizzazione tradizionali. Questi approcci, che includono algoritmi *lock-free* e *wait-free*, mirano a garantire che i thread possano proseguire le loro operazioni senza mai bloccarsi. In un ambiente ad alta concorrenza, gli algoritmi non bloccanti riducono i tempi di attesa e aumentano l'efficienza complessiva del sistema, assicurando, con vari livelli di garanzia, il progresso dei thread. L'idea fondamentale, è quella di permettere a più thread di operare sulla stessa struttura dati senza la necessità di un meccanismo di blocco esterno che garantisca la consistenza dei dati.

Tra vari esempi di algoritmi non bloccanti applicati a strutture dati concorrenti, le code FIFO sono un esempio paradigmatico di questo approccio. Una coda, nella sua essenza, è una struttura dati per cui i metodi seguono la politica *FIFO* (First-In-First-Out): il primo elemento inserito è il primo ad essere rimosso. Questa semplicità concettuale la rende uno strumento estremamente versatile e ampiamente utilizzato in svariati scenari applicativi, dal software di sistema fino al

software applicativo di alto livello. La struttura dati coda concorrente viene spesso utilizzata come un meccanismo di base per *disaccoppiare* i produttori di dati (threads o processi che inseriscono dati nella coda) dai consumatori (che rimuovono i dati dalla coda). Questa separazione offre il vantaggio di poter parallelizzare indipendentemente gli inserimenti (*enqueue*) dei produttori con le estrazioni (*dequeue*) dei consumatori. Tali operazioni solla coda devono essere eseguite in modo *atomico* per evitare corse critiche e garantire la consistenza della struttura dati stessa.

Per garantire accessi atomici alla coda, tradizionalmente si ricorre a meccanismi di sincronizzazione basati sulla *mutua esclusione*. Questi meccanismi prevedono l'incapsulamento di operazioni all'interno di sezioni critiche, in cui un solo processo alla volta può operare: ogni volta che un processo entra nella sezione critica, richiede un mutex tramite un'operazione di lock, mentre gli altri processi devono attendere il rilascio (unlock). Pur essendo efficaci nel garantire la correttezza, tali tecniche introducono un elevato overhead legato al blocco e al successivo riavvio dei thread che in molti casi richiede supporto dal sistema operativo, nonché rischi di incorrere in situazioni di deadlock o innescare problematiche come l'inversione di priorità [2].

In questo contesto, si è sviluppato il presente lavoro di tesi rivolto all'analisi e lo sviluppo di algoritmi non bloccanti in `C++` per l'implementazione di code concorrenti, con l'obiettivo di superare le limitazioni delle tecniche di sincronizzazione classiche e massimizzare le potenzialità delle struttura dati coda FIFO in ambienti ad alta concorrenza. Il `C++` moderno offre un supporto robusto per lo sviluppo di algoritmi non bloccanti grazie alle operazioni atomiche incluse nella sua libreria standard. L'uso di tipi come `std::atomic` permette di realizzare codice portabile su una vasta gamma di architetture hardware, eliminando la necessità di ricorrere a codice assembly *hardware-specific* per accedere alle istruzioni atomiche, semplificando notevolmente la scrittura e manutenibilità del codice.

Scopo della Tesi e lavoro svolto

La tesi esplora le motivazioni fondamentali che sottendono l'adozione di algoritmi non bloccanti nel contesto della programmazione concorrente. L'obiettivo primario è quello di fornire un fondamento semplificato e accessibile che giustifichi l'adozione di queste tecniche, evidenziandone i benefici in termini di robustezza ed efficienza.

Dopo una analisi approfondita degli algoritmi non bloccanti per la gestione di code FIFO, attraverso un approccio bottom-up, si ripercorreranno le principali implementazioni e i miglioramenti proposti dalla letteratura scientifica nel corso del tempo, permettendo al lettore di familiarizzare gradualmente con le sfide e le soluzioni legate a questo problema classico di concorrenza. Viene inoltre fornita una panoramica dello stato dell'arte, presentando le soluzioni più recenti, i loro punti di forza e le loro limitazioni maggiori.

Il contributo di questa tesi consiste nella metodologia proposta per l'implementazione di code FIFO bounded lock-free, sviluppata a partire dalle tecniche all'avanguardia impiegate nelle code unbounded lock-free. Le principali soluzioni presenti in letteratura si concentrano su code unbounded (non limitate nel consumo di memoria), poiché la gestione dinamica della memoria è essenziale per garantire buone prestazioni e affrontare le problematiche legate alla contesa tra thread. Tuttavia, applicando restrizioni specifiche alle tecniche allo stato dell'arte usate nelle implementazioni unbounded, è possibile realizzare implementazioni bounded, con un consumo di memoria limitato e predefinito, senza sacrificare significativamente le prestazioni. In particolare, i test condotti dimostrano che, in scenari favorevoli, le code bounded così ottenute offrono prestazioni comparabili a quelle delle alternative unbounded, mantenendo i vantaggi del paradigma lock-free ed offrendo garanzie sul consumo di memoria.

A supporto delle analisi teoriche, la tesi presenta una sezione dedicata ai benchmark condotti per valutare le prestazioni delle principali soluzioni proposte in letteratura per le code FIFO non bloccanti e per esaminare le implementazioni bounded proposte. I risultati sono stati confrontati con quelli ottenuti da altre soluzioni lock-free esistenti e da alternative basate su meccanismi di sincronizzazione tradizionali (mutex-based). Nei test sono state inoltre analizzate le prestazioni delle varie implementazioni utilizzando il pinning esplicito dei thread, offrendo una visione concreta delle performance ottenibili con il pinning rispetto alla gestione dell'allocazione dei thread effettuata dal Sistema Operativo. La piattaforma di calcolo utilizzata per i test, fornita dall'Università di Pisa, è un server con due CPU multicore NUMA, ognuna con 32 core fisici e 64 logici.

Dai risultati dei test condotti sul sistema di calcolo emerge chiaramente come l'adozione di tecniche non bloccanti nell'implementazione di strutture dati offra significativi vantaggi rispetto alla sincronizzazione tradizionale basata su mutex. Gli approcci lock-free, in particolare, migliorano le prestazioni complessive del sistema riducendo la contesa tra thread, minimizzando la latenza delle operazioni e aumentando la scalabilità su architetture multicore. Comunque, non tutte le implementazioni lock-free garantiscono la stessa efficienza: soluzioni naïve, pur offrendo i benefici dell'approccio non bloccante, risultano limitate in termini di scalabilità.

Infine, i test confrontano implementazioni lock-free "semplici" con soluzioni più avanzate, dotate di meccanismi sofisticati per minimizzare la contesa. I risultati mostrano come queste ottimizzazioni consentano di ottenere strutture dati significativamente più scalabili e performanti, sia in code unbounded che in code bounded, rispetto alle alternative più semplici, evidenziando l'importanza di strategie mirate nella progettazione di sistemi concorrenti.

Organizzazione della Tesi

La presente tesi è articolata in tre capitoli principali:

- *Background*: introduce le garanzie di progresso e le primitive atomiche essenziali per le code MPMC. Analizza problemi come il problema ABA e il memory order, offrendo una panoramica delle evoluzioni delle code FIFO fino allo stato dell'arte.
- *Progettazione di code MPMC Bounded*: esamina due implementazioni considerate lo stato dell'arte, evidenziandone vantaggi e limiti. Discute le sfide nella progettazione di code bounded e propone possibili soluzioni.
- *Esperimenti e Risultati*: descrive l'ambiente di test e le metriche di benchmark, proponendo un'analisi comparativa delle implementazioni considerate.

Capitolo 1

Background

Nella programmazione concorrente, la sincronizzazione tra thread è cruciale per garantire correttezza ed efficienza. Questo capitolo introduce i concetti fondamentali della sincronizzazione non bloccante, delle primitive atomiche e della gestione della memoria concorrente.
Dopo una breve introduzione sui sistemi multicore di tipo NUMA (Non-Uniform Memory Access), che rappresenta il modello server attualmente più diffuso, verranno definite le caratteristiche che identificano sistemi non bloccanti distinguendoli in categorie con differenti garanzie di progresso per i thread, e si porrà attenzione sulle principali primitive atomiche utilizzate negli algoritmi conocrrenti non bloccanti. Inoltre, saranno discussi aspetti critici come il problema ABA, il Memory Ordering e la linearizzazione che influenzano il comportamento delle operazioni concorrenti.
Infine, il capitolo esplorerà i modelli per l'implementazione di code Multi-Producer Multi-Consumer (MPMC) e le strategie per la gestione della memoria attraverso l'uso degli Hazard Pointers. L'obiettivo del capitolo è fornire una base solida per comprendere e progettare algoritmi concorrenti efficienti e scalabili.

1.1 Architetture Multicore NUMA

I sistemi di calcolo moderni affrontano sfide sempre più complesse nella gestione delle risorse per garantire efficienza, soprattutto dato l'aumento della velocità dei processori e l'intensificarsi dei carichi di lavoro. Tradizionalmente, l'architettura si basa sul modello *SMP* (Symmetric Multiprocessing), in cui uno o più processori condividono l'accesso alla memoria centrale tramite un *bus di comunicazione*. Questo modello è efficace per un numero limitato di processori, ma diventa inefficiente man mano che questo aumenta, causando congestioni dovute alla sincronizzazione del bus. Sistemi di questo tipo sono noti come *UMA* (Uniform Memory Access), poiché il tempo di accesso alla memoria è lo stesso per tutti i processori.

Per ottenere elevate prestazioni, è fondamentale ridurre il carico sul bus di sistema e sull'accesso alla memoria centrale. A tale scopo, vengono utilizzati diversi livelli di *memoria cache*, che riducono la distanza tra le CPU e la memoria centrale, permettendo accessi più rapidi e paralleli, dato che processori diversi possono avere memorie cache dedicati. Le cache si basano sul principio di *località dei dati*, per cui si osserva come i programmi tendano a fare riferimento ripetuto agli stessi dati o a quelli locati nelle vicinanze di quelli correnti. Queste memorie locali distribuite necessitano di algoritmi avanzati per gestire la memorizzazione dei dati più frequentemente utilizzati e per garantire la *coerenza* tra le cache dei diversi processori.

Sebbene l'uso delle cache migliori le prestazioni, tale meccanismo non è sufficiente a compensare l'aumento delle dimensioni della memoria e dei programmi. Inoltre, i sistemi operativi possono tendere a ridurre l'efficacia delle cache, in casi in cui lo scheduler posiziona il processo in core diversi (*core switching*) rendendo spesso le prestazioni sub-ottimali. Per risolvere questo problema, i sistemi a elevate prestazioni hanno iniziato a adottare l'architetture basata sul modello NUMA (*Non-Uniform Memory Access*), per cui gruppi di processori si distinguono in nodi locali e remoti, riducendo così la competizione per l'accesso alla memoria migliorando la scalabilità. Virtualmente la memoria rimane una risorsa condivisa e accessibile da ogni processore, ma risulta fisicamente partizionata e ubicata in modo che certe CPU abbiano tempi di acceso maggiori rispetto ad altre.

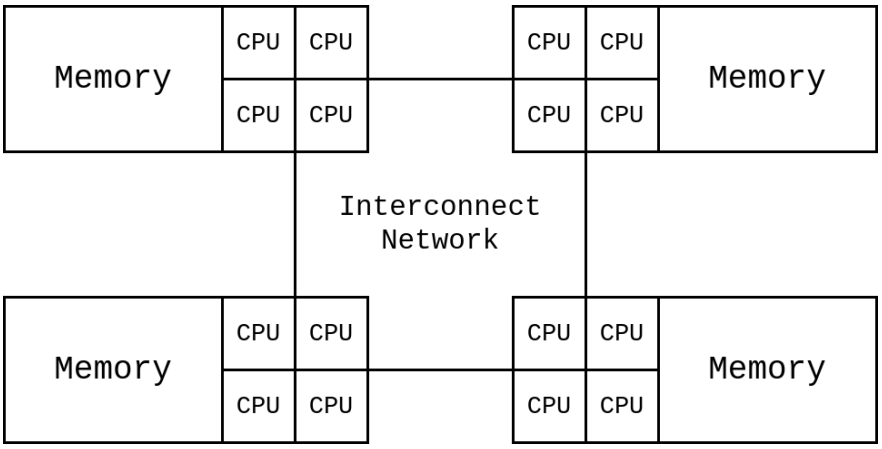


Figura 1.1: Struttura Generale di un'architettura NUMA

Lo spazio di indirizzamento totale è quindi condiviso ma un processore impiega meno tempo ad accedere a una porzione di memoria (*locale*) rispetto a un'altra (*remota*). Questo distribuise il traffico di comunicazione tra CPU e memoria generando minore contesa sui bus, permettendo ai sistemi di ospitare un numero di core maggiore rispetto a sistemi SMP UMA. Le architetture NUMA-based si dividono in due macrocategorie:

- **Non-Caching NUMA (NC-NUMA)**: sistemi specifici che non prevedono l'utilizzo di memorie cache. Questo rende l'accesso alla memoria remota molto inefficiente.
- **Cache-Coherent NUMA (cc-NUMA)** sistemi multi-purpose che implementano gerarchie di memorie cache per mitigare la congestione negli accessi alle memorie. Queste aumentano le prestazioni anche se introducono il problema della *cache invalidation* che necessita di protocolli specificatamente costruiti per garantire la scalabilità su architetture di questo tipo.

L'accesso alla memoria remota viene eseguito utilizzando un *interconnect network* (i.e. *AMD Infinity Fabric*, *Intel QuickPath*). Quando si parla di accessi a memoria remota si utilizza il termine *hop* poichè gli accessi passano attraverso la rete di interconnessione. Ogni coppia di *nodi NUMA* (insieme di CPU che accedono alla stessa memoria locale), può possedere un *interconnect fabric* che permette hop per richieste di accesso. La larghezza di banda di diversi *interconnect fabric* può essere diversa quindi un hop tra due coppie di nodi può avere costo diverso. La Figura 1.2 mostra come ogni Nodo NUMA gestisce accessi remoti e locali. Come si può vedere, ogni nodo è composto da più cores. Supponiamo esistano livelli di cache L1 e L2 nonostante non siano rappresentati. Ogni nodo contiene una cache L3 (LLC) condivisa tra i core che costituiscono il nodo. Le parti che coordinano l'accesso alla memoria sono la *Global Queue* (GC), l'*Integrated Memory Controller* (IMC) e il *QuickPath Interconnect* (QPI). Come specificato dal nome, l'IMC agisce

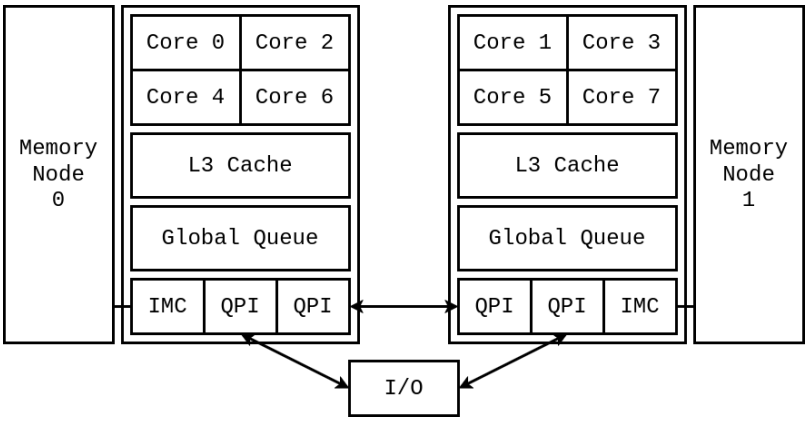


Figura 1.2: NUMA Node: Struttura di elaborazione interna

come un tipico memory controller, la differenza sta nel fatto che riceve le richieste di accesso attraverso la GQ invece dai cores. Le porte dell'IMC sono collegate alla GQ e alla memoria fisica.

Il QPI viene utilizzato come percorso per l'accesso remoto. Le porte di QPI sono collegate alla GQ del nodo a cui si accede; il resto delle porte sono connesse alla parte che richiede l'accesso. La ragione per cui si parla di "parte che richiede l'accesso" è che, come espresso in Figura 1.2, le porte potrebbero essere connesse a un altro nodo oppure a un dispositivo I/O. La Global Queue implementa un buffer di richieste da più sottosistemi (IMC,QPI,Cores). Questa ridireziona ogni richiesta di accesso verso il memory controller.

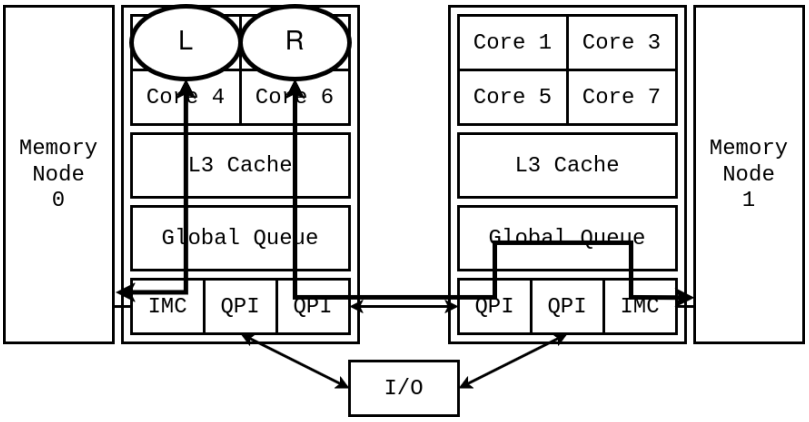


Figura 1.3: NUMA: accesso locale vs accesso remoto

La Figura 1.3 mostra la differenza in termini di passi di elaborazione tra accessi locali e remoti in un'architettura NUMA. La pagina L rappresenta una richiesta di accesso locale inviata da un core alla GQ (supponendo *cache miss*). All'interno della GQ, la richiesta viene inoltrata all'IMC, il quale si occupa della gestione dell'accesso alla memoria fisica e della trasmissione delle risposte.

Tuttavia, se l'accesso è remoto, il processo diventa oneroso. La pagina R descrive l'accesso remoto, in cui la richiesta viene inviata attraverso la GQ del nodo quindi verso il capo QPI sorgente, che è connesso al nodo corrispondente all'*entry point* per la memoria remota. Una volta che la richiesta raggiunge il capo QPI destinazione, essa viene inviata alla sua GQ e, da questo punto in poi, il processo diventa simile all'accesso locale, passando attraverso IMC e accedendo alla memoria fisica. Infine la risposta segue il percorso inverso per essere trasmessa.

Come si può osservare da questa struttura, rispetto all'accesso locale, l'accesso remoto aggiunge ulteriori fasi di elaborazione, comportando un overhead relativo. Tale effetto può essere mitigato se il dato richiesto è già caricato nella cache di ultimo livello del nodo remoto, permettendone la restituzione immediata. Ciò risulta particolarmente utile nei casi in cui l'accesso a un dato richiede multipli hop.

1.1.1 Tecniche Numa-Aware

Dato il modello hardware NUMA, la gestione delle risorse computazionali presenta delle sfide. Per massimizzare le prestazioni e ridurre la latenza, è fondamentale adottare strategie che considerino la struttura del sistema, in particolare l'allocazione della memoria. L'obiettivo è ridurre gli accessi remoti, posizionando i dati nei nodi NUMA più vicini ai thread che li utilizzano. La strategia *first-touch* prevede che la memoria venga allocata nel nodo in cui il thread la utilizza per la prima volta, riducendo i costi degli accessi *cross-nodes*. Allocatori come `jemalloc` e `tcmalloc` possono ridurre la frequenza di accessi remoti, mentre strumenti come `numactl` o API come `mbind()` consentono un controllo più preciso, vincolando le allocazioni a nodi specifici.

Lo scheduling dei processi è cruciale nell'efficienza di un sistema NUMA, specialmente quando operano su dati condivisi. Il *core-switch* di un thread può causare *cache thrashing*, aumentando la latenza. Per minimizzare questo effetto, si usa la tecnica del *thread-pinning*, vincolando un thread a un set specifico di core. La sincronizzazione tra thread di nodi diversi può introdurre overhead significativi, dovuti alla necessità di garantire la coerenza dei dati su memorie distribuite. Per mitigare il problema, si utilizzano tecniche come il *lock-cohorting* [3], che permette di creare primitive di sincronizzazione NUMA-aware, e il *cluster segmentation*, che riduce il traffico di invalidazione della cache segmentando una computazione condivisa in slot temporali che vengono eseguiti in numa nodes distinti.

1.2 Progresso in strutture dati concorrenti

Una *struttura dati condivisa* può essere definita come un insieme di locazioni di memoria a cui accedono, in lettura e scrittura, più *processi* mediante un insieme definito di *operazioni*. Questa espone dei metodi predefiniti che agiscono come unici *entry-points* per richieste specifiche. Generalmente si utilizzano primitive di mutua esclusione (*mutex*) per garantire che un insieme di operazioni appaia come eseguito in modo atomico e indivisibile, preservando così la consistenza della struttura dati in caso di accessi concorrenti. Sebbene tali primitive siano molto semplici da utilizzare, presentano molte criticità, tra queste, il fatto di fornire *garanzie di progresso* deboli. In particolare, può verificarsi il caso in cui tutti i processi rimangano bloccati indefinitamente (*stallo*) in attesa dell'acquisizione di una mutex che, per via di un bug o di una condizione non prevista, potrebbe non venir mai rilasciata. Per questo, generalmente, le mutex vengono acquisite in un qualche *ordine globale* che può prevenire l'attesa circolare ma portare a fenomeni di *inversione di priorità* che si si verifica quando un task a priorità più alta viene bloccato da uno a priorità più bassa, che detiene una risorsa condivisa (in questo caso la mutex), impedendo così l'esecuzione efficiente del sistema. Per affrontare questi problemi, sono stati sviluppati approcci basati su algoritmi *non bloccanti*, che offrono garanzie di progresso più forti escludendo l'utilizzo di meccanismi di mutua esclusione. Tutti gli algoritmi non bloccanti descritti in letteratura garantiscono che un processo in stallo non possa causare lo stallo di nessun altro processo [4]. Generalmente, per garantire proprietà di progresso più forti si introduce un costo in termini di prestazioni del sistema concorrente che si sta sviluppando. Tuttavia, per alcune specifiche strutture dati concorrenti, come ad esempio le code di tipo MPMC, è possibile ottenere contemporaneamente maggiori garanzie di progresso e migliori prestazioni rispetto all'approccio implementativo classico basato su mutex. Nel seguito, vengono approfondite le proprietà di progresso non-bloccanti di tipo *Lock Freedom*, *Wait Freedom* ed *Obstruction Freedom*. Verranno esaminate le caratteristiche distintive di ciascuna proprietà, evidenziando come esse garantiscano il completamento delle operazioni in ambienti concorrenti.

Lock Freedom

La proprietà di *lock freedom* [4] è una delle più rilevanti tra le proprietà non bloccanti. Un algoritmo o una struttura dati è definito lock-free se, in ogni esecuzione concorrente, è garantito che *almeno uno* dei processi riesca a completare la propria operazione in un numero finito di passi. In altre termini, il progresso lock freedom implica che il sistema non possa mai trovarsi in una condizione di stallo globale pur non assicurando che ogni singolo processo termini la propria operazione in tempi deterministici. Benché un processo possa essere soggetto a ripetuti ritardi o interruzioni a causa della contesa con altri processi, il progresso complessivo del sistema è sempre garantito. Rispetto ad altre proprietà non bloccanti, il lock-

freedom rappresenta un buon compromesso tra garanzie di progresso e efficienza computazionale.

Wait Freedom

Seppure il lock-freedom offra garanzie sul progresso complessivo del sistema, non assicura che ogni operazione sulla struttura condivisa termini in tempo finito. In scenari di alta contesa o per interleaving[1] sfavorevoli, un processo potrebbe fallire spesso o posticipare operazioni a causa della contesa con altri processi. La proprietà di *wait-freedom* [4] offre una garanzia più forte, assicurando che ogni operazione completi dopo un numero finito di passi, prevenendo oltre al *livelock* la *starvation*. Ciò implica che sia possibile, empiricamente, stimare un tempo di esecuzione nel caso pessimo per ogni operazione. Formalmente, un algoritmo o una struttura dati è detto wait-free se garantisce che ogni operazione, indipendentemente da interferenze e dalla contesa con altri processi, termini in un numero finito di passi. Ogni thread o processo ottiene la certezza di completare la propria operazione, prevenendo qualsiasi forma di starvation o stallo. La progettazione di algoritmi wait-free è complessa, in parte a causa dell'assenza di garanzie di accesso equo alla memoria da parte dell'architettura hardware. Solitamente, è necessaria una sincronizzazione software più avanzata per evitare la starvation che molto spesso viene ottenuta tramite metodologie *fast-path/slow-path* [5], in cui ogni operazione viene "annunciata" in una locazione di memoria ad accesso in scrittura esclusivo (single-writer). Le operazioni vengono eseguite tramite metodi cosidetti rapidi (fast-path), che ammettono starvation. I processi che effettuano operazioni con successo, scansionano periodicamente gli annunci, "aiutando" altri processi tramite metodi cosidetti lenti (slow-path), che richiedono sincronizzazione parziale o totale del sistema.

Le garanzie di progresso wait-free sono particolarmente rilevanti nella realizzazione di sistemi real-time (come ad esempio i sistemi di controllo di navigazione e di gestione del volo negli aeroplani) che devono garantire tempi di risposta massimi prestabiliti in fase di progettazione. Tali garanzie sono essenziali per rispettare requisiti di latenza molto stringenti per poter garantire la sicurezza, e richiedono il completamento di ogni operazione in un numero finito di passi.

Obstruction Freedom

La proprietà di progresso *obstruction-freedom* [6] punta a garantire molti aspetti positivi del lock-freedom riducendone la complessità nel design di strutture dati. Poiché garantire efficienza mentre i processi si aiutano reciprocamente a fare progressi è una delle principali sorgenti di complessità in molti algoritmi lock-free e wait-free (dato che aiuto eccessivo può generare ulteriore contesa in accessi alla

[1] fenomeno con cui le operazioni di diversi thread vengono eseguite in modo alternato e sovrapposto nel tempo

memoria), l'obstruction-freedom permette di ridurre l'overhead consentendo che un'operazione soggetta a contesa venga annullata e riprovata successivamente.

Un algoritmo o una struttura dati è detto obstruction-free se, quando un thread o processo esegue un'operazione in isolamento (ossia in assenza di interferenze concorrenti per un numero sufficiente di passi), l'operazione termina in un numero finito di passi. In sostanza, ogni thread garantisce il completamento della propria operazione in condizioni di non contesa, non offrendo garanzie di progresso in presenza di contesa. Sebbene l'obstruction-freedom sia una proprietà abbastanza forte da prevenire effetti come il deadlock o l'inversione di priorità, sono necessari meccanismi separati per gestire il livelock (che potrebbe essere causato da due operazioni che generano contesa reciproca che si annullano continuamente a vicenda). Per evitare il livelock negli algoritmi obstruction-free vengono spesso utilizzati metodi di *backoff* per cui un'operazione che fallisce, viene riprovata dopo un certo periodo di tempo che aumenta al crescere di fallimenti consecutivi. Nonostante questo approccio possa garantire buone prestazioni, è estremamente complesso dimostrare (considerando anche fattori di variabilità introdotti dallo scheduler) che un determinato valore di backoff sia il migliore per una data applicazione.

1.3 Operazioni Atomiche e Sincronizzazione

Le primitive atomiche sono operazioni che garantiscono l'esecuzione indivisibile di una o più istruzioni, ovvero vengono completate senza possibilità di interruzioni da parte di altre operazioni concorrenti. Tali operazioni, fondamentali in contesti di esecuzione concorrente, assicurando che l'accesso simultaneo a strutture dati condivise non causi errori o incoerenze. Queste primitive, supportate direttamente dall'hardware o dal sistema operativo e sono essenziali per l'implementazione di algoritmi di sincronizzazione non bloccante, in particolare in scenari che richiedono alta efficienza e scalabilità.

In contesti ad alta concorrenza, dove molteplici processi possono accedere simultaneamente alle stesse risorse, si presentano diverse problematiche legate alla gestione dei dati condivisi. In particolare, l'uso di operazioni atomiche non garantisce sempre un controllo completo sugli effetti collaterali che possono derivare da modifiche concorrenti. Ciò richiede degli accorgimenti aggiuntivi per mantenere l'integrità dei dati in ambienti di esecuzione parallela.

Compare-And-Swap (CAS)

L'istruzione atomica *Compare-And-Swap* (`CAS`), nota come `CMPXCHG` in architetture `Intel x86` [7], è una delle operazioni atomiche fondamentali in molti algoritmi non bloccanti. `CAS` confronta il valore attuale di una locazione di memoria con un valore atteso e in caso di uguaglianza, lo scambia con un nuovo valore. Se

il confronto non ha successo allora il valore attuale non viene aggiornato e generalmente viene restituito il valore attuale della locazione di memoria. Questo meccanismo atomico di aggiornamento condizionale permette di evitare inconsistenze e di costruire meccanismi di sincronizzazione che spesso permettono di sostituire le classiche primitive di mutua esclusione.
Molti algoritmi non bloccanti si basano sul meccanismo denominato `CAS`-*loop* per cui viene ritentata l'operazione fino a quando questa non ha successo. Sebbene molto efficace, sono comuni i fenomeni cosiddetti di *CAS-hotspot* in cui un gruppo elevato di processi esegue operazioni `CAS` su un numero molto ridotto di locazioni di memoria, generando contesa. Per mitigare questo problema, è spesso adottata

```
1 /* global backoff constansts */
2 const int MIN_WAIT = 128;
3 const int MAX_WAIT = 1024;
4
5 void backoff(int iter){
6     while(iter -- > 0); /* active waiting */
7 }
8
9 void CAS_loop(word_t * ctrl_val, word_t new_val){
10     int wait = MIN_WAIT;
11     do{
12         word_t expected = LOAD(ctrl_val);
13         if(CAS(ctrl_val,expected,new_val))
14             break;
15
16         /* exponential backoff */
17         backoff(wait);
18         wait *= 2;
19         if(wait > MAX_WAIT) wait = MAX_WAIT;
20     } while(true)
21 }
```

Figura 1.4: Pseudocodice CAS-loop

una strategia di *backoff*. In pratica, quando un'operazione CAS fallisce, il thread attende per un intervallo di tempo variabile (solitamente incrementato in modo esponenziale a seguito di ogni fallimento fino ad un valore soglia) prima di riprovare. Questo approccio aiuta a ridurre la contesa, poiché diminuisce la probabilità che più thread tentino contemporaneamente di aggiornare la stessa locazione di memoria.

Fetch-And-Add (F&A)

La primitiva *Fetch-And-Add* (`F&A`) è un caso specifico di una classe di primitive che eseguono delle operazioni binarie in modo atomico. Altre primitive di questo tipo (i.e *Fetch-Sub*, *Fetch-Or* ...) sono comuni nella maggior parte delle architetture. L'utilizzo più comune di `F&A` è l'implementazione di contatori atomici condivisi che non richiedono meccanismi di locking esterno.
Come nel caso di `CAS` anche `F&A` è predisposta a fenomeni di *hotspot* sebbene

nella maggior parte dei casi risultino più contenuti di CAS dato che l'operazione ha sempre successo.

```
1 struct collection {
2     atomic_int insert_idx;
3     atomic_int extract_idx;
4     word_t ** buffer;
5     int length;
6 }
7
8 const word_t empty; // reserved value for empty
9
10 bool insert(collection *c, word_t * data){
11     while(true){
12         int ticket = F&A(&(c->insert_idx), 1);
13         int cell_idx = ticket % (c->length);
14         word_t *cell_val = LOAD(&(c->buffer[cell_idx]));
15
16         if(cell_val == empty){
17             if(CAS(&(c->buffer[cell_idx],empty,data)))
18                 return true;
19         }
20         int extract_idx = LOAD(&(c->extract_idx));
21
22         //full collection
23         if(ticket > (extract_idx + c->length))
24             return false;
25     }
26 }
```

Figura 1.5: Inserimento F&A-loop

Nel design di strutture dati concorrenti l'operazione atomica F&A viene spesso utilizzata in un loop (denominato F&A-loop) per minimizzare la contesa in un singolo punto di aggiornamento di una struttura dati concorrente. La Figura 1.5 mostra un tipico F&A-loop in cui si suppone si debba aggiornare una struttura dati astratta, con operazioni che, per semplicità, non rispettano la semantica FIFO. L'inserimento utilizza un indice condiviso, acceduto ed incrementato tramite l'operazione F&A, che, tramite un accesso in modulo, mappa ad una cella di un buffer circolare. In questo modo, si aumenta la probabilità che operazioni concorrenti accedano a slot diversi, riducendo la contesa della più costosa operazione di CAS utilizzata per l'installazione dell'elemento.

Viene omesso lo pseudocodice per l'operazione di estrazione, che è simmetrico rispetto all'inserimento, ma richiede un controllo aggiuntivo per mantenere la coerenza tra gli indici.

Test-And-Set (TAS)

La primitiva *Test-And-Set*(T&S) è un'operazione atomica fondamentale, usata per gestire dati a livello di bit in modo efficiente. Data una locazione di memoria L_i, T&S legge il valore presente nella locazione di memoria e lo imposta a `true` in un'unica operazione indivisibile. In pratica, se il valore iniziale in L_i era `false`,

la primitiva restituisce `false` aggiornando immediatamente il contenuto a `true`; se invece L_i era già `true`, restituisce `true`, indicando che la risorsa è già stata impostata. Questa primitiva viene molto usata nell'implementazione di *spinlock*: meccanismi di sincronizzazione ad attesa attiva dove una sezione critica è protetta da un *retry loop* che testa una condizione tramite `T&S`. Quando questa restituisce `false` setta una variabile condizionale a `true` logicamente bloccando l'accesso alla sezione critica fino a un'operazione di reset.

Multiword-CAS

La primitiva `CAS` è abbastanza espressiva e ampiamente utilizzata nel design di algoritmi non bloccanti. In certi casi, può però risultare limitata dato che garantisce aggiornamenti atomici su una singola parola di memoria. Questo diventa un limite quando si ha la necessità di aggiornare in modo atomico più locazioni di memoria contemporaneamente. E' opinione comune che l'implementazione di strutture dati non bloccanti è complicata dalla mancanza di primitive che permettano di trattare casi specifici.

Consideriamo quindi un'estensione astratta a cui ci si riferisce con l'appellativo di *Multiword-CAS* (`MCAS`) che estende le funzionalità della versione a singola parola a un numero arbitrario di locazioni di memoria. Più precisamente, `MCAS` opera su n locazioni di memoria distinte (L_i), valori attesi E_i, nuovi valori N_i: ogni L_i è aggiornato al nuovo valore N_i se solo se *ogni* L_i contiene E_i ($\forall i \in [0, n]$) prima dell'invocazione dell'operazione.

```
1 atomic bool MCAS(word_t *loc[], word_t exp[], word_t new[], int n){
2     //Verifica valori attesi
3     for(int i = 0; i < n; i++){
4         if(*(loc[i]) != exp[i])
5             return false;
6     }
7     //Swap dei valori
8     for(int i = 0; i < n; i++)
9         *(loc[i]) = new[i];
10
11    return true;
12 }
```

Figura 1.6: Pseudocodice MCAS

Implementare una primitiva di questo tipo risulta essere molto complesso a livello hardware tanto che, a quanto risulta, nessuna architettura supporta questa operazione nativamente. Esistono comunque dei metodi che fanno uso di operazioni atomiche ampiamente supportate per l'implementazione di `MCAS` [8] ma il loro utilizzo è molto limitato a causa di fattori di contesa.

Nonostante la scarsa disponibilità dell'operazione `MCAS`, la maggior parte degli algoritmi non bloccanti non richede che l'operazione venga eseguita su numero arbitrario di locazioni di memoria: in prevalenza si vorrebbe aggiornare ato-

micamente solo due parole di memoria contigue (i.e un valore di controllo e un descrittore di dato associato).

Questa operazione risulta essere molto meno complicata ed è supportata dalle architetture `x86-64` (`CMPXCHG16B`) a cui ci si riferisce generalmente con il termine Double Compare-And-Swap (`DCAS`). Purtroppo, il supporto hardware per `DCAS` è limitato e alcuni processori moderni (i.e. `ARM`, `PowerPC`) non implementano questa operazione.

`CAS2` è un termine usato in letteratura per indicare una versione estesa della `CAS` che opera su due dati astratti non necessariamente contigui in memoria. Quando non è disponibile un'istruzione hardware `DCAS`, `CAS2` viene spesso implementata a software per emulare l'operazione atomica su due parole. Questa implementazione software può essere più complessa e, in certi casi, non supportare le stesse garanzie di progresso della corrispettiva primitiva hardware. Da questo punto in poi, il termine `CAS2`,`DCAS` sono interscambiabili.

Emulazione `CAS2` tramite `CAS`

Al fine di interagire con le routine hardware-specifiche in linguaggi ad alto livello (i.e `C++`), generalmente è necessario ricorrere all'integrazione di codice assembly, poiché le operazioni di basso livello, intrinsecamente legate alle peculiarità dell'architettura del processore, non sono direttamente esprimibili attraverso le astrazioni offerte dal linguaggio. Seppure tale funzionalità rappresenti una risorsa imprescindibile per il conseguimento di certi obbiettivi prestazionali e per l'accesso diretto a capacità hardware avanzate, essa risulta incompatibile con esigenze di portabilità. L'introduzione di routine in linguaggio assembly, infatti, vincola il software alla specifica architettura di macchina, rendendo difficile il suo corretto funzionamento su altre piattaforme. Risulta quindi vantaggioso emulare funzionalità normalmente accessibili solo tramite accesso diretto all'hardware, utilizzando costrutti del linguaggio di alto livello.

Si consideri un'operazione `CAS2` che agisce su due locazioni di memoria distinte, L_1 e L_2 (non necessariamente contigue), confrontando i valori E_1 e E_2 con il contenuto corrente di queste locazioni. L'aggiornamento ai valori N_1 e N_2 avviene se solo se entrambi i confronti hanno esito positivo. Per ottenere tale comportamento, si supponga che esista un valore `busy` non utilizzabile in operazioni di assegnamento o confronto (siano esse atomiche o meno). Questo valore viene utilizzato per bloccare temporaneamente una delle due locazioni di memoria, mentre viene eseguita un'operazione `CAS` a singola parola sulla seconda locazione. Indipendentemente dall'esito dell'operazione, sia esso positivo o negativo, un semplice assegnamento atomico (`STORE`) viene utilizzato per aggiornare il contenuto della locazione bloccata, assegnando il nuovo valore o ripristinando quello precedente. Il procedimento è illustrato in Figura 1.7.

La corretta esecuzione dell'operazione è assicurata dall'utilizzo di `busy` che garantisce assenza di interferenze su una delle due locazioni L_i fino al completa-

mento del metodo, tutelando il principio di atomicità.

Vengono proposte varianti[9] di questo metodo, che si adattano a semantiche algoritmiche specifiche. E' importante sottolineare che il codice rappresentato

```
1
2 const word_t reserved; //valore riservato
3
4 bool CAS2(  word_t*l_1, word_t*l_2,
5             word_t e_1, word_t e_2,
6             word_t n_1, word_t n_2
7          )
8 {
9     //blocco prima locazione
10    if(CAS(l_1, e_1,reserved) {
11        if(CAS(l_2, e_2, n_2)) {
12            //assegnamento atomico del nuovo valore
13            STORE(l_1, n_1);
14            return true;
15        } else {
16            //ripristino del valore atteso
17            STORE(l_1, e_1);
18        }
19    }
20    return false;
21 }
```

Figura 1.7: Pseudocodice che emula l'operazione CAS2.

in Figura 1.7 assume un sistema con processi/thread non interrumpilibli. L'interruzione di un processo che non riesce a completare l'operazione potrebbe determinare lo stallo del sistema, dovuto al fatto che altri processi potrebbero non proseguire su una cella che rimane riservata. In sistemi reali questo sistema può essere adattato utilizzando delle operazioni di `CAS` sostituendo le operazioni di `STORE` e implementando un sistema di recupero per le celle riservate che vengono riportate a un valore precedente. Questo non può essere applicato in tutti i casi: infatti se il valore della cella riservata non era un valore predefinito (i.e. `nullptr`) allora l'unico processo che mantiene memoria del valore precedente è quello interrotto e non c'è modo di ripristinare un dato coerente.

1.4 Il Problema ABA

Il problema ABA[10] è un fenomeno che può verificarsi nei sistemi concorrenti, specialmente in quelli che utilizzano operazioni atomiche per la gestione di memoria condivisa. In un sistema con più processi, ogni processo può leggere e scrivere valori in locazioni di memoria condivise, ma il verificarsi di interruzioni o di altre operazioni concorrenti può causare comportamenti imprevisti.

Il problema ABA si verifica quando un processo legge un valore da una locazione di memoria, viene interrotto ed un altro processo modifica quel valore. Se successivamente il valore viene ripristinato a quello precedente, il primo processo che riprende l'esecuzione non ha avuto modo di rilevare il cambiamento momen-

taneo, il che potrebbe portare a un errore logico nel sistema. Questo fenomeno è pericoloso specialmente quando si utilizzano `CAS` per effettuare aggiornamenti di locazioni condivise. Se un processo non rileva che il valore è stato modificato, può applicare l'operazione `CAS` con successo (falso positivo), anche se la locazione di memoria è stata, nel frattempo, alterata e ripristinata.

Il seguente esempio, riportato nella Tabella 1.1, illustra il problema ABA in modo dettagliato.

Step	*Azioni*
Step 1	P_1 legge A_i da L_i; P_1 viene interrotto
Step 2	P_2 scrive B_i su L_i
Step 3	P_2 scrive A_i su L_i; P_2 viene interrotto
Step 4	P_1 effettua CAS(L_i, A_i, X_i) con successo

Tabella 1.1: ABA su locazione L_i

Naturalmente, il problema ABA è di natura logica, quindi è possibile che alcuni algoritmi siano *ABA-free*, nel caso in cui il verificarsi di questo non influenzi la semantica operazionale. Questo caso è poco comune, in algoritmi lock-free.

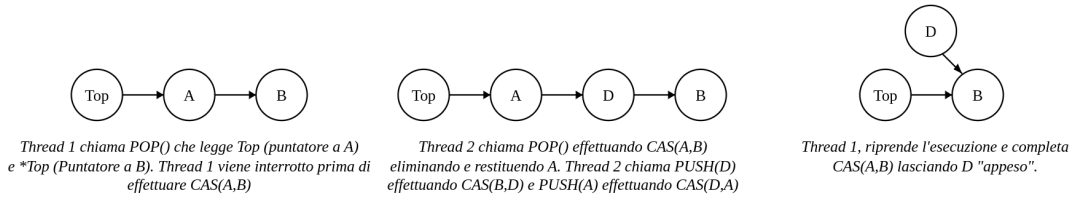


Figura 1.8: Problema ABA su lista concatenata.

Si illustra il problema ABA in Figura 1.8 considerando l'esempio di una lista concatenata con operazioni a semantica LIFO[2]. Una strategia generale per evitare il problema ABA si basa sulla garanzia fondamentale che nessun processo P_j ($P_j \neq P_1$) possa mai riscrivere A_i a una data locazione L_i (Step 3, Tabella 1.1). Un modo semplice per garantire questa condizione è richiedere che tutti i valori memorizzati in una determinata locazione siano univoci, ad esempio impedendo il riutilizzo di un dato valore di controllo.

In realtà, per la maggior parte delle strutture dati, un tale vincolo sarebbe molto limitante: strutture dati come pile o code dovrebbero supportare l'inserimento

[2] *Last-In-First-Out*: semantica operazionale per cui il primo elemento estratto dalla struttura dati è l'ultimo inserito

dello stesso valore più volte. Un'altra tecnica è invece quella di utilizzare dei *version tag/reference counters* per tenere traccia di ogni accesso eseguito alla stessa locazione di memoria. Questa è in generale la soluzione più adottata che però soffre di un difetto significativo: l'operazione di `CAS` non è sufficiente per garantire l'aggiornamento atomico di un valore di controllo e di un varsion tag (assumendo ognuno di dimensione di una parola di memoria). Un'applicazione efficace di questo sistema richiederebbe quindi primitive più complesse in grado di aggiornare atomicamente più parole di memoria (i.e. `CAS2/DCAS`) [10]. Primitive di questo tipo esistono in certe architetture ma il loro utilizzo implica ridotta portabilità su architetture diverse. Alcuni approcci [11] suggeriscono di ridurre la precisione del del valore di controllo e il *version tag* in modo da memorizzarli in una sola parola di memoria. Questo mina significativamente i tipi di dati strutturati utilizzabili (i.e puntatori hanno dimensione di una parola di memoria).

1.5 Memory Ordering

L'esecuzione delle istruzioni nei processori segue un ciclo macchina standard, che include le fasi di recupero, decodifica ed esecuzione dell'istruzione[3]. Tradizionalmente, queste fasi vengono eseguite in sequenza per ogni singola istruzione. Tuttavia, i processori moderni adottano un'architettura a modello *pipeline*, suddividendo il ciclo macchina in stadi distinti, che operano simultaneamente. Questo approccio consente di sovrapporre le fasi di esecuzione di istruzioni differenti, migliorando significativamente l'efficienza e riducendo la latenza di esecuzione di istruzioni successive.

Il parallelismo introdotto dall'architettura pipeline, tuttavia, comporta delle sfide come la gestione delle dipendenze tra istruzioni[4]. Un esempio specifico di dipendenza di dati è la cosiddetta *Read-After-Write* (RAW), in cui un'istruzione dipende dal risultato di una precedente che non ha ancora completato la sua esecuzione. Questo tipo di dipendenza diminuisce il parallelismo poiché obbliga il processore ad attendere il completamento dell'istruzione precedente. Per mitigare l'effetto di queste dipendenze, i processori moderni utilizzano tecniche di *out-of-order execution*, che permettono di eseguire le istruzioni non seguendo strettamente l'ordine di emissione, ma adattandosi dinamicamente alla disponibilità delle risorse e dei dati necessari[12]. Questo approccio ottimizza l'utilizzo delle risorse hardware disponibili e riduce i periodi di inattività. L'esecuzione fuori ordine e la bufferizzazione hardware di letture e scritture rende necessario l'introduzione di un preciso modello di *memory ordering* che definisca in maniera rigorosa l'ordine con cui le operazioni di memoria diventano globalmente visibili, garantendo così la coerenza dei dati e prevenendo situazioni di corse critiche (*race conditions*).

[3] Ciclo macchina o ciclo Fetch-Decode-Execute

[4] Generalmente si dividono in macrocategorie: Address Dependancies, Control Dependancies, Data Dependancies

Architetture diverse possono implementare differenti memory orders [13]. Ognuno di questi modelli definisce specifiche garanzie sull'ordine con cui vengono eseguite le operazioni di lettura e scrittura sulla memoria, offrendo quindi differenti livelli di coerenza e vincoli sulle operazioni eseguite concorrentemente.

Total Store Order (TSO)

Il *Total Store Order*[14] è uno dei modelli di memoria più utilizzati nelle architetture moderne. Esso impone un ordinamento stretto e sequenziale delle operazioni di scrittura, garantendo che tutte le scritture siano rese visibili nello stesso ordine in cui sono state effettuate. Al contrario, le operazioni di lettura possono essere riordinate liberamente rispetto alle scritture precedenti, consentendo al processore una certa flessibilità nell'esecuzione fuori ordine. Il modello TSO è largamente adottato nelle architetture basate su `x86`, dove costituisce il modello di riferimento per la coerenza della memoria.

Weak Memory Order (WMO)

Il *Weak Memory Order*, diversamente del TSO, offre maggiore libertà nel riordinamento delle operazioni di memoria. In questo modello, le operazioni di lettura e scrittura non vengono necessariamente eseguite nell'ordine in cui sono emesse dal processore, ma sono comunque soggette a regole specifiche che garantiscono la coerenza tra processi concorrenti. Il WMO è utilizzato su numerose architetture moderne, come ad esempio quelle basate su ARM, le quali ottimizzano la velocità di esecuzione proprio attraverso un approccio più flessibile nella gestione degli accessi in memoria. Tuttavia, questa flessibilità implica una maggiore complessità nello sviluppo del software concorrente, poiché rende necessaria una gestione attenta e precisa della sincronizzazione e dell'utilizzo di istruzioni specifiche (i.e. *memory fences*) per garantire la corretta visibilità e coerenza dei dati condivisi.

Sequential Consistency (SC)

Il modello di memoria *Sequential Consistency*, introdotto da Lamport[15], è considerato il più semplice e intuitivo tra i modelli di memory ordering. Questo modello, impone che tutte le operazioni di memoria siano eseguite in un ordine sequenziale globalmente visibile. In altre termini, tutte le operazioni di lettura e scrittura devono apparire eseguite nello stesso ordine in cui compaiono nel programma, senza alcuna possibilità di riordinamento. Sebbene questo modello fornisca forti garanzie di coerenza e faciliti la comprensione e l'analisi del comportamento dei programmi concorrenti, limita le opportunità di ottimizzazione delle prestazioni tipiche dei modelli di memory ordering più deboli, risultando quindi meno efficiente dal punto di vista computazionale.

Modello Architecture-Independent: C++

I diversi memory orders imposti dalle varie architetture offrono una sfida significativa per la programmazione di software concorrente portabile. La difficoltà principale risiede nel fatto che le tecniche di sincronizzazione e la gestione delle operazioni atomiche possono variare sensibilmente tra le diverse architetture.

Il linguaggio C++ (C++11 o superiori) affronta questa complessità fornendo un'astrazione a livello di linguaggio [16, 17] che definisce un modello di memoria coerente e indipendente dall'hardware sottostante, offrendo garanzie di coerenza e sincronizzazione uniformi, indipendentemente dal memory order adottato dall'architettura in cui il codice viene eseguito. In questo modo il memory order `C++` si pone tra la logica applicativa e il memory order hardware-specific, garantendo portabilità nello sviluppo.

Questo modello di memoria si basa sul principio per cui le operazioni atomiche devono apparire eseguite in un ordine sequenziale tra di loro, pur non garantendo lo stesso livello di ordinamento per le operazioni non atomiche. In altre parole, mentre le operazioni atomiche sono viste da tutti i thread come se fossero eseguite in un'unica sequenza globale, non esiste lo stesso vincolo per le operazioni ordinarie. La sincronizzazione tra processi o thread viene realizzata tramite costrutti come `std::atomic`, che mettono a disposizione metodi quali `load()`, `store()`, `compare_exchange_strong()` e `fetch_add()`.

Oltre a questi metodi di base, la libreria standard `C++` consente anche di definire esplicitamente il `memory_order` delle operazioni atomiche. I tipi principali di questa categoria sono:

- `memory_order_relaxed`: non impone alcuna garanzia sul riordinamento delle operazioni. Le operazioni atomiche possono essere riordinate liberamente, senza garantire alcuna visibilità immediata tra i thread. Questo tipo di ordinamento è utile in contesti dove le prestazioni sono prioritarie e la sincronizzazione stretta non è necessaria.

- `memory_order_acquire`: garantisce che tutte le operazioni di lettura e scrittura prima dell'operazione atomica siano visibili al thread che acquisice il valore. Un esempio è `load(...,memory_order_acquire)` che garantisce che tutte le scritture precedenti non possano essere riordinate dopo tale operazione.

- `memory_order_release`: garantisce che tutte le operazioni di lettura e scrittura effettuate dopo l'operazione siano visibili a tutti i thread che eseguono un'operazione con `memory_order_acquire`. Questo tipo di ordinamento è spesso utilizzato in scenari dove si vuole che un thread rilasci informazioni su un dato prima di terminare un'operazione.

- `memory_order_acq_rel`: combina le garanzie dei due modelli precedenti, assicurando che le operazioni precedenti e successive all'operazione atomica siano visibili.

- `memory_order_seq_cst`: il livello di ordinamento più rigoroso. Garantisce che tutte le operazioni atomiche siano visibili e globalmente ordinate in modo sequenziale, rispettando l'ordine di esecuzione come appare nel programma. Questo è il comportamento predefinito delle operazioni atomiche in C++.

1.6 Hazard Pointers

I *garbage collector* (GC) sono componenti fondamentali in molti linguaggi di programmazione, progettati per gestire automaticamente la memoria. Il loro obiettivo principale è quello di identificare e liberare regioni di memoria non più utilizzata, evitando così di compromettere l'efficienza di un programma. Alcuni esempi di garbage collector includono quelli utilizzati in linguaggi come *Java*, *C#* e *Go*, dove il GC opera in background per gestire la memoria dinamica. Le strategie di raccolta delle aree di memoria non utilizzate possono variare, ma la maggior parte si basa su tecniche come *mark-and-sweep*, *copying collections* e *reference counting* [18, 19].

Nei garbage collector, la sincronizzazione è necessaria per evitare corruzioni nei dati durante le operazioni di raccolta, soprattutto quando ci sono più thread in esecuzione.

Esistono alcuni modelli di GC che utilizzano tecniche lock-free per la gestione della concorrenza. Uno di questi è quello proposto da Herlihy[20] che si basa sul modello di GC generazionale. In questo approccio, la memoria viene suddivisa in diverse generazioni, partendo dall'ipotesi che le aree allocate più di recente abbiano una maggiore probabilità di essere eliminate. Il modello integra diverse strategie di *reclamation*, ovvero il recupero delle regioni di memoria non più utilizzate.

Sebbene i garbage collector siano costrutti molto comuni in linguaggi di alto livello, nei linguaggi a più basso livello come `C` e `C++`, la gestione della memoria è generalmente affidata al programmatore. In tali contesti, la deallocazione della memoria in ambiente concorrente è un'operazione delicata che necessita di accorgimenti per evitare problemi quali i *dangling pointers*, ossia puntatori che riferiscono zone di memoria già deallocate, con il rischio di generare accessi non validi e conseguenti comportamenti non prevedibili.

Gli *Hazard Pointers*, proposti da Michael [21], sono una tecnica comunemente utilizzata per gestire la memoria in ambienti concorrenti, in particolare in algoritmi lock-free. Sebbene i linguaggi come il `C++` offrano smart pointers per la gestione automatica della memoria, in scenari altamente concorrenti sono necessarie soluzioni più sofisticate. Un problema frequente in questi contesti è rappresentato dall'accesso concorrente che rende difficile determinare quale memoria può essere deallocata in sicurezza. Gli Hazard Pointers affrontano questo problema tramite un sistema di *reference marking*, che consente di identificare in modo esplicito le risorse di memoria ancora in uso, prevenendo deallocazioni premature e garantendo l'integrità della memoria.

Il sistema si basa su una struttura condivisa tra i diversi thread che accedono alla memoria. Ogni thread mantiene una propria lista di puntatori *hazard*, ovvero riferimenti a porzioni di memoria che il thread sta utilizzando attivamente. Quando un thread desidera operare su un puntatore, lo aggiunge alla propria lista mediante una scrittura atomica, segnalando così che quella specifica area non deve essere deallocata. Una volta completata l'operazione, il puntatore viene rimosso dalla lista, ad esempio aggiornandolo ad un valore nullo, segnalando che la memoria è nuovamente disponibile per il reclamation.

Quando un thread intende deallocare un'area di memoria, inserisce il puntatore nella *free list*, ovvero una lista speciale che raccoglie i riferimenti alle regioni di memoria da ritirare. Prima di procedere alla deallocazione effettiva, per ciascun puntatore presente nella lista, vengono esaminate tutte le liste hazard dei vari thread, al fine di assicurarsi che nessun altro stia ancora accedendo a quella zona di memoria. Solo se il puntatore non compare in nessuna lista hazard, la memoria può essere liberata in modo sicuro.

Nonostante l'implementazione proposta goda della proprietà di wait-freedom per le operazioni di protezione e lock-freedom per l'operazione di deallocazione, introduce sicuramente dell'overhead non trascurabile. La trasmissione di informazioni tra processi, soprattutto in ambienti non bloccanti, può determinare un degrado delle prestazioni dovuto allo scarso utilizzo delle memorie cache. Fenomeni come il *cache ping-pong*[5] e il *false-sharing*[6] devono essere tenuti in considerazione. Per contrastare tali problemi, le strutture dati utilizzano spesso tecniche di *padding*[7].

Oltre a questo, dato che il sistema mira ad abbattere i costi di sincronizzazione, non garantisce alcun memory bound: infatti un singolo thread bloccato può essere sufficiente per compromettere la deallocazione della memoria. Sono stati proposti sistemi più avanzati tra cui le *Hazard Eras* [22] proposto da Ramalhete. Integrando un sistema basato su epoch, le Hazard Eras riescono a limitare il problema e a offrire un memory bound anche in presenza di thread bloccati.

[5] situazione in cui i dati vengono continuamente trasferiti tra livelli di cache a causa di conflitti di accesso concorrente

[6] invalidazione continua di linee di cache contenenti parole condivise tra thread, che implica il ricaricamento dalla memoria dell'intera linea

[7] tecnica di allineamento dati in memoria per far si che i dati non confliggano sulla stessa linea di cache

1.7 Linearizzabilità in strutture dati non bloccanti

Il concetto di *linearizzabilità* rappresenta uno dei criteri fondamentali per assicurare correttezza negli algoritmi e strutture dati concorrenti.

Introdotta formalmente da Herlihy e Wing [23], la linearizzabilità definisce che ogni operazione concorrente deve apparire come se fosse stata eseguita istantaneamente, in un unico momento preciso compreso tra l'inizio (invocazione) e la fine (risposta) dell'operazione stessa. Questo punto esatto nel tempo prende il nome di *linearization point*. La proprietà di linearizzabilità consente di interpretare l'esecuzione parallela delle operazioni come se fosse un'esecuzione sequenziale, semplificando notevolmente la verifica della correttezza e la comprensione del comportamento del sistema concorrente.

In strutture dati *non bloccanti*, la linearizzabilità assume un'importanza fondamentale. Queste, non utilizzando meccanismi di mutua esclusione tradizionali (lock), si basano invece su istruzioni atomiche che l'hardware garantisce essere indivisibili e prive di interferenze. Tuttavia, la mancanza di lock rende ancora più critica la definizione precisa del *linearization point*, per assicurare che tutte le operazioni concorrenti abbiano effetti coerenti e visibili in modo sequenziale.

Un esempio tipico è dato dall'istruzione atomica `CAS` in cui il linearization point coincide esattamente con l'istante in cui questa operazione atomica aggiorna con successo il valore in memoria. Sebbene l'istruzione `CAS` possa richiedere internamente più cicli macchina per completarsi, dal punto di vista logico la modifica appare immediata e indivisibile agli altri thread. Un altro esempio è l'istruzione atomica `F&A` in cui il linearization point è il momento esatto in cui il valore del contatore viene incrementato atomicamente. Ciò assicura che, sebbene più thread possano richiedere simultaneamente l'incremento del contatore, ogni incremento risulti osservabile dagli altri thread in modo ordinato e coerente, come se le operazioni avvenissero una dopo l'altra [1].

Un'importante conseguenza della linearizzabilità è che semplifica la verifica della correttezza delle strutture dati concorrenti. Dimostrando che le operazioni sono linearizzabili, è possibile garantire che, indipendentemente dal numero di thread concorrenti, il comportamento della struttura dati sia sempre prevedibile e coerente, facilitando il debugging e migliorando la robustezza complessiva del software.

1.8 Code MPMC: Modelli e Stato dell'Arte

Nel contesto delle strutture dati concorrenti, le code MPMC (Multiple Producers, Multiple Consumers) rappresentano una classe di problemi complessi, fondamentali per applicazioni che richiedono un accesso concorrente a risorse condivise.

Uno dei primi modelli di code MPMC lock-based fu proposto da Hierily nel 1991, con l'introduzione di algoritmi basati su lock e mutex, concepiti per garantire l'integrità dei dati in ambienti altamente concorrenti. Sebbene tali approcci siano relativamente semplici da implementare, la loro scalabilità su sistemi paralleli di grandi dimensioni è spesso limitata.

Un aspetto cruciale è la granularità del locking. Il *coarse-grained* locking prevede che l'intera struttura dati venga protetta da un singolo lock, il che porta inevitabilmente a un elevato grado di contesa, soprattutto in scenari di alta concorrenza. Dall'altra parte, il *fine-grained locking* suddivide la struttura dati in più sezioni, ognuna protetta da un proprio lock, aumentando così il potenziale parallelismo. Questa suddivisione a granularità fine introduce una serie di complicazioni:

- Il costo di acquisizione e rilascio di un grande numero di mutex potrebbe essere troppo elevato, specialmente se in contesti a bassa contesa uno schema a granularità più bassa sarebbe sufficiente per garantire buone prestazioni.
- Coordinare correttamente l'accesso a molteplici mutex (comprese *reader mutex*[8]), richiede attenzione per evitare condizioni di stallo o errori di sincronizzazione. Evitare problemi di deadlock efficientemente impone, in generale, che le mutex debbano essere acquisite e rilasciate in un qualche tipo di *ordine globale* il che può causare problemi di *priority inversion*.
- Una granularità fine, può ridurre la contesa, ma al costo di un'implementazione più complessa e di un maggiore overhead (anche dovuto al traffico generato dalle cache), mentre una granularità troppo grossolana limita il parallelismo. Identificare il corretto trade-off non è spesso semplice.

Questi aspetti evidenziano come, nonostante la semplicità concettuale dei modelli lock-based, la loro effettiva applicazione in sistemi altamente paralleli richieda un attento bilanciamento tra contesa, overhead e complessità gestionale. In questo contesto, gli approcci lock-free per strutture dati concorrenti offrono una valida alternativa, in grado di ridurre tali problematiche e garantire prestazioni generalmente superiori, sebbene a scapito di una maggiore complessità nell'implementazione.

[8] *reader mutex*: mutex utilizzate in modelli reader-writer che permettono più letture concorrenti

Code CAS-Loop

Le code basate su CAS-loop sono tra le più diffuse grazie alla loro struttura semplice ed intuitiva. Un esempio rilevante è la Michael-Scott Queue (MS-Queue) [24], proposta nel 1996. Questa implementazione si basa su una lista concatenata che supporta operazioni di inserimento (enqueue) e rimozione (dequeue) rispettivamente sul nodo di coda e sul nodo di testa. Le operazioni evitano le race conditions aggiornando i puntatori dei nodi mediante istruzioni CAS (Compare-And-Swap).

La scelta di una lista concatenata semplifica notevolmente l'implementazione: un thread che intende inserire un nuovo elemento alloca un nuovo nodo, vi scrive il valore desiderato e, successivamente, lo rende visibile agli altri thread aggiornando in modo atomico il puntatore del nodo precedente. Analogamente, un thread che rimuove un valore dalla coda aggiorna atomicamente il puntatore pubblico (spostando la testa della lista) e restituisce il valore del nodo rimosso, deallocando quest'ultimo.

Le operazioni CAS vengono eseguite in un ciclo ripetuto fino al successo dell'aggiornamento. Tuttavia, l'aggiornamento atomico dei puntatori tramite CAS non elimina completamente il rischio di inconsistenze dovute al problema ABA, trattato in Sezione 1.4. Infatti, un tipico allocatore può effettuare caching delle regioni di memoria precedentemente liberate, facendo sì che i nuovi puntatori restituiti non siano necessariamente univoci. L'implementazione originale della MS-Queue affronta questa problematica impacchettando, in una singola parola di memoria, un contatore di riferimento univoco insieme al puntatore. Come sottolineato dagli autori, questa tecnica non elimina il problema ABA, ma ne riduce notevolmente la probabilità.

La lista concatenata di MS-Queue rende la struttura molto semplice e offre prestazioni competitive rispetto alla sincronizzazione bloccante basata su mutex ma introduce anche degli svantaggi:

- implica intrinsecamente l'indirezione dovuta all'accesso a regioni di memoria non necessariamente contigue. Questo sicuramente limita la località dei riferimenti sfavorendo buone prestazioni a livello di memoria cache
- ogni operazione richiede allocazione o deallocazione di memoria dinamica. Nonostante gli allocatori utilizzino sofisticati sistemi di caching, generalmente utilizzano primitive di sincronizzazione per gestire l'integrità delle risorse, creando quindi una fonte di rallentamento in uno scenario ad alta contesa
- si utilizza la tecnica del packing di più informazioni in una singola parola di memoria per contenere il problema ABA utilizzando operazioni di CAS a singola parola. Questa tecnica può essere valida in alcuni sistemi, ma non è supportata in generale limitando la portabilità dell'implementazione
- MS-Queue è proposta come una coda illimitata (unbounded) il che può costituire uno svantaggio in situazioni a throughput altamente variabile e requisiti sulla memoria stringenti.

Mantenendo lo stesso principio della MS-Queue, è possibile realizzare una coda MPMC (Multi-Producer, Multi-Consumer) basata su un buffer a dimensione limitata, come implementata nella libreria parallela FastFlow [25].

La coda viene gestita attraverso due indici atomici *head* e *tail*, che indicano rispettivamente la posizione dell'elemento da rimuovere e quella del prossimo spazio disponibile per l'inserimento. Entrambi gli indici vengono incrementati atomicamente a ogni operazione, garantendo operazioni CAS su dati univoci rendendo quindi la struttura *ABA-free*.

Il buffer è organizzato come un vettore di nodi, in cui ogni nodo contiene un campo per il valore e un indice numerico, che rappresenta il ciclo operazionale della cella. In questo modo, ogni nodo non solo memorizza i dati, ma è anche associato a un identificatore che si allinea al ciclo operazionale attuale della coda.

Una generica operazione di inserimento in una cella `c[i]` effettua un caricamento atomico dell'indice `tail` e dell'indice `c[i].idx`. Si controlla quindi l'uguaglianza tra i due indici e in caso affermativo si effettua una CAS su `tail` incrementandolo. In questo modo la cella è "riservata" e può essere modificata, aggiungendo il valore. Infine si incrementa l'indice della cella atomicamente. Quest'ultimo incremento atomico dovrebbe sempre avvenire dopo che la scrittura del valore è avvenuta in modo da non incorrere in successive letture dell'indice out-of-order. Nel caso di non uguaglianza tra i due indici allora si può discriminare il caso di lettura di una cella inconsistente (modificata da un altro processo) oppure il caso di coda piena che fa fallire l'operazione.

L'operazione di rimozione è simmetrica per quanto riguarda l'incremento dell'indice condiviso. In caso di corretto aggiornamento dell'indice il valore interno alla cella può essere copiato e restituito dopo aver incrementato l'indice della cella al prossimo ciclo operazionale. Questa operazione è richiesta per mantenere la semantica FIFO della struttura per assicurare che la cella appena svuotata sia l'ultima a essere riempita nuovamente.

Code F&A-Loop

Le code lock-free che si basano su CAS-loop, in situazioni di alta contesa soffrono di fenomeni cosiddetti CAS-*hotspots*: le operazioni di Compare-And-Swap sono localizzate su un numero ridotto di locazioni di memoria costituendo un fattore di bassa scalabilità. Questo problema può risultare limitante su certi tipi di architetture. Il modello x86 utilizza dei meccanismi specifici (come protocolli di coerenza come *MESI* e *bus locking*) per garantire che le istruzioni atomiche eseguano in tempo finito e non possano fallire arbitrariamente a causa di interferenze. Altre architetture, principalmente per ragioni di complessità, scelgono deliberatamente di implementare l'atomicità tramite istruzioni generiche come *Load-Linked/Store-Conditional* per cui le operazioni hanno successo, ma vengono riprovate in caso di interferenze. Limitare il numero di tentativi consecutivi è necessario per bene-

ficiare di una garanzia di wait-freedom ma nella maggior parte dei casi limita le performance.

Il problema dei CAS-*hotspot* viene affrontato implementando un sistema di assegnamento degli indici basato su un'operazione atomica F&A. In questo modo, invece di concentrare tutte le operazioni su poche locazioni di memoria, i processi vengono distribuiti su celle differenti. Successivamente, ciascuna cella viene aggiornata mediante operazioni CAS. Poiché queste operazioni vengono eseguite su locazioni di memoria distinte, la contesa viene distribuita, alleviando così il problema originario.

Per illustrare il funzionamento, si consideri il modello logico di una coda F&A-based implementata a partire da un buffer infinito. In questo esempio, sono definiti due valori riservati `empty`, `seen`, che indicano rispettivamente una cella vuota e una cella già esaminata da un consumatore. Inoltre vengono utilizzati due indici `head` e `tail`, entrambi inizializzati 0, per gestire l'accesso al buffer. Le operazioni di inserimento e rimozione funzionano secondo quanto illustrato in Figura 1.9.

```
1  //indici condivisi
2  atomic_int head = 0; atomic_int tail = 0;
3  //valori riservati
4  const word_t empty, seen;
5
6  void push(word_t item){
7      while(true){
8          t = F&A(&tail,1);
9          if(SWAP(&Q[t],item) != seen)
10             return;
11     }
12 }
13
14 bool pop(word_t *item){
15     while(true){
16         h = F&A(&head,1);
17         word_t x = SWAP(&Q[t],seen);
18         if(x != empty){
19             *item = x;
20             return true;
21         }
22         //coda vuota
23         if(LOAD(&tail) <= LOAD(&head) + 1)
24             return false;
25     }
26 }
```

Figura 1.9: Operazioni su Array infinito

In questa implementazione teorica non è necessario ricorrere a operazioni di CAS. Poiché il buffer è infinito e ogni processo che accede a una cella incrementa l'indice relativo all'operazione, non sussiste il rischio che una cella diventi inconsistente. Per questo motivo, al posto di CAS, si possono utilizzare operazioni di SWAP (`atomic_exchange`), che aggiornano in modo atomico una locazione di memoria restituendo il valore che vi era precedentemente.

Questo modello presenta un limite significativo: è soggetto a situazioni di *livelock*. Questo si presenta proprio a causa dell'operazione `F&A` che restituisce e incrementa l'indice del consumatore. Se un consumatore tenta di estrarre un elemento da una cella vuota, tale cella viene "bloccata" e risulta impossibile per un produttore inserirvi un nuovo elemento. Se questa condizione si verifica ripetutamente, i processi continuano ad operare, ma senza compiere progressi utili, configurando così una situazione di livelock.

Le implementazioni reali basate su questo modello logico risolvono il problema del livelock facendo in modo che, durante un'operazione di inserimento, un processo rilevi la condizione di livelock e la segnali agli altri processi, provocando il fallimento di ogni tentativo di inserimento successivo. Per questa ragione, tali modelli sono noti come *tantrum queues*. Il termine "tantrum" richiama infatti un comportamento impulsivo: una volta individuata la situazione di starvation negli inserimenti, il sistema interrompe ulteriori operazioni di inserimento. In sistemi reali, questo meccanismo è probabilistico e può essere influenzato dall'interleaving delle operazioni. Inoltre si utilizzano buffer circolari che simulano il comportamento di un array infinito, organizzati in una lista concatenata. Quando un processo rileva una situazione di livelock, segnala la condizione agli altri processi che falliranno ogni successiba operazione di inserimento sul buffer corrente. Successivamente, il processo procede ad allocare un nuovo buffer e inserisce il proprio elemento, garantendo così il progresso[9], e quindi tenta di concatenare il puntatore al nuovo buffer all'ultimo nodo della lista mediante un'operazione `CAS`. Questo paradigma rappresenta un buon compromesso, poiché nella maggior parte delle situazioni le operazioni avvengono internamente al buffer, sfruttando la località degli accessi di memoria, mentre solo in contesti di contesa elevata è necessario allocare un nuovo buffer.

Un'implementazione di una coda MPMC lock-free che segue questo approccio è LCRQ [26] di Afek e Morrison pubblicata nel 2013. Questa soluzione ha rappresenta lo stato dell'arte nel campo delle code lock-free, ed ha ispirato e influenzato numerosi modelli successivi.

CRQ[10] adotta buffer circolari, in cui ogni cella è strutturata per contenere due elementi fondamentali: un campo dedicato al valore di controllo e un indice numerico che rappresenta il ciclo operazionale della cella. Per garantire aggiornamenti atomici e mantenere la coerenza della struttura, CRQ utilizza operazioni di `CAS2`. Questo meccanismo è essenziale, poiché in ambienti concorrenti più processi potrebbero accedere simultaneamente alla stessa cella e tentare operazioni differenti. Di conseguenza, l'aggiornamento della cella avviene solo se i valori attuali, relativi sia all'indice sia al campo di controllo, coincidono con quelli letti dal processo in esecuzione, assicurando così che le modifiche vengano applicate solo in presenza di una corrispondenza completa dei dati.

[9] l'inserimento non può mai fallire dato che il buffer è locale al thread che lo alloca

[10] LCRQ è l'acronimo di *Linked Concurrent Ring Queue* dove "Linked" indica la necessità di utilizzare una lista concatenata di buffer CRQ

Come gia menzionato, l'operazione di `CAS2` rende il design più semplice ma costituisce una criticità in termini di portabilità dell'implementazione (dato che classi differenti di architetture la implementano diversamente). Per ovviare a questo problema, varie implementazioni hanno cercato di adattare il modello CRQ affinché utilizzi soltanto primitive portabili, facilmente gestibili a livello di linguaggio.

Ramalhete [27] ha proposto *F&AArrayQueue* che non riutilizza i buffer allocandone di nuovi quando terminato il loro utilizzo, disponendoli in una lista concatenata. Questo approccio non richiede indici interni alle celle del buffer quindi supporta operazioni di `CAS` a singola parola. Tuttavia soffre di una *memory-footprint* più elevata.

Nikolaev[28] ha proposto SCQueue che sfrutta il packing di valori di controllo e indici (entrambi 32-bit) in una sola parola di memoria. Questo approccio non è generalizzabile dato che non tutti i tipi di dati supportano il packing e che questo procedimento richiede accorgimenti particolari (soprattutto in sistemi a 32-bit) al fine di adattarsi ai fenomeni di overflow degli indici.

Un'altra proposta è quella di Feldman e Dechev [29] che integrano direttamente l'indice della cella nel campo del valore di controllo. Il loro sistema però utilizza celle vuote per immagazzinarlo e non risulta praticabile per linguaggi che utilizzano sistemi di recupero della memoria automatici (i.e Java, C#).

Infine, l'apporto più significativo è quello di Romanov e Koval [9], che nel 2023 hanno proposto LPRQ. Questo approccio introduce un metodo per sostituire le operazioni `CAS2` presenti in LCRQ con `CAS` a singola parola, mantenendo prestazioni comparabili a LCRQ e migliorando la portabilità dell'implementazione.

Capitolo 2

Progettazione di Code MPMC Bounded

Questo capitolo descrive il processo di progettazione e implementazione di una coda lock-free limitata, partendo dallo studio della struttura e del funzionamento di un buffer PRQ. La soluzione proposta si fonda sulla gestione concorrente di buffer circolari.

Viene illustrata l'implementazione di un wrapper per la gestione dell'allocazione dinamica dei buffer circolari, integrando il meccanismo di *Hazard Pointers* (vedere Sezione 1.6) per la corretta gestione della memoria dinamica. Questo approccio evita i rischi legati ad accessi a memoria non più valida, ottimizzando la gestione della memoria in ambienti ad alta concorrenza.

Successivamente, sono presentate due soluzioni per limitare la dimensione della coda, esplorando due metodi distinti per affrontare il problema della gestione della capacità e del controllo della memoria in scenari di elevato carico. In conclusione, viene fornita un'analisi dei *memory bounds* delle implementazioni proposte, valutando le implicazioni della dimensione limitata della coda in termini di consumo di memoria e scalabilità del sistema.

2.1 PRQ: Modello e Funzionamento

La *Portable Ring Queue* (PRQ) proposta da Romanov e Koval [9] rappresenta una delle soluzioni più avanzate nel contesto delle code lock-free MPMC. Il modello di funzionamento di PRQ si fonda sul precedente approccio denominato CRQ (*Concurrent Ring Queue*) di Afek e Morrison [26], che ha introdotto un'importante ottimizzazione che permette di ottenere maggiori prestazioni rispetto alle tradizionali implementazioni basate su CAS-loop. Tuttavia, CRQ utilizza operazioni di `CAS2` che non la rendono portabile su tutte le architetture. PRQ risolve questo problema modificando la struttura, in modo da utilizzare operazioni di `CAS`.

Per comprendere a pieno il funzionamento di PRQ, è necessario prima esplorare il modello di base di CRQ.

2.1.1 CRQ: Concurrent Ring Queue

Le code concorrenti classiche soffrono di degrado prestazionale a causa della contesa sui puntatori di inserimento ed estrazione. CRQ introduce il concetto di *ring buffer* per cui si utilizzano delle operazioni di `F&A` per accedere a una cella specifica del buffer, utilizzando operazioni di `CAS2` per inserire e estrarre i valori di controllo. L'utilizzo di `F&A` permette di distribuire la contesa delle *Compare-and-Swap* su multiple celle allo stesso tempo, il che migliora fortemente le prestazioni, riducendo i fenomeni di *CAS-hotspot* 1.3.

```
1 struct Cell {
2     safe:   1  bit (bool),
3     epoch:  63 bit (int),
4     val:    64 bit (void *)
5 }
6
7 struct CRQ {
8     int head 64 bit,
9     tail: struct {closed: 1 bit, t: 63 bit},
10    CRQ * next,
11    Cell * array;
12    int length;
13 }
```

Figura 2.1: Struttura CRQ [26]

La coda è implementata a partire da un buffer circolare di dimensione R, dove ogni cella contiene un indice interno, detto *epoch*, e un valore di controllo. L'accesso al buffer è gestito attraverso due indici atomici condivisi: `head`, `tail` utilizzati rispettivamente per operazioni di estrazione e inserimento.

Gli indici supportano unicamente operazioni di incremento (tramite `F&A`). Si lavora sull'assunzione realistica che gli indici non superino mai il valore di 2^{63}. Considerato questo, l'MSB viene utilizzato per il set di alcune flags. Poiché gli indici crescono indefinitamente, l'accesso alle posizioni del buffer avviene tramite l'operazione modulo.

2.1.2 Operazioni della coda

Estrazione

L'estrazione dalla coda avviene incrementando l'indice `head` tramite l'operazione `F&A` (Fetch-and-Add), ottenendo così il valore `h`, che corrisponde all'indice dell'operazione in corso. Successivamente, il valore viene letto dalla cella corrispondente nel buffer, previa verifica dell'epoch:

- Se `epoch > h`: la coda risulta già vuota e l'operazione fallisce.
- Se `epoch = h`: la cella contiene un elemento valido che può essere estratto.

- Se `epoch < h`: la cella contiene dati obsoleti e deve essere eseguita una transizione per garantirne la coerenza.

In caso di estrazione corretta, la cella viene aggiornata attraverso una *transizione di estrazione*, che utilizza l'operazione `CAS2` per marcare la cella come `empty` e avanzare l'`epoch` di R posizioni. In questo modo, la cella potrà essere utilizzata per un'estrazione solo quando tutte le altre celle saranno utilizzate, rispettando la semantica FIFO.

Se invece la cella risulta già vuota, viene effettuata una *transizione vuota*, che aggiorna l'`epoch` (avanzandola di R posizioni[1]) tramite `CAS2`, per assicurarsi che la cella non venga riscritta prematuramente.

Nel caso in cui l'`epoch` della cella sia inferiore all'indice `h`, ciò indica che il valore presente è obsoleto. In altre parole, l'operazione di estrazione corrente è arrivata "in anticipo" rispetto all'estrazione che avrebbe dovuto effettivamente rimuovere l'elemento, violando la semantica FIFO. In tal caso, viene effettuata una *transizione unsafe*, utilizzando una `CAS2` per settare il bit più significativo dell'`epoch`, segnalando alla prossima operazione di inserimento che la cella è occupata. Questa transizione previene la contesa tra l'operazione di inserimento successiva e l'operazione di estrazione che estrarrà dalla cella (rispettando la semantica FIFO). Una volta settato, il bit rimane attivo fino a quando un'operazione di inserimento non lo resetta con successo.

Se la coda risulta vuota, l'operazione di estrazione fallisce. Poiché gli indici della coda sono aggiornati tramite `F&A`, si può verificare uno stato inconsistente in cui `head > tail`. Questo stato può derivare da operazioni di estrazione successive che falliscono a causa della coda vuota, ma incrementano l'indice `head`. Per questo motivo in caso di coda vuota viene controllata questa condizione, e gli indici vengono riportati a uno stato consistente avanzando `tail` al valore di `head` qualora quello fosse superiore.

```
1 void fixState(CRQ * queue){
2     while(true){
3         uint64_t head = LOAD(&(queue->head));
4         uint64_t tail = LOAD(&(queue->tail));
5         if((head > tail) && CAS(&queue->tail),tail,head)
6             return;
7     }
8 }
```

Figura 2.3: Coerenza Indici CRQ

[1] emula il modello di array infinito trattato in Sezione 1.8

```
1 void * pop(CRQ* queue){
2     while(true){    //outer loop
3         uint64_t h = F&A(&(queue->head),1);
4         Cell* cell = &(queue->array[h % queue->length])
5         while(true){ //inner loop
6             void *val = LOAD(&cell->val);
7             [bool safe,uint64_t epoch] = LOAD(&[cell->safe, cell->epoch]);
8             if(epoch > h) break;
9             if(val != empty){
10                if(epoch == h){ //Dequeue transition
11                    if(CAS2(cell,[safe,h,val],[safe,h+length,empty]))
12                        return val;
13                } else    //Unsafe transition
14                    if(CAS2(cell,[safe,epoch,val],[0,epoch,val])) break;
15            } else    //Empty transition
16                if(CAS2(cell,[safe,idx,empty],[safe,h+length,empty])) break;
17        }   //end inner loop
18        //check for empty queue
19        [bool closed,uint64_t t] = LOAD(&(queue->tail));
20        if(t <= (h + 1)){
21            fixState(queue);
22            return empty;
23        }
24    }   //end outer loop
25 }
```

Figura 2.2: Estrazione CRQ [26]

Inserimento

L'inserimento avviene incrementando `tail` tramite `F&A`, ottenendo così l'indice `t` corrispondente all'indice dell'operazione. Una cella è in uno stato valido per l'inserimento quando questa è `empty` e la sua epoch è minore o uguale all'indice dell'operazione.

Quando l'operazione tenta di scrivere in una cella, deve prima controllare il bit più significativo della cella (bit *safe*). Se il bit non è stato resettato, l'inserimento può procedere aggiornando l'epoch e scrivendo il valore. Alternativamente, una precedente estrazione ha segnalato la cella come non sicura per gli inserimenti. In questo caso, si deve verificare che l'estrazione corrispondente all'epoch della cella non sia ancora iniziata. Si confronta l'indice `head` con l'indice dell'operazione: che si ricorda dovesse essere maggiore o uguale all'epoch della cella. Nel caso il primo sia strettamente minore del secondo allora l'operazione di inserimento è sarebbe supportata.

Come descritto nel caso dell'array infinito in Sezione 1.8, la coda può essere soggetta a fenomeni di *livelock*. In particolare, la *transizione vuota* potrebbe incrementare ripetutamente le epoch delle celle, impedendo il completamento delle operazioni di inserimento.

Per rilevare questa situazione, la coda implementa un meccanismo di chiusura unilaterale che segnala la condizione di *starvation* a ogni successiva operazione di inserimento, facendola fallire prematuramente. In questo contesto, viene impostato il bit `closed` nell'indice `tail`. Poiché tutte le operazioni di inserimento

leggono questo indice durante la F&A, è possibile verificare lo stato di chiusura e, se necessario, interrompere l'operazione anticipatamente.

```
1 const void * empty; //reserved value for empty
2
3 bool push(CRQ* queue, void* item){
4     while(true){
5         [bool closed,uint64_t t] = F&A((&queue->tail),1);
6         if(closed) return false;     //tantrum queue precondition
7         cell = &(queue->array[t % queue->length]);
8         void *val  = LOAD(&(cell->val));
9         [bool safe,uint64_t epoch] = [cell->safe,cell->epoch] // ATOMIC LOAD
10        if(val == empty){
11            if( (epoch <= t) &&
12                (safe || LOAD(&queue->head) <= t) &&
13                (CAS2(cell,[safe,epoch,empty],[true,t,item])))
14                    return true;
15        }
16        uint64_t h = LOAD(&(queue->head));
17        if(t - h >= length){ //starvation or full queue
18            TAS(&(queue->tail->closed));     //test and set
19            return false;
20        }
21    }
22 }
```

Figura 2.4: Inserimento CRQ [26]

2.1.3 Algoritmo PRQ

L'algoritmo PRQ modifica la struttura di CRQ per eliminare la necessità di utilizzare CAS2 nell'aggiornamento delle celle del buffer. Modificando le precondizioni di installazione e rimozione di valori dalle celle è possibile trasformare tutte le CAS2 dell'operazione di estrazione. Rimane necessaria una CAS2 nell'operazione di inserimento. Questa viene ottenuta tramite l'emulazione descritta in sezione 1.3.

Rimozione CAS2 in estrazione

In CRQ, ogni estrazione eseguita con successo avanza l'epoch della cella corrispondente, rendendola disponibile per un nuovo inserimento. Questo richiede l'uso di CAS2, poiché è necessario aggiornare simultaneamente sia il campo epoch sia il valore di controllo della cella (impostandolo a empty).

PRQ adotta un approccio differente: le operazioni di estrazione lasciano invariato il campo epoch, che viene invece incrementato in fase di inserimento. Per garantire il corretto funzionamento del meccanismo, un inserimento può avvenire solo in celle il cui epoch sia strettamente minore dell'indice dell'operazione. Di conseguenza, gli indici head e tail non possono più essere inizializzati a 0, ma vengono impostati alla lunghezza del buffer R, mentre le epoch assumono valori compresi tra 0 e R-1.

Questo permette di sostituire tutte le CAS2 nell'operazione di inserimento con CAS a singola parola.

Perchè questo procedimento funzioni, è fondamentale ottenere uno snapshot consistente della cella prima di aggiornarla. Dopo aver caricato atomicamente i campi di una cella (come avviene in CRQ), si verifica che il campo epoch, inclusa la safe flag, non sia cambiato. Se il valore è rimasto invariato, lo snapshot è consistente, poiché le epoch possono solo aumentare e la safe flag viene resettata esclusivamente da un'operazione di inserimento, che comporta un incremento di epoch. In caso contrario, si ripete il tentativo di acquisire uno stato consistente.

Emulazione CAS2 in inserimento

Come descritto precedentemente, la riformulazione delle condizioni di estrazione e inserimento delle celle permette di spostare l'incremento dell'epoch delle stesse in fase di inserimento. Questo richiede l'utilizzo di CAS2 per installare il valore e aggiornare l'epoch. In questo caso l'operazione di CAS2 viene emulata tramite tre operazioni di CAS eseguite in sequenza, come illustrato in Sezione 1.3. Questo procedimento utilizza un valore riservato per bloccare temporaneamente lo stato di una locazione di memoria mentre si performa un'operazione di CAS sull'altra parola. Se questa ha successo allora è possibile sostituire il valore riservato con il nuovo valore nella prima parola.

In PRQ, un unico valore riservato non è sufficiente, poiché operazioni di inserimento diverse potrebbero entrare in conflitto nella stessa cella, causando inserimenti inconsistenti. Per evitare questo, ogni operazione di inserimento riceve un valore riservato univoco, calcolato a partire dal tid del thread che la effettua. Questo valore riservato deve essere memorizzato nella cella che contiene il valore di controllo, che in C++ è un puntatore.

Per ottenere un puntatore non valido, si utilizza una tecnica di *bit stealing*. Il puntatore viene creato eseguendo un casting sul valore (2tid) + 1. Poiché, per motivi di allineamento della memoria, i puntatori dei valori di controllo delle celle (void *) sono sempre pari, l'indirizzo risultante è non valido e può essere impiegato come valore riservato.

La figura 2.5, mostra nel dettaglio l'emulazione CAS2 di un'operazione di inserimento. Nonostante l'emulazione consenta di installare correttamente gli elementi, essa presenta una carenza fondamentale: l'atomicità. L'uso di più operazioni atomiche implica che esista una *finestra temporale* tra ogni operazione, durante la quale il thread esecutore può essere interrotto. Inoltre, a causa delle interferenze durante l'inserimento degli elementi, alcune celle potrebbero rimanere bloccate indefinitamente, creando una condizione di stallo in cui i produttori possono inserire nuovi dati e i consumatori possono estrarre dati utili.

```
1 void *Reserved(int tid){ return (void *) ((tid * 2) + 1)}
2
3 bool push(PRQ* queue,int tid, void* item){
4     while(true){
5         [closed,t] = F&A((&queue->tail),1);
6         if(closed) return false; //tantrum queue precond.
7         Cell *cell = &(queue->array[t%length]);
8         void *val  = LOAD(&(cell->val));
9         [safe,epoch] = [cell->safe,cell->epoch]
10        if( insertion precondition ){ //CAS2 EMULATION
11            void *reserved = Reserved(tid);
12            if(CAS(&(cell->val),val,reserved){  //locks cell
13                if(CAS(&(cell->epoch),[safe,epoch],[true,t]){//updates epoch
14                    if(CAS(&(cell->val),reserved,item))//insert value
15                        return true;
16                // if interference: try to unlock the cell
17                } else CAS(&(cell->val),reserved,empty);
18            }
19        }
20        if(overflow || starvation){
21            T&S(queue->tail->closed);
22            return false;
23        }
24    }
25 }
```

Figura 2.5: Inserimento PRQ [9]

```
1 bool isReserved(void *ptr){return ((int) ptr % 2) != 0;}
2
3 void* pop(PRQ* queue,int tid){
4     while(true){
5         uint64_t  h = F&A(&(queue->head),1);
6         while(true){
7             Cell *cell = &(queue->array[head % queue->length]);
8             [bool safe, uint64_t epoch] = [cell->safe,cell->epoch];
9             if(cell inconsistent) continue;
10            /* ... try to dequeue ...*/
11            else {    //Unlock the cell
12                if(isReserved(value) && !CAS(&(cell->val),empty)) continue;
13                //advance the epoch
14                if(CAS(&(cell->safe_epoch),[safe,epoch],[safe,h])) break;
15            }
16        }
17        if(underflow){
18            fixState(queue);
19            return empty;
20        }
21    }
22 }
```

Figura 2.6: Estrazione PRQ [9]

Per risolvere questo problema, si introduce un meccanismo nel metodo di estrazione che permette di ”sbloccare” le celle. Se l’inserimento era in corso nella cella che viene sbloccata, questo fallisce e viene selezionata un’altra cella per l’inserimento. Se, invece, la cella è rimasta bloccata a causa di interferenze, essa viene ripristinata come vuota.

2.2 Linked Adapter

Entrambi i buffer circolari CRQ e PRQ rappresentano una soluzione efficiente, ma presentano una criticità fondamentale: il rischio di livelock in condizioni di elevata contesa tra i thread produttori, come specificato nelle sezioni 1.8 2.1.2. Il livelock si verifica quando i produttori non riescono a completare le operazioni di inserimento, portando il sistema a una condizione di starvation.

Per mitigare questo problema, un approccio efficace consiste nell'adozione di una lista concatenata di segmenti, come suggerito da Afek e Morrison [26]. In questo schema, quando un segmento rileva una condizione di livelock, esso viene chiuso a nuovi inserimenti e un nuovo segmento viene allocato per garantire la continuità delle operazioni. Questa strategia permette di costruire una coda unbounded che benefici del modello `F&A`-loop evitando il livelock.

Questa sezione descrive l'implementazione dell'interfaccia *Linked Adapter*, che permette di costruire una coda unbounded Multi-Producer-Multi-Consumer basata su una lista concatenata di segmenti contigui in memoria.

Questa interfaccia è essenziale per la realizzazione di code lock-free con garanzie di progresso, in particolare quelle basate su segmenti che adottano un modello `F&A`-loop. Tuttavia, il suo utilizzo non si limita a questo scenario: può infatti essere impiegata per implementare code unbounded sfruttando qualsiasi tipo di segmento FIFO che implementa la semantica di `tantrum queue`.

Un caso d'uso significativo è la variante linkedFastflow (L-ff), una versione unbounded della coda MPMC proposta nella libreria FastFlow [25], che si basa su un modello `CAS`-loop. I benchmark di L-ff e della sua controparte bounded sono presentati nella sezione 3.3.

2.2.1 Funzionamento

La struttura espone due metodi stub `push` e `pop` che oltre a installare e rimuovere i valori dai segmenti alla base dell'interfaccia, si occupano dell'allocazione e deallocazione di questi ultimi.

```
1 struct LinkedAdapter {
2     Segment_t* tail;
3     Segment_t* head;
4     int segmentSize;
5     HazardStruct* hazard;
6 }
```

Figura 2.7: Struttura Linked Adapter [26]

L'interfaccia espone due metodi principali, `push` e `pop`, che gestiscono l'inserimento e la rimozione degli elementi nei segmenti interni, oltre a occuparsi dell'allocazione e deallocazione della memoria associata.

In figura 2.7 sono illustrate le principali componenti necessarie al Linked Adapter. Il campo `hazard` è utilizzato per la gestione della memoria, ma verrà tralasciato in questa descrizione e illustrato successivamente.

L'adapter utilizza due riferimenti, `head` e `tail`, che sono globali a tutti i thread e vengono impiegati rispettivamente dalle operazioni di rimozione e inserimento. Ogni segmento contiene un riferimento al successivo, che viene settato dalle operazioni di `push` quando un nuovo segmento viene aggiunto alla lista, permettendo così di navigarla.

Le operazioni di `push` ottengono il valore del puntatore `tail` globale tramite un caricamento atomico. Per mantenere la semantica FIFO, l'inserimento deve avvenire sull'ultimo segmento. Se il campo `next` del segmento corrente è già settato, viene aggiornato il riferimento globale e l'inserimento viene riprovato.

Una volta identificato il segmento in cui inserire, l'inserimento viene effettuato tramite il metodo `push` specifico per la tipologia di segmento scelta. In caso l'inserimento fallisse, la semantica *tantrum queue* dei segmenti garantisce che nessun altro elemento possa essere piazzato. In questo caso, viene allocato un nuovo segmento e l'inserimento vi avviene con successo, poiché questo non è ancora pubblico al resto dei thread. La pubblicazione è realizzata tramite una `CAS` sul campo `next` del segmento corrente. Se la `CAS` ha successo, l'inserimento è completato, altrimenti un altro thread ha già aggiunto un nuovo segmento, quindi l'inserimento verrà riprovato su quest'ultimo.

Va sottolineato che la procedura di inserimento in un segmento privato e la successiva concatenazione alla lista è una condizione di progresso che garantisce la qualità di lock freedom della struttua.

Le operazioni di `pop` sono simmetriche. Viene effettuato un caricamento atomico del riferimento `head`. L'estrazione dal segmento viene tentata e, se non ha successo, si verifica l'esistenza di un riferimento `next`. In tal caso, si riprova l'estrazione sul segmento corrente, poiché esiste una finestra temporale in cui, tra l'ultimo tentativo di estrazione e la concatenazione del nuovo segmento, potrebbe essere stato eseguito un inserimento. Se l'estrazione non ha successo, significa che il segmento è vuoto e nessuna estrazione è più possibile.

La transizione da un segmento a quello successivo viene sempre eseguita aggiornando il riferimento globale tramite `CAS`. Se l'operazione ha successo, il segmento corrente viene rimosso dalla lista e può essere deallocato. La deallocazione è gestita tramite un procedimento speciale descritto nella sezione successiva.

2.2.2 Gestione della Memoria

La gestione della memoria è un aspetto cruciale nel design dell'adapter. Poiché l'accesso è multithreaded e non sincronizzato, è necessario implementare meccanismi che prevengano problemi come dangling references o memory leaks. Nei linguaggi ad alto livello, i sistemi di garbage collection semplificano questa gestione automatizzandola. Tuttavia, tali sistemi spesso impiegano primitive di sin-

cronizzazione come mutex o semafori, il che può compromettere l'efficienza delle strutture lock-free, vanificando gli sforzi per eliminare i colli di bottiglia legati alla sincronizzazione esplicita.

In C++, vengono proposti gli smart pointer come astrazioni per semplificare la gestione della memoria. Un esempio tipico sono gli `std::shared_ptr`, che utilizzano un sistema di reference counting per determinare quando la deallocazione di un oggetto condiviso è sicura. Un `shared_ptr` è costituito da un raw pointer e un control block, che contiene un contatore di riferimenti sul quale le operazioni di incremento e decremento sono atomiche. Sebbene questi smart pointer siano thread-safe in alcuni contesti, la modifica del raw pointer da parte di più thread può generare race condition, e il reference counting provoca cache thrashing, riducendo l'efficienza complessiva.

Per affrontare queste problematiche, si opta per un modello alternativo di gestione della memoria che utilizza gli *Hazard Pointers* [21], una tecnica più adatta per ambienti lock-free. In questo modello, ogni thread scrive i puntatori che sta attualmente utilizzando in una struttura dati condivisa, tipicamente un vettore, dove ogni cella è scrivibile da uno specifico thread. Questo approccio evita il cache thrashing grazie all'uso di tecniche di padding, poiché ogni thread modifica solo la propra cella, riducendo le interferenze tra thread diversi.

In questo contesto ogni caricamento atomico di puntatori `head` o `tail` deve corrispondere a una scrittura sul vettore hazard per segnalare l'utilizzo del riferimento. Quando il metodo corrispondente risulta completato, viene effettuata un'altra scrittura nel vettore che annulla il riferimento precedentemente settato (viene utilizzato `nullptr` come valore nullo).

In aggiunta al vettore condivisa, ogni thread mantiene una *free list*, un vettore di dimensione arbitraria utilizzato per raccogliere i riferimenti alla memoria che deve essere deallocata. Una regione di memoria può essere deallocata in sicurezza solo quando il corrispondente puntatore non appare più in nessuna delle celle del vettore hazard, il che indica che nessun thread la sta più utilizzando. Questo si basa sul modo in cui le operazioni di estrazione gestiscono i riferimenti. Infatti, quando l'operazione di pop avanza il riferimento della testa della lista, lo fa tramite un'operazione atomica (CAS). Una volta che la testa della lista è avanzata, i thread che successivamente leggono il riferimento globale non possono più ottenere il vecchio riferimento.

Per ridurre l'overhead[2] associato alla completa lettura del vettore hazard durante le operazioni di deallocazione, si decide che solo il thread che avanza la testa della lista possa tentare di deallocare il segmento corrente. Questo approccio, sebbene ottimizzi l'efficienza, introduce una limitazione. Se un thread consumatore aggiorna la testa della lista senza riuscire a deallocare il segmento corrente, e successivamente tutti gli altri aggiornamenti della testa sono effettuati da altri thread, il segmento rimarrà pendente nella free list per un tempo indefinito, seppur limitato. Questo comportamento è un compromesso che consente di mantenere

[2]dovuto allo scarso utilizzo della cache

l'efficienza e la progressione lock-free, ma che potrebbe portare a una memory footprint considerevole.

2.2.3 Pseudocodice

Nelle successive due figure è illustrato lo pseudocodice di Linked Adapter. Si precisa che il i metodi proposti, sono esplicativi e mancano di dettagli per la riduzione dell'overhead. Inoltre i metodi di gestione degli Hazard Pointers sono omessi per ragioni di semplicità e dato che il funzionamento di questi è *implementation specific*.

```
1 //Adds ptr to the shared vector [return ptr]
2 void *protectPtr(HazardStruct *hp, void *ptr, int tid);
3 //Clears ptr from the shared vector
4 void releasePtr(HazardStruct *hp, void *ptr, int tid);
5
6 void push(LinkedAdapter *queue, void *item, int tid){
7     Segment_t *tail = protectPtr(queue->hazard, LOAD(&(queue>tail)), tid);
8     while(true){
9         if(tail inconsistent) continue;
10        if(nextTail != null){    //check if exists next segment
11            CAS(&(queue->tail),tail,newTail);
12            continue;
13        }
14        if(segmentPush(tail,item)){ // try push in current segment
15            releasePtr(queue->hazard,tail,tid);
16            return; //successful insertion
17        }
18        Segment_t * newTail = new Segment(queue->size);
19        segmentPush(newTail,item); //always successful
20        if(CAS(&(tail->next),NULL,newTail)  // link the new segment
21            return;
22        else delete newTail;    //try again
23    }
24 }
```

Figura 2.8: Linked Adapter: Inserimento [9]

```
1 //Try dealloc a ptr [adds to the free list]
2 void retirePtr(HazardStruct *hp, void *ptr, int tid);
3
4 void *pop(LinkedAdapter *queue, int tid){
5     void *item = NULL;
6     Segment_t *head = protectPtr(queue->hazard, LOAD(&(queue->head)), tid);
7     while(true){
8         if(head inconsistent) continue;
9         if(item = segmentPop(head) != NULL) break;
10        Segment_t *nextHead = LOAD(&(head->next));
11        if(nextHead == NULL) break;
12        if(item = segmentPop(head) != NULL) break;
13        if(CAS(&(queue->head),head,nextHead)){  //advance global head
14            retirePtr(queue->hazard,head,tid);  //try-dealloc current head
15            break;
16        }
17    }
18    releasePtr(queue->hazard,head,tid);
19    return item;
20 }
```

Figura 2.9: Linked Adapter: Rimozione [9]

2.3 Bounded Adapter

Posto che Linked Adapter, illustrato in sezione precedente permette di costruire delle code unbounded a partire da segmenti limitati contigui in memoria, la qualità non limitata non risulta essere sempre desiderevole. Infatti l'imprevedibilità dell'impatto sulla memoria può essere un fattore limitante per sistemi con requisiti di memoria stringenti o applicazioni a carico di lavoro altamente variabile. In questa sezione viene proposto un metodo algoritmico che modifica Linked Adapter per ottenere *Bounded Adapter* che costruisce delle code limitate a partire da segmenti arbitrari con semantica di *tantrum queues* (Sezione 2.2). Successivamente viene analizzato il memory bound della soluzione proposta. Infine viene proposto il concept design di un Bounded Adapter che non richiede di effettuare allocazioni di memoria successive all'inizializzazione.

2.3.1 Intuizione: global counter

Un approccio immediato per imporre un vincolo alla memoria utilizzata consiste nell'introdurre un contatore atomico globale che tenga traccia del numero di elementi attualmente presenti nella struttura. Tale contatore viene aggiornato attraverso operazioni atomiche come Fetch&Add e Fetch&Sub, consentendo un incremento e un decremento concorrente senza ricorrere a meccanismi di lock espliciti.

L'idea di base è che ogni operazione di inserimento controlli il valore corrente del contatore prima di procedere. Se il numero di elementi è inferiore a una soglia prestabilita, l'inserimento avviene normalmente, aggiornando il contatore per riflettere la nuova dimensione della coda. Se invece la soglia viene superata, l'inserimento fallisce, evitando così un'espansione eccessiva della struttura e garantendo il rispetto del vincolo di memoria.

Sebbene questa soluzione sia semplice da implementare, presenta criticità che ne limitano l'efficacia in contesti altamente concorrenti. Il primo problema è il *cache thrashing*: ogni operazione di inserimento deve leggere e aggiornare il contatore globale, il che porta a una continua invalidazione della linea di cache che lo contiene. In scenari con un elevato numero di thread, ciò introduce un collo di bottiglia che limita il throughput massimo della coda. Una possibile mitigazione consiste nell'introdurre due contatori distinti: uno globale ai produttori per tracciare gli elementi inseriti e uno globale ai consumatori per registrare gli elementi estratti. La differenza tra questi due valori determina il numero effettivo di elementi presenti. Tuttavia, pur riducendo l'overhead in scenari con parallelismo contenuto, questo approccio diventa inefficace quando il numero dei thread cresce oppure quando un singolo thread svolge sia il ruolo di produttore che di consumatore.

Un'ulteriore criticità riguarda la *coerenza dello stato* della coda. Poiché le operazioni sono non bloccanti, possono verificarsi finestre di inconsistenza in cui il

contatore segnala una condizione di coda piena, ma nuovi elementi riescono comunque a essere inseriti. Questo disallineamento si verifica perché il contatore e la struttura dati sottostante non vengono aggiornati in un'operazione atomica unica. Nel caso peggiore, il numero di elementi erroneamente aggiunti può essere stimato come $\min(p, n)$, dove p è il numero di thread che eseguono operazioni di inserimento concorrenti e n è il numero di celle dei segmenti concatenati (supponendo che ogni segmento abbia lo stesso numero di celle).

Infine, la gestione della memoria rappresenta un ulteriore punto critico. Poiché il Bounded Adapter deriva dal Linked Adapter, utilizza segmenti concatenati allocati dinamicamente. Dato che non esiste un limite esplicito sul numero di segmenti che possono essere allocati, in condizioni di starvation dei produttori, ogni nuova operazione di inserimento può determinare l'allocazione di un segmento aggiuntivo, portando nel caso peggiore a una complessità spaziale pari a $O(n^2)$.

2.3.2 Approccio: segment counting

La principale limitazione del contatore globale sugli elementi è il cache thrashing dovuto al continuo accesso concorrente. Una strategia alternativa consiste nel spostare il livello di *granularità*: invece di contare i singoli elementi, possiamo monitorare il numero di segmenti attualmente presenti nella lista concatenata. In questo modo, il contatore viene aggiornato solo quando un nuovo segmento viene concatenato o rimosso, riducendo significativamente l'overhead.

Impostando un numero massimo s di segmenti utilizzabili, è possibile costruire una coda di dimensione n con segmenti di ampiezza $\frac{n}{s}$. Data la natura circolare della maggior parte dei segmenti FIFO basati su `F&A`, in contesti bilanciati la necessità di allocazione di nuovi segmenti tende a distribuirsi nel tempo, il che può distribuire le letture e scritture sul contatore dato che l'overhead di inserimento all'interno di un segmento può considerarsi trascurabile.

Rispetto alla gestione tramite contatore globale per singoli elementi, questo approccio offre una gestione della memoria più prevedibile, poiché limita il numero di segmenti allocati e migliora l'efficienza del caching dell'allocatore. Tuttavia, introduce un compromesso: il vincolo sul numero massimo di segmenti attivi può impedire alcuni inserimenti anche quando esiste ancora spazio libero all'interno dei segmenti già allocati. Questo fenomeno è riconducibile a un problema di *frammentazione interna*, dato che la condizione di inserimento dipende da un limite sui segmenti allocati rispetto che un limite sugli elementi presenti nella coda.

Il grado di frammentazione interna dipende direttamente dal numero di segmenti scelto in fase di inizializzazione. Se s aumenta, la frammentazione si riduce, ma la dimensione più contenuta dei segmenti tende ad aumentare la frequenza con cui ne è richiesta l'allocazione. Nel caso limite in cui $s = n$, questo approccio diventa equivalente alla gestione con contatore globale per singoli elementi, riproponendone le stesse limitazioni.

La frammentazione interna può essere approssimata come una percentuale degli elementi totali della coda. In particolare, si osserva che al caso pessimo la frazione di spazio non utilizzabile risulta essere la lunghezza di un singolo segmento, stimata come $\frac{n}{s}$.

Ad esempio, si consideri il caso in cui $s = 4$. La frammentazione interna, stimata attorno al 75% della dimensione della coda, implica che nel caso peggiore la coda risulterà piena pur contenendo solo il 75% degli elementi rispetto all'utilizzo di un buffer contiguo. Questo effetto deriva rigidità imposta dal numero massimo di segmenti allocabili.

Con un contatore globale per segmenti, rimane possibile un disallineamento tra il valore del contatore e lo stato effettivo della coda. Tuttavia, in questo modello, il numero massimo di segmenti allocabili erroneamente diventa costante: anche in presenza di più thread che tentano di concatenare nuovi segmenti, solo uno può riuscire grazie al limite imposto dalla `CAS` necessaria per il linking.

Posto questo e considerata la frammentazione interna al segmento è possibile stimare un intervallo per il numero di elementi che possono essere contenuti nella coda. Il *limite inferiore* è dato dalla *frammentazione interna*: una volta allocato un numero di segmenti che garantisce che la coda possa ospitare n elementi, il segmento in testa (di dimensione $\frac{n}{s}$) deve essere completamente svuotato prima che possa essere allocato un nuovo segmento. D'altra parte, il *limite superiore* è determinato dalla *possibile allocazione di un segmento aggiuntivo*, che può verificarsi a causa del disallineamento del contatore dallo stato corrente della coda, portando a un errore nella gestione degli slot disponibili.

Dato che questa soluzione utilizza una precondizione basata sul numero di segmenti da cui la coda è composta è necessaria una considerazione sul *limite inferiore* di elementi contenibili, nel caso i segmenti fossero di tipo `F&A` based. Come gia trattato questi segmenti utilizzano la semantica rilassata di *tantrum queues* quindi sono possibili condizioni in cui è necessario allocare un nuovo segmento nonostante quello corrente non sia pieno. Questo è dovuto alla precondizione che rileva il possibile livelock all'interno del segmento. Possiamo considerare come caso pessimo la situazione in cui un thread inserisce un unico elemento all'interno del segmento che viene successivamente chiuso a causa del livelock rilevato. Questo è giustificato dal fatto che ogni segmento correttamente concatenato alla lista contiene l'elemento della corrispondente operazione di inserimento. In questo modo il limite inferiore al numero degli elementi che una coda di questa fattura contiene, data una condizione arbitraria, diventa uguale al numero dei segmenti dichiarato in fase di inizializzazione.

Pertanto, il numero effettivo di elementi contenibili nella coda indipendentemente dal contesto e tenendo conto di possibili interferenze tra i thread risulta essere un intero n_{real} compreso nell'intervallo sottostante:

$$n_{\text{real}} \in \left[s, n + \frac{n}{s} \right]$$

Sebbene il limite inferiore sia basso è doveroso riconoscere che questo non si deve all'implementazione del *Bounded Adapter* ma alle scarse garanzie di progresso di segmenti di tipo `F&A` che rendono complicato il loro utilizzo in situazioni reali. Infatti anche la verisone *Linked Adapter* proposta da Afek e Morrison [26] soffre di una problematica analoga che però è irrilevante dato che nuovi segmenti possono essere allocati arbitrariamente data la qualità unbounded dell'implementazione. D'altra parte, i risultati sperimentali espressi in sezione 3.3 mostrano che in tutti i constesti proposti il limite inferiore nel caso di Bounded Adapter non si ponga come un fattore limitante.

2.3.3 Dettagli Implementativi

Sebbene questa implementazione appaia semplice e diretta, è fondamentale adottare alcune precauzioni per prevenire situazioni di livelock, in particolare considerando la struttura dei segmenti basati su `F&A`, come discusso nella Sezione 1.8. Afek e Morrison [26] hanno evidenziato come la semantica delle *tantrum queues* garantisca la chiusura dei segmenti CRQ in caso venga rilevato il livelock, impedendo ulteriori inserimenti[3]. Quando un segmento viene chiuso, risulta teoricamente necessario deallocarlo o almeno reinizializzarlo. Infatti, nuovi tentativi di inserimento causano un incremento dell'indice (dovuto all'operazione di `F&A`), il che, se protratto indefinitamente, può generare un ciclo in cui le operazioni di inserimento non riescono a rilevare la condizione di coda vuota.

Nel caso di contatori per singoli elementi, il problema non sussiste, poiché il fallimento dell'operazione viene intercettato anticipatamente. Tuttavia, nel caso di un contatore per segmenti, un fallimento dell'inserimento potrebbe impedire l'allocazione di nuovi segmenti a causa della coda piena. In tale scenario, tentativi ripetuti di `push` sul segmento corrente rischiano di condurre a una situazione di attesa indefinita.

Per mitigare questo problema, è necessario un meccanismo in grado di rilevare la chiusura del segmento senza incrementare l'indice dell'operazione e senza invocare il metodo di inserimento. Convenzionalmente, lo stato di chiusura di una *tantrum queue* è indicato dal bit più significativo dell'indice utilizzato per le operazioni di inserimento. Pertanto, è possibile effettuare un caricamento atomico dell'indice e applicare un'operazione bitwise per verificare se l'inserimento sia sicuro. Sebbene efficace, questo approccio introduce il caricamento atomico di una variabile condivisa, con possibili ripercussioni sul cache thrashing. Un'alternativa consiste nell'impiego di una flag locale per ogni thread [4], che segnali se un'operazione di inserimento potrebbe risultare non sicura. In tal caso, il caricamento dell'indice condiviso viene eseguito solo quando la flag è stata impostata

[3] Lo stesso principio si applica anche ai segmenti PRQ e, in generale, alle *tantrum queue* basate su `F&A`

[4] In `C++` è possibile utilizzare variabili `thread_local`

da un'operazione di inserimento fallita, resettandola alla transizione su un nuovo segmento.

Come evidenziato dai risultati sperimentali in Sezione 3.3, questa rappresenta una strategia efficace, sebbene la sua formulazione rimanga in parte arbitraria.

Un altro aspetto cruciale riguarda la gestione della memoria. Sebbene il consumo di memoria utilizzata dalla lista sia limitato, poiché nuovi inserimenti vengono bloccati in caso di coda piena, la memoria effettivamente allocata dipende dall'impiego di *Hazard Pointers*, necessari per prevenire errori di accesso concorrente. Nel caso del *Linked Adapter* (Sezione 2.2), un elevato consumo di memoria non costituiva un problema critico, grazie alla natura unbounded della struttura dati. In tal caso, la deallocazione poteva avvenire in modo "lazy", tentando di liberare i segmenti presenti nelle *free list* solo all'inserimento di un nuovo segmento. Tuttavia, nel contesto attuale, la gestione della memoria richiede un approccio più raffinato.

Non risulta vantaggioso tentare la deallocazione dell'intera *free list* a ogni operazione di estrazione, poiché ciò comporterebbe lo scorrimento di un vettore condiviso tra tutti i thread, con un impatto negativo sulle prestazioni della cache. Per affrontare questa problematica, è stata adottata un'euristica simile a quella precedentemente descritta. In primo luogo, il metodo di deallocazione della *free list* è stato modificato in modo da restituire il numero di segmenti effettivamente deallocati. Inoltre, ogni thread che esegue operazioni di estrazione dispone di una flag locale, impostata quando una richiesta di deallocazione fallisce. In tal caso, nelle operazioni di estrazione successive, il thread tenterà la deallocazione fino a quando almeno un segmento verrà rilasciato, momento in cui la flag viene resettata.

2.3.4 Memory Bound

Nonostante *Bounded Adapter* si proponga come *wrapper* per la costruzione di code limitate a partire da buffer `F&A`, garantire il progresso in scenari arbitrari richiede un supporto costante da parte di un allocatore di memoria per il riciclo dei segmenti. Il corretto funzionamento della coda risulta strettamente legato alla continua allocazione di memoria, il che impone che l'allocatore stesso fornisca garanzie di progresso *lock-free* o *wait-free* per preservare la natura non-blocking dell'algoritmo. Tuttavia, anche supponendo l'uso di allocatori non bloccanti, il problema dell'allocazione di memoria persiste: come ogni altra risorsa, la memoria è limitata e, in caso di esaurimento, il progresso dell'algoritmo non potrebbe più essere garantito. Sia LCRQ che LPRQ [26], [9] non esplorano questo problema in dettaglio dato che la semantica di una coda unbounded deve necessariamente garantire il supporto a un'allocatore di memoria.

Nonostante la necessità di allocazione di memeoria dinamica, *Bounded Adapter* propone un memory bound finito con complessità asintotica $O(n)$, lineare nel numero degli slot della coda. Questo memory bound non include il costo in spazio

aggiuntivo associato alla gestione della memoria. Infatti dato che l'implementazione è portabile verso ambienti a più alto livello di astrazione, l'implementazione dedicata al *safe memory management* può avere overhead e garanzie di progresso variabili.

2.4 Concept: Memory-Free Bounded Queue

A partire dal *Bounded Adapter* discusso nella sezione precedente, si propone un approccio per costruire una *Bounded Queue* che non richieda allocazioni aggiuntive oltre alla fase di inizializzazione.

Per garantire questa proprietà, pur mantenendo l'assenza di livelock, la coda deve essere composta costantemente da un numero fisso di segmenti *s*. L'allocazione dei segmenti avviene in fase iniziale, e i riferimenti vengono gestiti in una struttura dedicata, denominata *Buffer Pool*, la cui implementazione verrà descritta successivamente. Oltre a fungere da allocatore astratto per i thread produttori, il metodo di accesso al *Buffer Pool* costruisce la precondizione di inserimento che determina la natura *bounded* della coda.

Affinché la coda mantenga la proprietà di *lock-freedom*, tutte le strutture dati necessarie al suo funzionamento devono offrire garanzie di progresso non bloccanti espresse nella forma di proprietà *lock-free* o *wait-free*.

Una possibile implementazione di un *Buffer Pool lock-free*, si basa su un modello di coda `CAS`-loop. Questo può determinare penalità di `CAS`-*hotspot* di cui andrebbe analizzato accuratamente l'overhead, però considerando segmenti `F&A`-based di una certa dimensione e un carico di lavoro semi-bilanciato, possiamo supporre che questi siano ammortizzati da transizioni di segmenti ben distribuite nel tempo.

Poiché in questo caso la semantica FIFO non è strettamente necessaria, è possibile adottare una soluzione che minimizzi la contesa. Come discusso nella Sezione 1.8, le code FIFO che utilizzano un modello `F&A` loop distribuiscono meglio gli *hotspot* derivanti dalle operazioni `CAS` al costo di una predisposizione al livelock. Questo fenomeno è una conseguenza diretta della necessità di garantire la semantica First-In-First-Out nell'inserimento e estrazione di elementi. Teoricamente, una struttura con una semantica operazionale rilassata, che sfrutta `F&A`-loop per distribuire la contesa, non soffre di problemi di livelock. Un esempio di una generica collezione non ordinata è trattato nella Sezione 1.3.

Dimostrata la fattibilità della costruzione del *Buffer Pool* con questa strategia, il fallimento nell'ottenimento di un nuovo buffer può essere utilizzato come precondizione di fallimento per inserimenti in caso di coda piena. Dopo l'inizializzazione della coda, i thread consumatori inseriranno riferimenti di segmenti vuoti nel *Buffer Pool* (alternativo alla deallocazione dell'implementazione in sezione 2.3), mentre i thread produttori, quando necessitano di un nuovo segmento da concatenare, ispezionano il pool. In teoria, se la coda è piena, il *Buffer Pool* risulterà vuoto e l'operazione di inserimento fallirà come previsto dalla semantica. In pratica, la condizione di fallimento richiede ulteriori verifiche, approfondite in seguito.

Un ulteriore aspetto da affinare rispetto all'esempio precedente è la gestione delle *free list* da parte dei consumatori. L'impiego del *vettore hazard*, descritto in sezione 2.2.2, rimane necessario anche in assenza di allocazioni dinamiche. Infatti, un segmento non può essere reso disponibile per il riutilizzo finché un thread ne detiene un riferimento, pena la violazione della semantica FIFO della struttura. Le operazioni di allocazione e deallocazione nelle implementazioni *Linked Adapter* e *Bounded Adapter* servono a compensare la mancanza di un controllo aggressivo per la deallocazione dei segmenti. Ciò è dovuto al fatto che ogni thread necessita di una *free list* dedicata: non è possibile condividerne una tra più thread, poiché riferimenti allo stesso segmento potrebbero essere inseriti da thread distinti[5], con il rischio di errori di tipo `double-free`. Tuttavia, eliminata la necessità del supporto di memoria dinamica, diventa teoricamente possibile e corretto implementare una *free list* condivisa. La scelta della struttura dati più adatta non è triviale: infatti più a lungo un segmento rimane nella *free list*, maggiore è la probabilità che non sia più utilizzato e quindi riutilizzabile. L'uso di una coda FIFO appare vantaggioso, sebbene implichi necessariamente un'implementazione basata su `CAS`-loop per garantire l'assenza totale di fenomeni di livelock. Realisticamente, è possibile fare a meno dell'ordinamento FIFO quando il numero dei segmenti è contenuto, rendendo così possibile un'implementazione simile a quella del buffer pool. Per completare il ragionamento, però, è necessario definire le operazioni di inserimento e rimozione. Il funzionamento principale di queste operazioni rimane analogo a quanto descritto nelle sezioni 2.2 e 2.3, ma occorre specificare la gestione del riciclo dei segmenti. In questa implementazione, si suppone che la *free-list* condivisa sia realizzata come una collezione non ordinata basata su `F&A`-loop, analogamente al buffer pool. Data una coda di n slot composta da s segmenti di dimensione $\frac{n}{s}$, si rende necessario un buffer pool e una free list entrambi da $(s-1)$ slot[6].

Le operazioni di estrazione, in questo caso, non prevedono la deallocazione dei segmenti quando questi vengono svuotati. Invece, viene verificato se l'*hazard vector* contiene il riferimento al segmento corrente. Se il segmento è ancora in uso da almeno un thread, non può essere riutilizzato e viene inserito nella *free list* condivisa. In caso contrario, il segmento può essere inserito direttamente nel *buffer pool* previa reinizializzazione. Questo può avvenire tramite il flip del bit di chiusura con complessità in tempo costante. È importante notare che la reinizializzazione degli slot non è strettamente necessaria per garantire il corretto funzionamento delle operazioni di inserimento ed estrazione, poiché la chiusura del segmento è necessaria per evitare il livelock ma non modifica la semantica FIFO delle operazioni. Inoltre, dato l'uso di dati a 64 bit, la probabilità di overflow degli indici è escludibile[7]. Nelle implementazioni precedenti 2.2, 2.2 il costo di

[5] I riferimenti a zone di memoria restituiti dagli allocatori non sono univoci a causa di tecniche di caching

[6] Uno degli s segmenti è sempre in uso come testa della lista concatenata.

[7] supponendo un'operazione di incremento atomica ogni nanosecondo un contatore a 64 bit

inizializzazione di ogni segmento ha complessità $\Theta(n)$ nel numero degli slot del segmento.

Quando un'operazione di inserimento richiede un nuovo segmento, dapprima si verifica se il *buffer pool* ne contiene almeno uno. Se è così, questo viene rimosso e l'inserimento può avvenire al suo interno. Se il pool è vuoto, si verifica se nella *free list* esiste un segmento che può essere riutilizzato. In questo modo, un thread produttore esamina la *free list*. Per evitare estrazioni e reinserimenti ripetuti dalla struttura, è possibile implementare la logica di controllo direttamente nelle operazioni di estrazione della *free list*. Se viene individuato un segmento riutilizzabile, l'inserimento può avvenire su di esso; in caso contrario, l'operazione fallisce per la mancanza di ulteriori segmenti, come previsto dalla specifica bounded.

In conclusione, è si propone un idea di implementazione di una coda limitata *allocation-free*. La coda mantiene la stessa semantica dell'implementazione *Bounded Adapter* descritta in Sezione 2.3.

richiederebbe circa 292 anni per subire un singolo overflow

Capitolo 3

Esperimenti e Risultati

In questo capitolo vengono descritti gli esperimenti condotti per valutare le prestazioni delle soluzioni esistenti e di quelle proposte in un ambiente multithread. L'obiettivo principale è analizzare il comportamento delle diverse implementazioni in scenari differenti. In particolare, ci si concentrerà sull'analisi dell'overhead interno della coda e su come le prestazioni variano al variare di specifici parametri. Inoltre, verranno esaminati i limiti legati all'occupazione della memoria delle soluzioni *bounded* proposte. Il codice sorgente delle implementazioni e dell'ambiente di test è pubblico e consultabile tramite il link: `https://github.com/MPiras055/testQueue.git`

3.1 Specifiche Hardware

Per l'esecuzione degli esperimenti è stata utilizzata la piattaforma di calcolo *Titanium*, fornita dall'Università di Pisa. Titanium è equipaggiata con un'architettura NUMA dual-socket, dotata di processori `AMD EPYC 7551` operanti a una frequenza di clock base di 2.0 GHz. Ogni nodo NUMA della macchina è composto da *otto core fisici*, organizzati in due *Core Complex (CCX)* che identificano un insieme di quattro core che condividono almeno un livello di cache. Ogni CCX comprende *tre livelli di cache*:

- `L1-cache` è privato per core e composto da `L1i,L1d`: rispettivamente cache istruzioni da 32 kB e cache dati da 64 kB.
- `L2-cache` è privato per ciascun core con dimensione di 512 kB.
- `L3-cache` è condiviso tra i quattro core del CCX con dimensione di 8196 kB.

La macchina è organizzata in 8 *Core Complex Die* (CCD), ognuno dei quali è composto da due CCX, come indicato in Figura 3.1. Ogni CCD identifica un Numa Node. In questo caso specifico solo 2 nodi (*Memory Nodes*) hanno accesso alla memoria principale, la quale è partizionata fisicamente in due regioni da 64 GB. Il

resto dei nodi (*Memoryless Nodes*) accede alla memoria principale ridirezionando le richieste ai due nodi precendenti.

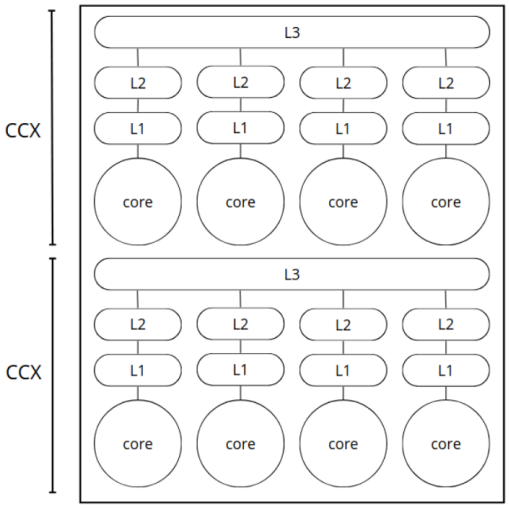


Figura 3.1: Struttura CCD Titanium

Il sistema operativo installato sulla macchina di test è `Ubuntu 22.04`, con kernel `Linux 5.15.0-x86_64`. Il compilatore disponibile è `gcc-12.3.0` e l'allocatore di memoria utilizzato è *jemalloc*[1], un allocatore multi-purpose che garantisce prestazioni efficienti in scenari concorrenti.

3.2 Architettura di Test

Per analizzare le prestazioni delle differenti implementazioni, sono stati progettati tre tipi di test: *ManyToMany*, *ManyToOne* e *OneToMany*, che differiscono principalmente nel numero di produttori e consumatori coinvolti, variando così il livello di concorrenza e interazione tra i processi. Questi scenari permettono di simulare carichi di lavoro con un grado di parallelismo variabile.

Ogni test specifica diversi parametri configurabili:

- **P,C**: numero di produttori e consumatori che accedono alla coda.
- **L**: Nel caso delle code unbounded corrisponde alla lunghezza dei buffer concatenati. Nel caso delle code bounded si riferisce al numero massimo[2] di elementi ospitabili nella coda.

[1] `https://github.com/jemalloc/jemalloc`

[2] le implementazioni Bs-CRQ e Bs-PRQ (Sezione 2.3), utilizzano di una semantica rilassata, discussa in Sezione 2.3.4. **L** indica il numero massimo di elementi senza considerare possibili interferenze tra i thread

- **D**: numero di dati che vengono inseriti ed estratti dalla coda.
- **W**: lavoro non conteso tra operazioni successive.

Per ogni configurazione sperimentale, viene misurato il tempo necessario per il trasferimento di un milione di dati. I dati in questione sono rappresentati da puntatori placeholder, che fungono da simboli temporanei per le entità gestite all'interno dei test ma non corrispondono a zone di memoria allocate. Le uniche operazioni di allocazione e deallocazione di memoria si riferiscono esclusivamente alla creazione e distruzione di nuovi buffer all'interno delle implementazioni delle differenti code.

Tali allocazioni di memoria sono utilizzate per simulare l'esecuzione di code unbounded, in cui la dimensione delle strutture dati può crescere indefinitamente in risposta ai carichi di lavoro. Inoltre, le allocazioni dinamiche si rivelano essenziali per la gestione e il recupero da situazioni di livelock delle code `F&A`-loop based, dove i processi, se non opportunamente gestiti, rischiano di entrare in uno stato di stallo che impedisce il progresso delle operazioni. In tali circostanze, i nuovi buffer sono impiegati per ripristinare l'ordine e riprendere l'esecuzione, consentendo al sistema di continuare a operare in maniera efficiente nonostante i potenziali impedimenti derivanti dalla concorrenza.

Il carico di lavoro di ciascun produttore $D_p \mid \quad p \in [1, P]$ è espresso come una frazione di elementi da inserire in coda a partire da una quantità totale D parametro di test; risulta sempre bilanciato sul numero dei produttori ed è calcolato come segue:

$$D_p = \left\lfloor \frac{D}{P} \right\rfloor + 1_{\{p < (D \bmod P)\}}$$

3.2.1 Random Work

Per simulare il comportamento realistico dei produttori e consumatori durante l'esecuzione dei test, è stato introdotto un concetto di *random work*, che rappresenta un carico di lavoro non conteso tra operazioni consecutive dello stesso tipo. Questo introduce variabilità temporale tra operazioni eseguite dallo stesso thread, creando un effetto di attesa pseudo-randomica che imita situazioni pratiche in cui i produttori e i consumatori non sono sincronizzati perfettamente e le operazioni non avvengono in modo uniforme.

In assenza del random work, la simulazione potrebbe generare risultati che dipendono fortemente da *eventi di contesa* tra i produttori e i consumatori. Questo accadrebbe, per esempio, quando più produttori cercano di inserire oggetti nella coda contemporaneamente o più consumatori cercano di rimuoverli nello stesso momento, ma senza introdurre una *casualità* che riflette l'effettiva variabilità temporale che si verifica nelle situazioni pratiche.

I parametri di *random work* che vengono sottoposti ad analisi sono dell'ordine dei microsecondi. Questi ritardi brevi sono necessari perché aumentando anche

di un solo ordine di grandezza diventa difficile distinguere il comportamento delle diverse implementazioni (Figura 3.6).

I metodi di libreria storici per la misurazione del tempo, come ad esempio `gettimeofday` su Unix, offrono una granularità abbastanza grossolana, tipicamente di diversi microsecondi. Questa precisione è adeguata per misurare operazioni di durata maggiore, come la lettura di un blocco di dati dal disco. Tuttavia, nei benchmark della CPU e della memoria, è necessario misurare operazioni di durata molto breve. In questi casi, l'unica opzione disponibile è quella di contare i cicli di clock.

Per eseguire il *random work*, viene quindi utilizzata l'istruzione `RDTSC`, che su architetture `x86` restituisce il numero di cicli di clock (ticks) trascorsi dall'ultima invocazione dell'istruzione o dal riavvio del sistema. Questa istruzione restituisce il contenuto del registro *Time Stamp Counter*, che viene incrementato ad hardware per ogni tick.

Tuttavia, l'uso di `RDTSC` per misurare il tempo può presentare alcuni problemi. In primo luogo, il valore restituito dipende dalla frequenza del clock che non è fissa. La variabilità può derivare da tecniche che riducono il consumo energetico o adattano dinamicamente la potenza della CPU, complicando la misurazione precisa dei tempi. Inoltre, su sistemi multi-core, ogni processore potrebbe avere una frequenza di clock diversa, il che complica la misurazione precisa e coerente.

Fortunatamente, i processori `x86` più recenti, inclusi quelli della macchina presa in esame, implementano la funzionalità di `constant_tsc` (consultabile dal file `proc/cpuinfo` in sistemi Unix-based), che fa sì che il Time Stamp Counter venga incrementato a una frequenza costante, indipendentemente dalla frequenza operativa della CPU. Questo rende la misurazione del tempo tramite `RDTSC` più affidabile e precisa in questi contesti.

Per calcolare i ticks corrispondenti a un determinato lasso di tempo viene utilizzato un metodo basilare di convergenza numerica per cui si eseguono ripetuti *random work* misurando il tempo di ognuno utilizzando `high_resolution_clock` aggiornando i parametri in ingresso fino a raggiungere il lasso di tempo desiderato (considerata una certa tolleranza).

Per prevenire la presenza di outliers nei risultati dei benchmark proposti, il random work viene sempre eseguito su un intervallo parametrico (specificato da un centro e un'ampiezza). In questo caso specifico si è deciso di considerare intervalli della forma $\left[\frac{c}{2}, \frac{3c}{2}\right]$.

3.2.2 Thread Pinning

Per garantire la coerenza e la prevedibilità dei test eseguiti, sono state utilizzate delle tecniche di thread pinning (affinity). Questo consiste nello specificare un insieme specifico di cores delle CPU su cui uno o più threads vengono vincolati esplicitamente. In questo modo è possibile ridurre considerevolmente l'overhead introdotto dalla gestione dei threads specifica del sistema operativo. La produ-

zione di euristiche di pinning sofisticate permette di ridurre fenomeni di *cache invalidation* che può verificarsi quando un thread viene spostato su un core diverso (*core switch*). La coerenza della cache ha un contributo significativo nella riduzione della latenza dell'accesso alla memoria.

Il *thread pinning* ha un impatto ancora più significativo se si considera l'architettura NUMA del sistema preso in esame. Ogni *NUMA Node* è collegato agli altri tramite una rete di Cache-Coherent Interconnect. In Titanium, su otto nodi solo due hanno accesso diretto alla memoria principale, per cui richieste di allocazione o recupero di dati devono necessariamente passare per questi. Le reti di interconnessioni tra i nodi hanno larghezze di banda diverse, come illustrato nella Figura 3.1, per cui generare un pinning efficiente da questo punto di vista è molto importante per applicazioni che operano su dati condivisi e richiedono costante supporto di allocazione di memoria dinamica.

Node	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	10	16	16	16	28	28	22	28
1	16	10	16	16	28	28	28	22
2	16	16	10	16	22	28	28	28
3	16	16	16	10	28	22	28	28
4	28	28	22	28	10	16	16	16
5	28	28	28	22	16	10	16	16
6	22	28	28	28	16	16	10	16
7	28	22	28	28	16	16	16	10

Tabella 3.1: Distanze tra i NUMA Node su Titanium

Per ottimizzare l'allocazione e l'accesso alla memoria nei vari test, la strategia di thread pinning è stata adattata in base alla topologia NUMA del sistema. La matrice di distanza (3.1) fornisce informazioni cruciali sulla latenza di accesso tra i vari NUMA nodes. Le distanze sono codificate nel firmware nelle tabelle `ACPI SLIT` e rappresentano la latenza di memoria relativa tra i nodi NUMA. (i.e. distanza di 20 indica un tempo di accesso due volte maggiore all'accesso locale al nodo). Questa informazione è stata utilizzata per ridurre i picchi di latenza dovutti ad accessi *cross-node*. Per creare dei pinning NUMA-Aware prendiamo in considerazione dei cluster di nodi. Ogni cluster viene costruito a partire dai memory nodes. Per ogni nodo rimanente, questo viene aggiunto al cluster contenente il memory node per cui la distanza rispetto al nodo è minima. I nodi del cluster vengono poi ordinati in base alla distanza dal relativo memory nodes. In base a questa euristica, è possibile dividere i NUMA nodes di Titanium in due cluster centrati sui nodi 1,5.

Ogni nodo NUMA è composto da otto CPU fisiche divise in due gruppi di quattro che condividono un livello di cache L3. Per massimizzare il traffico cache, i `cpu_id` vengono ordinati sulla base dei rispettivi `cache_id`.

I test prestazionali proposti non spingono il sistema a situazioni di SMT (aka Hyperthreading) dato che per livelli di parallelismo cosi elevati non si ottengono *gain* prestazionali dovuto l'overhead dell'accesso alla memoria. Per ogni processore fisico si considera solo uno dei due processori logici supportati.

L'euristica sopra descritta produce una mappatura che consiste in un insieme di cluster, ciascuno dei quali contiene un gruppo di nodi. Ogni nodo, a sua volta, include gli `cpu_id` dei processori, ordinati in modo da ottimizzare l'utilizzo delle cache condivise.

Partendo dalla mappatura delle distanze, si sviluppa un approccio di pinning, *Pin Cluster*, con l'obiettivo primario di prevenire la formazione di colli di bottiglia. La scelta della sequenza di core è dettata dalla necessità di gestire le diverse modalità di allocazione delle code: le code unbounded richiedono blocchi di memoria contigui, mentre le bounded potrebbero necessitare di allocazioni aggiuntive per la gestione interna dei buffer. L'euristica *Pin Cluster* si basa sul principio di riempire completamente un cluster prima di passare al successivo, massimizzando così l'utilizzo della memoria locale e delle cache L3 condivise tra i core dello stesso nodo NUMA. Questa strategia mira a ridurre gli accessi "cross-nodes", cruciali per minimizzare la latenza in scenari con parallelismo limitato. Tuttavia, quando il parallelismo satura un intero cluster, la sincronizzazione del bus associato alla memoria locale di quel cluster può diventare un punto critico.

Come alternativa, si considera una tecnica iterativa che popola un sottoinsieme di nodi NUMA di un cluster, per poi passare a un altro cluster popolando un numero equivalente di nodi. L'intento è distribuire le allocazioni e gli accessi di memoria su diverse unità di memoria. Sebbene questa strategia possa offrire vantaggi con elevati gradi di parallelismo o in situazioni con frequenti richieste di allocazione di memoria, non è ideale per carichi di lavoro bilanciati. In questi casi, la natura globale delle code porterebbe a un mix di accessi di memoria locali e remoti. I test hanno dimostrato che, per parallelismi contenuti, il costo degli accessi cross-node impone un limite significativo alle prestazioni, mentre per parallelismi elevati le prestazioni tendono a convergere con quelle ottenute con l'approccio *Pin Cluster*.

3.2.3 Implementazioni

I test sono stati eseguiti su varie implementazioni di code MPMC (Multiple Producer, Multiple Consumer). Ogni implementazione è stata selezionata per valutare diversi aspetti delle prestazioni in scenari di carico bilanciato e sbilanciato. Le principali sono:

I test sono stati eseguiti su una gamma limitata di implementazioni di code MPMC (Multi Producer - Multi Consumer). Ogni caso è stato selezionato per valutare diversi aspetti delle prestazioni in scenari di carico bilanciato e sbilanciato. Ogni implementazione si identifica da un nome mnemonico che verrà utilizzato nei commenti delle figure che è accompagnato da uno o due identificativi sinteti-

ci (che si riferiscono alle corrispondente implementazione unbounded e bounded) utilizzato nelle legende.

- **MutexQ** (u-mutex/b-mutex): Utilizza internamente una `std::deque` con operazioni serializzate tramite una mutex globale. La coda può essere configurata come *bounded*, utilizzando un contatore che limita gli inserimenti oltre una certa capacità, oppure come *unbounded*, senza restrizioni sul numero di elementi.

- **FastFlowQ** (linked-ff/ff): Implementa un buffer circolare lock-free basato su `CAS`-loop. La coda può essere configurata come *unbounded* utilizzando *Linked-Adapter* discusso in Sezione 2.7, che costruisce una lista concatenata di buffer lock-free.

- **CRQ** (lcrq/bs-crq): Basato sul lavoro di Afek e Morrison [26], e sull'implementazione fornita da Romanov [9], questa coda utilizza dei buffer circolari `F&A`-based, su cui vengono effettuate operazioni utilizzando `CAS2`, non supportata dal modello atomico di C++. L'uso del buffer in sé soffre di problemi di livelock, come discusso nella Sezione 1.8. Per questo motivo, la coda istanziabile è costruita a partire da modello a lista concatenata di buffer di questo tipo. Oltre alla variante *unbounded*, sono state testate anche varianti *bounded* tramite il wrapper *Bounded-Adapter* discusso nella Sezione 2.3.

- **PRQ** (lprq/bs-prq): Basata sul lavoro e sull'implementazione di Romanov [9], questa coda emula le operazioni di `CAS2` utilizzando `CAS` a singola parola, come descritto nella Sezione 1.3. Per garantire il progresso lock-free, è necessaria una lista concatenata di buffer, sviluppata allo stesso modo del caso precedente.

- **All2All** (a2a): Basata sul modello FastFlow [25], questa implementazione emula una coda MPMC *bounded* tramite una rete di code SPSC (Single Producer, Single Consumer) lock-free. Ogni produttore possiede un vettore di riferimenti a code 1:1 verso ogni consumatore. La politica di scelta della coda su cui inserire l'elemento corrente è di tipo *Round-Robin*.

Nella fase di analisi delle implementazioni sono state effettuate delle modifiche descritte in seguito.

PRQ e CRQ utilizzano un sistema di "chiusura" per ogni buffer, che avviene quando si rileva starvation (secondo un'euristica) o quando il buffer è pieno. La chiusura è unilaterale e non prevede riapertura, causando il fallimento dell'inserimento corrente. In caso di coda *unbounded*, viene allocato un nuovo buffer per garantire l'inserimento. Questa viene gestita tramite un'operazione di `test_and_set` sul bit più significativo dell'indice condiviso, preservando i 63 bit meno significativi. Inizialmente implementata tramite inline assembly `x86`-specific, è stata adattata usando l'istruzione più portabile di `fetch_or` offerta dalla libreria `std::atomic` per l'applicazione di una *bitmask*.

Durante il testing funzionale, è stato osservato che l'implementazione **PRQ** proposta da Romanov [9] presentava inconsistenze, tra cui la restituzione di valori duplicati e errori di estrazioni *out-of-order*, contraddicendo la semantica FIFO della coda. Dopo un'attenta analisi, si è identificata la causa del problema in un bug negli artifacts proposti che provocava letture inconsistenti dei buffer slots. Il problema è stato risolto e segnalato agli autori.

La coda *FastflowQ* adottava una tecnica di *backoff esponenziale* per gestire i `CAS`-faults. Inizialmente, il metodo presentava un errore logico, successivamente risolto. Dopo aver eseguito un fine-tuning dei parametri di backoff, è emerso che, in determinati contesti, la coda superava tutte le alternative, un risultato che inizialmente sembrava controintuitivo. Teoricamente, l'implementazione avrebbe dovuto soffrire di `CAS`-*hotspot*, come discusso in Sezione 1.3, a causa della concentrazione delle operazioni di `CAS` su due parole di memoria: gli indici della coda, indipendentemente dal numero di produttori e consumatori.

La spiegazione di questo comportamento risiede nel fatto che, in un ambiente NUMA fortemente controllato (i.e. considerando un pinning molto aggressivo), i ritardi introdotti dal backoff tendono a causare fenomeni di *cluster segmentation*. Questo ritardo sincronizza le operazioni di inserimento ed estrazione su un sottoinsieme di nodi NUMA. Di conseguenza, i thread su nodi diversi subiscono più frequentemente ritardi a causa del backoff, riducendo il cache-thrashing e permettendo agli altri thread di beneficiare di accessi più rapidi.

Il fine-tuning dei parametri di backoff risultava sempre in situazioni in cui il suo utilizzo non portava a un miglioramento delle prestazioni oppure causava cluster-segmentation non deterministico dato che indotto dall'interleaving delle operazioni. Teoricamente, sarebbe stato possibile implementare un metodo di backoff specifico per ciascuna implementazione. Un esempio di tale approccio è descritto da Afek e Morrison [26], dove viene aggiunto un campo `cluster` alla struttura di *CRQ*, e l'accesso ai metodi innesca esplicitamente un backoff nel caso in cui il nodo da cui un thread accede sia diverso da quello specificato nel campo. In questo caso, viene previsto un metodo di attesa che infine aggiorna il campo cluster per consentire l'accesso. Tuttavia, il parametro di backoff risulta altamente suscettibile a variabili come il carico di lavoro della macchina o la lunghezza del buffer. Pertanto, la ricerca di un parametro di backoff che non svantaggiasse né avvantaggiasse alcuna implementazione avrebbe richiesto un fine-tuning estremamente accurato. Per questa ragione, si è deciso di non adottare alcun tipo di backoff per nessuna coda testata.

3.3 Risutati

In questa sezione vengono presentati i risultati relativi all'analisi delle implementazioni di code lock-free *unbounded* e *bounded*. I grafici che seguono mostrano i dati ottenuti da diverse configurazioni e implementazioni, con l'obietti-

vo di evidenziare le prestazioni, la scalabilità e gli overhead associati a ciascuna soluzione.

Ogni grafico presenta nel titolo informazioni relative allo scenario di contesa, al tipo di pinning dei thread, alla lunghezza del buffer utilizzato e alla durata del *random work*, illustrato in sezione 3.2.1.

Salvo altre indicazioni, ogni esperimento illustrato prevede la trasmissione di 10^6 puntatori (*placeholder*) divisi egualmente tra tutti i produttori.

I risultati relativi al throughput delle code unbounded mostrano come le implementazioni lock-free, rispetto alle tradizionali soluzioni basate su lock, siano in grado di scalare in modo più efficiente con l'aumento del grado di parallelismo. In particolare, queste implementazioni gestiscono la contesa delle risorse in maniera più efficace, riducendo l'overhead dovuto alla sincronizzazione.

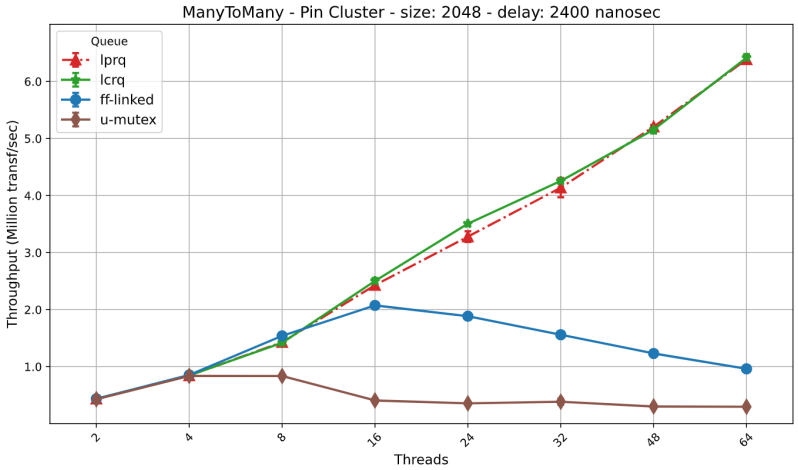


Figura 3.2: Throughput campione di code unbounded

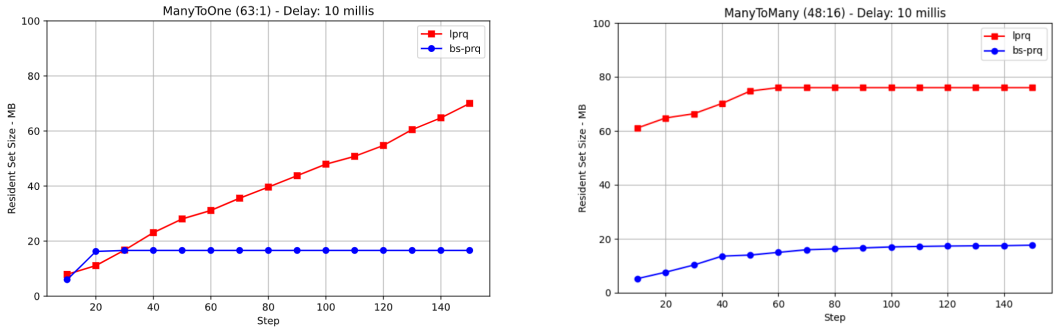
In Figura 3.2 sono presentate le performance in termini di throughput (trasferimenti al secondo) per le diverse implementazioni. Si è scelto di riportare questo risultato con il pinning dei thread alle CPU attivo, nonostante gli effetti del pinning vengano approfonditi in seguito. Il motivo di questa scelta è che il controllo del pinning permette di mettere in evidenza meglio le differenze tra le varie implementazioni, mitigando il fenomeno di core switching e cache thrashing.

Dalla figura, si può osservare che, dopo una prima fase di scalabilità simile per tutte le implementazioni, le uniche che continuano a scalare correttamente sono quelle basate su loop-`F&A`. È evidente che, nell'implementazione lock-based, l'acquisizione e il rilascio di una mutex globale diventi rapidamente un collo di bottiglia. Per quanto riguarda le implementazioni lock-free, si nota che le code `F&A`-loop based superano le `CAS`-loop based con l'aumentare della contesa. Questo perché l'operazione `F&A` consente di distribuire le operazioni atomiche su diverse

locazioni di memoria, riducendo così i fallimenti CAS derivanti dalla contesa e aumentando significativamente l'efficienza del sistema.

Rispetto alle implementazioni LCRQ e LPRQ, si osserva che la loro scalabilità è molto simile, nonostante le differenze nel loro funzionamento. Sebbene entrambe siano basate sullo stesso principio, LCRQ sfrutta operazioni CAS2 direttamente supportate dall'hardware, mentre LPRQ le emula utilizzando una sequenza di tre operazioni CAS a singola parola. Tuttavia, dai risultati emerge che il costo aggiuntivo derivante dall'esecuzione di tre operazioni atomiche è praticamente trascurabile. Questo può essere attribuito al fatto che l'esecuzione di più operazioni atomiche sulla stessa linea di cache non comporta un impatto significativo.

Le implementazioni presentate finora fanno riferimento a strutture dati di tipo *unbounded*. Questo significa che la dimensione di tali strutture si adatta dinamicamente in base al carico di lavoro. Questa caratteristica può risultare accettabile, specialmente per contesti in cui il *workload* è imprevedibile, consentendo alle code di adattarsi agli scenari più disparati. Oltre a questo, l'allocazione dinamica di nuova memoria è necessaria per permettere alle implementazioni F&A-loop based di funzionare correttamente, secondo quanto presentato nelle Sezioni 1.8, 2.1.



(a) Scenario: 63 prod - 1 cons

(b) Scenario: 48 prod - 16 cons

Figura 3.3: Memory Usage: Unbounded vs Bounded

Sebbene l'allocazione dinamica della memoria non sia intrinsecamente negativa, la sua natura incontrollata può risultare problematica in sistemi con requisiti stringenti o carichi di lavoro altamente variabili. Questo aspetto diventa evidente confrontando l'utilizzo della memoria tra implementazioni unbounded e bounded.

In Figura 3.3, si analizza il comportamento di LPRQ e della sua controparte bounded, Bs-PRQ, in due scenari sbilanciati. Il primo (Figura 3.3a), con 63 produttori e un consumatore, mostra come la coda bounded, raggiunto un certo limite, smetta di accettare nuovi inserimenti, stabilizzando il consumo di memoria. Al contrario, la versione unbounded cresce senza controllo, con un aumento esponenziale dell'occupazione di memoria.

Il secondo caso in Figura 3.3b, con 48 produttori e 16 consumatori considera uno scenario più realistico seppur sbilanciato. In questo caso la coda bounded, nonostante un consumo di memoria molto più alto del corrispettivo bounded, si

stabilizzi. Questo è dovuto al caching della memoria dell'allocatore che rispetto al primo caso ha modo di agire maggiormente. L'implementazione bounded si stabilizza più gradualmente rispetto al primo caso dove il consumo di memoria è molto più marcato. Questo è dovuto all'utilizzo di *free-lists* separate per ogni consumatore e dal fatto che un buffer può rimanere allocato, dovuto alla necessità di utilizzare Hazard Pointers, come trattato in sezione 2.3.3.

È importante sottolineare che i valori assoluti registrati sono indicativi e dipendono dai parametri di test; l'attenzione va posta sui trend delle implementazioni.

Il consumo di memoria è misurato tramite il monitoraggio del *Resident Set Size* (RSS), che rappresenta la quantità di memoria fisica utilizzata da un processo. Tuttavia, a causa delle politiche di *lazy reclamation* adottate dai sistemi operativi, il RSS potrebbe non riflettere immediatamente le deallocazioni. Nonostante questa imprecisione, il confronto tra implementazioni bounded e unbounded mostra chiaramente che le prime mantengono un uso più controllato della memoria, mentre le seconde tendono a una crescita incontrollata.

Nel tentativo di limitare l'allocazione di memoria, pur mantenendo un certo livello prestazionale, vengono proposte le implementazioni bounded: Bs-PRQ e Bs-CRQ, esposte nella sezione 2.3.

In questa sezione vengono analizzate le prestazioni di queste implementazioni a confronto con le prestazioni delle varianti unbounded. Questo permette di valutare in termini pratici l'overhead introdotto dalla limitazione della memoria e di delineare dei casi di studio che si applicano ad entrambi i tipi di implementazione.

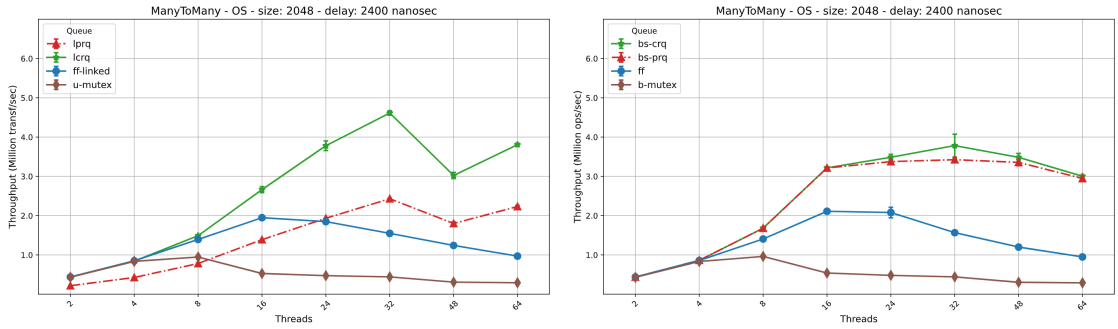
Per questi confronti, consideriamo ogni coda bounded di dimensione l costruita come una lista concatenata di buffer di dimensione $\frac{l}{4}$. In un dato momento, il numero di buffer allocati è compreso tra 1 e 5[3], determinando una dimensione massima della coda pari a $\frac{5l}{4}$.

Questa scelta si è rivelata un buon compromesso tra la necessità di ri-inizializzare i buffer a causa di fenomeni di starvation e la limitazione della frammentazione interna ai buffer concatenati, discussa in sezione 2.3.2.

3.3.1 Caso di Studio: Many To Many

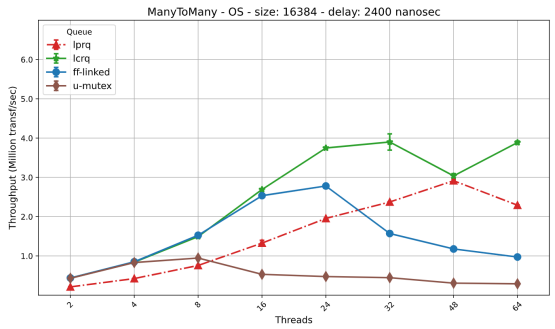
Per valutare le prestazioni delle varie implementazioni in contesti ad alta concorrenza, viene analizzato il caso di studio Many To Many. In questo scenario, un numero uguale di produttori e consumatori interagiscono simultaneamente con la coda, stressandone la capacità di gestione della memoria e delle operazioni di accesso concorrente.

[3]le implementazioni bounded possono eccedere la dimensione dichiarata in fase inizializzazione di un limite massimo fissato 2.3.4

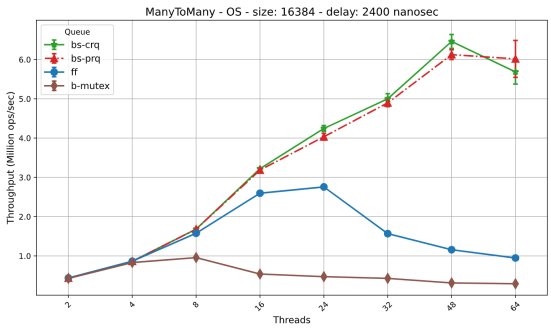


(a) Unbounded

(b) Bounded



(c) Unbounded



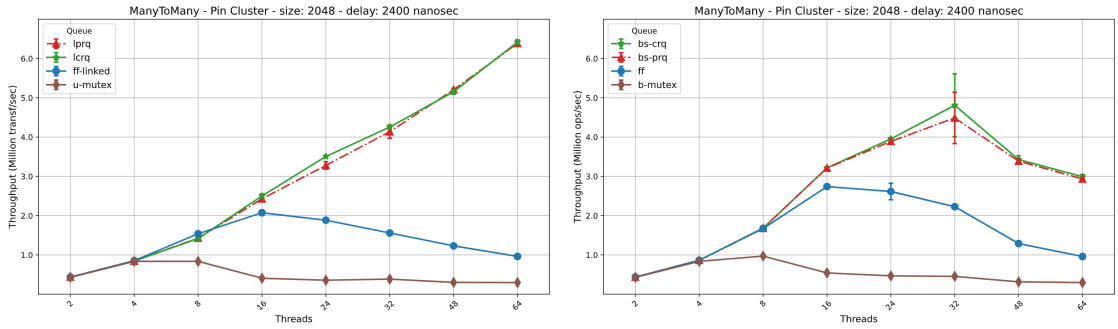
(d) Bounded

Figura 3.4: Caso ManyToMay (#prod = #cons) al variare del buffer size.

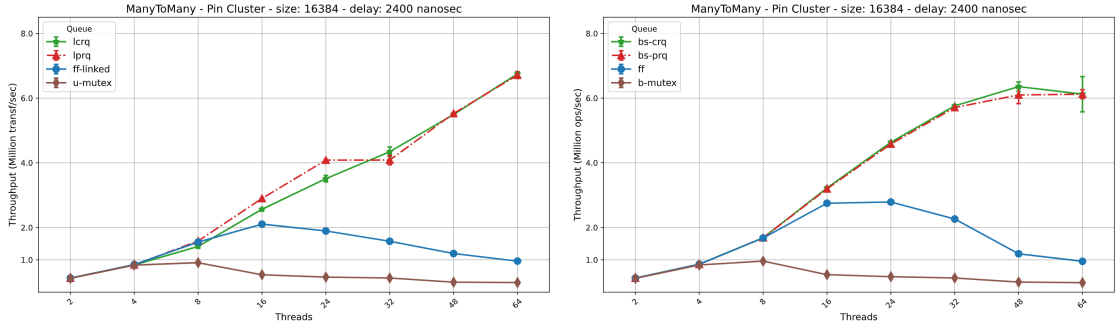
La Figura 3.4 presenta due gruppi di risultati, distinti in base alla lunghezza delle code considerate per le implementazioni. Questo caso considera scheduling del sistema operativo. In particolare, i grafici 3.4a e 3.4b si riferiscono a code di lunghezza 2^{10} elementi, mentre i casi 3.4c e 3.4d analizzano code con una lunghezza pari a 2^{16} elementi.

Nei casi unbounded è possibile osservare un divario netto tra le implementazioni LPRQ e LCRQ, con un evidente vantaggio prestazionale a favore della seconda. L'overhead introdotto da LPRQ risulta trascurabile quando le tre istruzioni di `CAS`, utilizzate per emulare la `CAS2` di LCRQ (trattato in Sezione 2.1.3), operano sulla stessa linea di cache. Questa differenza potrebbe suggerire che il cache thrashing sia la causa principale del divario prestazionale; tuttavia, se così fosse, l'effetto dovrebbe manifestarsi anche nelle varianti bounded. Al contrario, le prestazioni di Bs-PRQ e Bs-CRQ risultano abbastanza simili. La gestione della memoria diventa in questo caso un fattore limitante presente nel caso dove le code sono composte da buffer da 2^{10} slots (Figura 3.4a) accentuato nel caso di buffer da 2^{16} slots ((Figura 3.4c)) considerata la memory pressure più elevata nel caso unbounded, mentre nelle code bounded ha una pressione trascurabile perchè fortemente limitata.

La coda `CAS-loop` (ff in figura) mostra una curva a campana netta in entrambi i casi bounded che unbounded. Questo valida la tesi secondo cui la contesa data dai fenomeni di `CAS`-hotspot sia la causa del degrado prestazionale per parallelismi medio-alti.



(a) Unbounded (b) Bounded



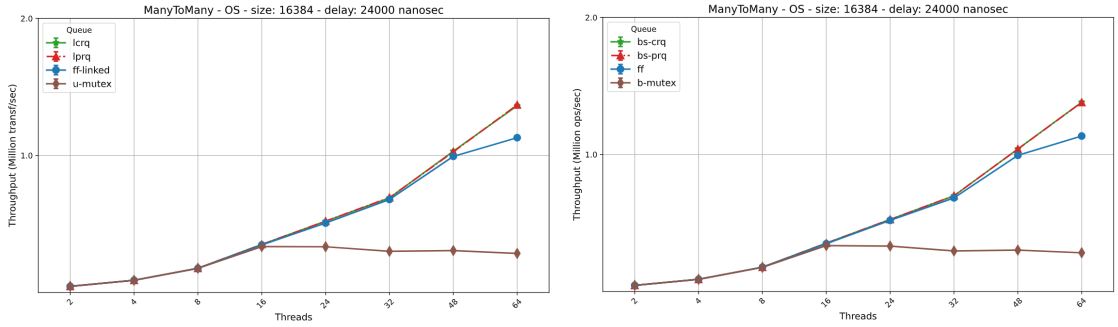
(c) Unbounded (d) Bounded

Figura 3.5: Caso ManyToMany al variare di buffer size: pinning attivo

Gli stessi scenari Many-To-Many sono riproposti in Figura 3.5 attivando il pinning dei thread alle CPU. In questo caso si osserva come indipendentemente dalla lunghezza dei buffer, le code LCRQ e LPRQ presentano un andamento simile. Inoltre, in Figura 3.5c emerge un fenomeno di iperscalabilità per LPRQ nel range di parallelismo compreso tra 8 e 32 thread, indicando che l'overhead dovuto a più operazioni di `CAS` sulla stessa linea di cache è inferiore rispetto a quello introdotto dall'uso di `CAS2`.

Si nota, inoltre, che le implementazioni bounded, pur traendo vantaggio dal pinning, risultano più resilienti rispetto alle alternative unbounded in assenza di tale meccanismo, dovuto al fatto che il pinning sul sistema NUMA utilizzato in fase di test permette di ottenere un overhead più limitato nelle allocazioni continue di memoria.

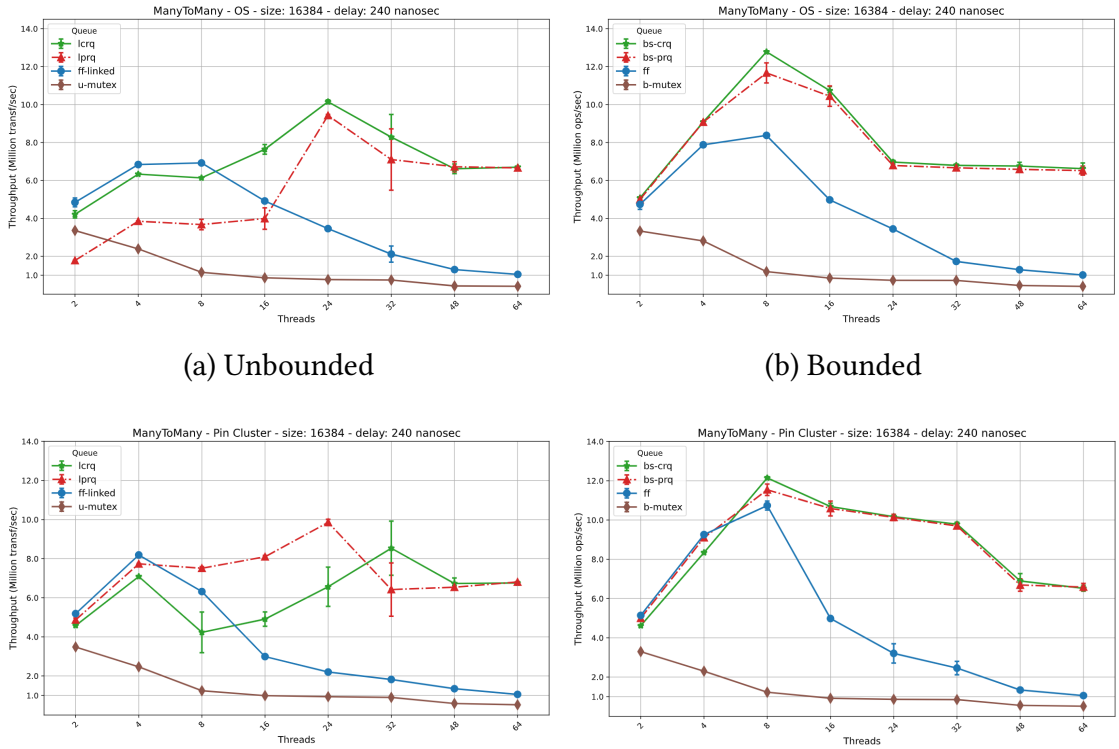
In entrambi i casi, sia con il pinning attivato sia con il normale scheduling del sistema operativo, si osserva che tutte le code basate su `F&A`-loop superano in prestazioni l'alternativa basata su `CAS`-loop (ff in figura).



(a) Unbounded (b) Bounded

Figura 3.6: Caso ManyToMany con random work elevato

La Figura 3.6 mostra che aumentando di un ordine di grandezza il random work (Sezione 3.2.1) associato a ciascuna operazione, l'overhead legato alla gestione della coda diventa pressoché trascurabile. Sebbene la differenza tra le soluzioni mutex-based e lock-free rimanga evidente, l'impatto degli CAS-hotspot si riduce al punto da diventare trascurabile.



(a) Unbounded (b) Bounded

(c) Unbounded: Pinning (d) Bounded: Pinning

Figura 3.7: Caso ManyToMany con random work ridotto: confronto tra pinning e OS scheduling

Diminuendo il random work di un ordine di grandezza (Figura 3.7), i trend

principali evidenziati in Figura 3.4 risultano ancora più marcati. In particolare, per l'implementazione CAS-loop si osserva un netto peggioramento delle prestazioni, mentre nel caso delle implementazioni F&A-loop la differenza tra LCRQ e LPRQ diventa più pronunciata a causa del cache thrashing, che risulta più stabile nello scenario bounded.

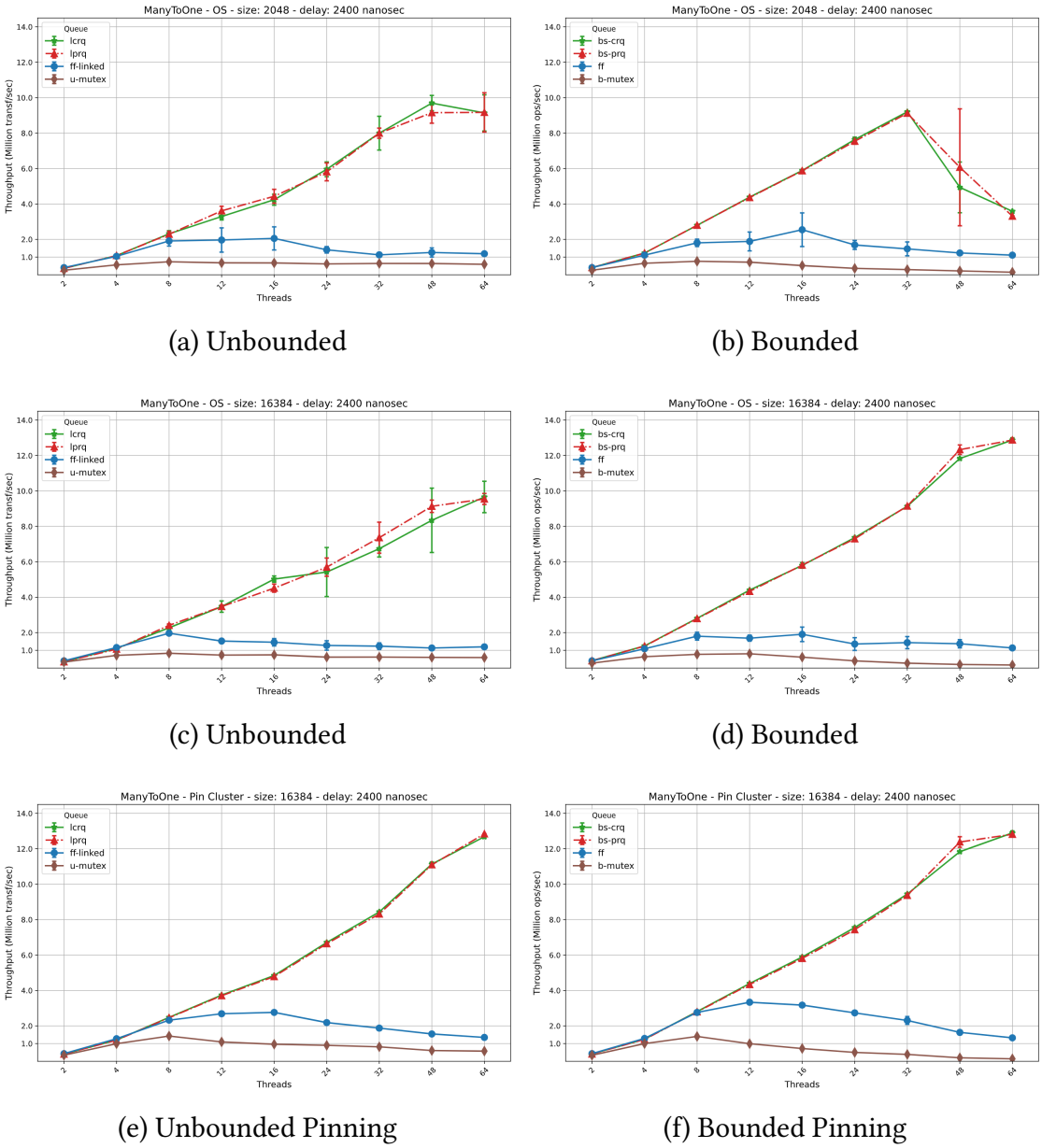
Con un delay così ridotto, è possibile notare che, nelle implementazioni bounded, le prestazioni delle code F&A-loop tendono a stabilizzarsi per parallelismi superiori a 24. Inoltre, anche nel caso peggiore, in cui la memoria non venisse sottoposta a caching, le code unbounded continuano a scalare fino a stabilizzarsi a parallelismo 32. Questo suggerisce che l'overhead non sia dovuto all'accesso diretto alla memoria.

L'ipotesi più plausibile è che, nonostante sia numero di produttori e consumatori che la quantità di lavoro simulato che compiono sia bilanciato, le operazioni di estrazione risultina più onerose rispetto a quelle di inserimento. Ciò è dovuto al fatto che, quando un consumatore effettua una transizione da un buffer al successivo perché il primo è vuoto, quest'ultimo può non essere immediatamente deallocato. Per ridurre la pressione sulla memoria, lo stesso consumatore tenterà di deallocare il buffer nelle estrazioni successive (Sezione 2.3.3). Poiché le code unbounded non applicano questo meccanismo il costo medio delle operazioni di inserimento ed estrazione rimane più bilanciato, giustificando così la scalabilità.

Sono riportati anche gli stessi scenari attivando il pinning dei thread alle CPU. Si nota come le code bounded siano resilienti in attesa di questo meccanismo dai confronti delle Figure 3.7b e 3.7d dove lo scenario è lo stesso ma nel primo caso il pinning risulta disabilitato. Lo stesso confronto, effettuato sulle code unbounded nelle Figure 3.7a e 3.7c dove nel caso delle code LCRQ e LPRQ i risultati sono ribaltati. Nel caso con pinning disattivato LCRQ utilizzando istruzioni di CAS2 è molto meno interessata dal cache thrashing dovuto all'OS scheduling diversamente da LPRQ in cui l'emulazione CAS2 ne risente. Attivando il pinning invece LPRQ performa molto meglio, mentre LCRQ performa peggio del caso senza pinning, dovuto al *bus-locking* necessario per garantire l'atomicità delle CAS2, che introduce sicuramente più overhead di quello effettuato per CAS. Dato che attivando il pinning si riduce il cache thrashing, LPRQ ha scalabilità più elevata del caso senza pinning.

3.3.2 Caso di Studio: Many To One

Si considera il caso di studio Many To One, in cui molteplici produttori inseriscono elementi nella coda mentre un singolo consumatore li estrae, mettendo particolarmente sotto stress la gestione della sincronizzazione tra i thread e l'efficienza delle operazioni di accesso concorrente. Questo scenario evidenzia eventuali colli di bottiglia dovuti alla contesa sulle risorse e all'overhead delle operazioni atomiche. Per mitigare lo sbilanciamento tra produttori e consumatore, il lavoro simulato tra le operazioni viene eseguito esclusivamente tra inserimenti successivi.



(a) Unbounded

(b) Bounded

(c) Unbounded

(d) Bounded

(e) Unbounded Pinning

(f) Bounded Pinning

Figura 3.8: Caso ManyToOne al variare del buffer size: confronto tra OS scheduling e pinning

La Figura 3.8 mostra due scenari su cui vengono valutate le implementazioni. Come il caso precedente, per ogni caso specifico si ritrovano sulla sinistra, le prestazioni delle code unbounded e sulla destra quelle delle code bounded per gli stessi parametri. Viene riportato anche un caso di test eseguito con l'attivazione del pinning dei thread in figura 3.8e confrontabile con il caso in figura 3.8c.

In tutte le figure si nota la differenza di prestazioni tra le implementazion `F&A`-loop e le alternative. Le prime, sia nella versione bounded che unbounded, mostrano una scalabilità quasi doppia rispetto ai test Many To Many della Figura 3.4. Al contrario, l'implementazione `CAS`-loop non beneficia dello stesso miglioramento, con una scalabilità invariata o addirittura peggiorata.

Questo si deve al fatto che l'overhead di concorrenza tra diverse operazioni di inserimento nelle code F&A è trascurabile, dato che gli inserimenti dei valori di controllo tramite CAS vengono distribuiti. Nel caso di semplice CAS-loop invece, tutte le operazioni di inserimento effettuano CAS sulla stessa locazione di memoria, quindi al crescere del parallelismo la contesa aumenta.

La Figura 3.8b mostra un calo delle prestazioni in entrambe le code bounded F&A-based a parallelismo 48, con un'elevata deviazione standard, e un ulteriore peggioramento a parallelismo 64. In questo scenario, la coda può contenere al massimo 2560 elementi[4]. Poiché questo fenomeno non si osserva nel caso della Figura 3.8d dove la dimensione della coda è quadruplicata, il calo di prestazioni è attribuibile alla dimensione limitata della coda, che, una volta piena, provoca lo stallo dei produttori. Anche considerando questo, le prestazioni rimangono ben maggiori rispetto all'alternativa ff.

3.3.3 Caso di Studio: One To Many

Si analizza ora il caso di studio One To Many, in cui un unico produttore inserisce elementi nella coda, mentre molteplici consumatori li estraggono in parallelo. Questo scenario mette alla prova l'efficienza nella distribuzione del carico tra i consumatori e la capacità della coda di gestire accessi concorrenti delle operazion idi estrazione. In particolare, vengono esaminati i potenziali colli di bottiglia legati alla sincronizzazione tra i consumatori e l'impatto delle operazioni atomiche sulle prestazioni complessive. Per ridurre gli effetti della contesa, il lavoro simulato viene eseguito solo tra operazioni di estrazione successive.

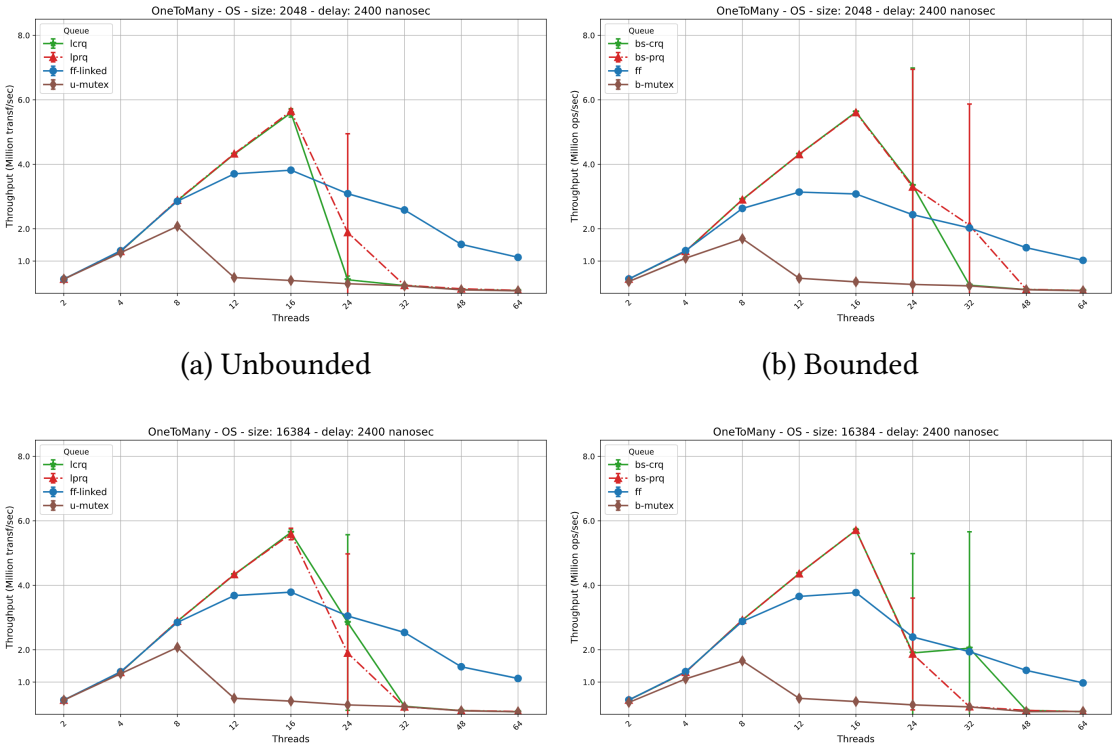
Questo scenario rappresenta una sfida per le implementazioni F&A-loop based, poiché i segmenti interni a tutte le varianti risultano suscettibili al livelock in situazioni di starvation dei produttori, come discusso nella Sezione 1.8.

Per prevenire il livelock, il meccanismo di recupero prevede la creazione di un nuovo segmento, che viene poi concatenato alla lista dei segmenti esistenti. Questo garantisce che l'inserimento dell'elemento avvenga con successo nel momento in cui il nuovo segmento viene collegato alla struttura dati. Sebbene tale meccanismo assicuri il progresso e prevenga il blocco indefinito delle operazioni, l'overhead introdotto dalla continua aggiunta e rimozione di segmenti risulta significativo.

All'aumentare della contesa tra consumatori, la frequenza con cui nuovi segmenti vengono creati e rimossi può crescere considerevolmente, peggiorando le prestazioni complessive. Questo comportamento si riflette in un incremento dell'overhead sulla memoria e della contesa in sincronizzazione, portando a un degrado dell'efficienza rispetto ad altri approcci meno sensibili a fenomeni di starvation e contesa.

[4]Essendo composta da buffer di 512 elementi e potendo allocare fino a quattro buffer più un quinto per il vincolo rilassato sulla memoria (Sezione 2.3.4), la dimensione massima raggiungibile è 2560 elementi

Questo caso evidenzia il trade-off delle implementazioni basate su F&A-loop, le quali, anche nel contesto bounded, presentano una gestione della memoria poco intuitiva. E' difficile imporre dei limiti rigidi sull'allocazione di memoria, pur consentendo di mantenerla limitata adottando dei bound rilassati.



(a) Unbounded (b) Bounded

(c) Unbounded (d) Bounded

Figura 3.9: Caso OneToMany al variare del buffer size

Nel caso delle implementazioni F&A-loop unbounded, l'andamento è inizialmente lineare fino a parallelismi di 16, ma al crescere del numero di thread si osserva un rapido declino delle prestazioni. A partire da parallelismi 24 e 32, la deviazione standard aumenta notevolmente, indicando una crescente variabilità nelle prestazioni e un forte stress sulla struttura dati. Questo suggerisce che, nonostante la capacità di scalare inizialmente, le implementazioni texttttF&A-loop sono sensibili a fenomeni di contesa e starvation, i quali introducono inefficienze quando il numero di consumatori aumenta significativamente. Il calo di prestazioni è particolarmente marcato in corrispondenza di parallelismi più elevati, dove la gestione della sincronizzazione tra i consumatori e l'accesso concorrente alla coda diventa sempre più complessa e costosa.

Nei casi bounded, nonostante l'inserimento di limiti sulla memoria e la gestione più controllata della coda, si osserva un comportamento simile, ma con una maggiore stabilità. Tuttavia, anche in queste configurazioni, l'efficienza diminuisce progressivamente al crescere del parallelismo, indicando che la gestione della memoria e la sincronizzazione non sono completamente prive di overhead,

soprattutto quando il sistema viene sottoposto a carichi elevati. Dopo i parallelismi 32, le prestazioni si stabilizzano su livelli molto più bassi rispetto ai casi con parallelismi più contenuti.

Il comportamento osservato in questi test evidenzia un trade-off significativo nelle implementazioni `F&A`-loop. Se da un lato queste offrono una buona scalabilità in scenari a carico di lavoro bilanciato o sbilanciato in favore dei produttori, dall'altro lato sono soggette a un rapido degrado delle prestazioni quando il numero di thread aumenta, specialmente in scenari altamente sbilanciati come il caso One To Many proposto. Questo comportamento si manifesta soprattutto nelle versioni unbounded, ma anche in quelle bounded, la gestione delle risorse e la sincronizzazione continuano a costituire un fattore critico nella definizione delle prestazioni a parallelismi elevati.D'altro canto, le varianti mutex-based tendono a mostrare prestazioni più prevedibili e stabili, sebbene con una scalabilità inferiore rispetto alle implementazioni `F&A`-loop nelle fasi iniziali.

3.3.4 Caso di Studio: All2All

L'idea di buffer SPSC (*Single Producer Single Consumer*) è stata ampiamente esplorata nel corso del tempo, costituendo un elemento fondamentale per costruire sistemi di comunicazione concorrente efficienti. In particolare, Lamport [30] ha dimostrato come, sotto la supposizione di *Sequential Consistency* (trattato in Sezione 1.5), non siano necessari meccanismi di sincronizzazione esplicita per garantire la consistenza delle operazioni su tali buffer. Successivamente, Higham et al. [31] hanno proposto delle modifiche basate sull'atomicità intrinseca delle singole operazioni di `STORE`, una caratteristica garantita da tutti gli hardware moderni. Questa osservazione permette di realizzare buffer FIFO SPSC senza la necessità di ricorrere a complesse operazioni `CAS` per l'inserimento e la rimozione dei valori, semplificando notevolmente l'implementazione. Partendo da questa efficiente e semplice realizzazione di buffer SPSC, è possibile costruire approcci alternativi per la comunicazione Multi-Producer Multi-Consumer (MPMC). Un caso interessante da considerare è il modello denominato *All2All* [25]. L'idea alla base di All2All è quella di emulare una comunicazione MPMC minimizzando la condivisione di risorse, al fine di ridurre l'overhead dovuto al *cache-coherent traffic*. Per raggiungere questo obiettivo, il modello proposto utilizza una rete di buffer SPSC wait-free, dove viene creato un buffer dedicato per ogni coppia produttore-consumatore. Questo approccio genera una coda distribuita composta da $|P| \times |C|$ buffer circolari SPSC.

Un aspetto positivo di All2All risiede nell'avere buffer dedicati per ogni coppia di comunicatori. Se si adottano accorgimenti particolari, questa struttura può diminuire significativamente la *cache invalidation*, portando a un miglioramento del *throughput* complessivo del sistema. Inoltre, l'esistenza di un canale dedicato permette di indirizzare i dati a specifici consumatori, aprendo la strada a pattern di comunicazione più complessi e mirati.

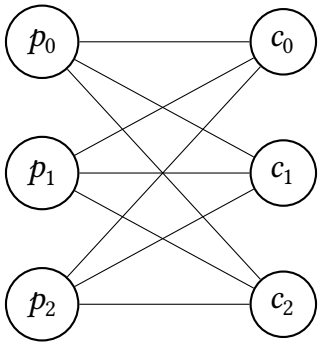
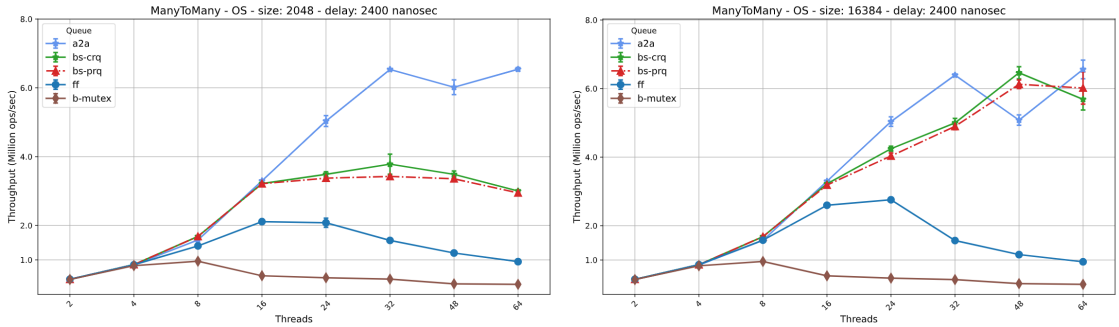


Figura 3.10: Struttura di All2All (3:3)

Tuttavia, l'approccio All2All presenta anche dei fattori negativi. Uno di questi è la potenziale frammentazione della memoria, che spesso richiede l'aggiunta di *padding* tra i buffer per minimizzare il fenomeno del *false sharing*. Un'altra criticità è la necessità di implementare un algoritmo di *scheduling* per gestire la distribuzione dei dati tra i vari buffer e consumatori. Nelle code MPMC tradizionali, questo problema è intrinsecamente gestito dal fatto che la coda è una risorsa globale a cui i consumatori accedono in base alla propria velocità di elaborazione. È opinione comune che gli algoritmi di *scheduling* che tengono conto del livello applicativo siano i più efficaci, il che rende ogni euristica generalizzata probabilmente subottimale per contesti specifici. Un ulteriore svantaggio significativo è che il numero massimo di produttori e consumatori supportati deve essere conosciuto in anticipo e non può essere modificato dinamicamente nel tempo, poiché i buffer che costituiscono la coda sono preallocati.

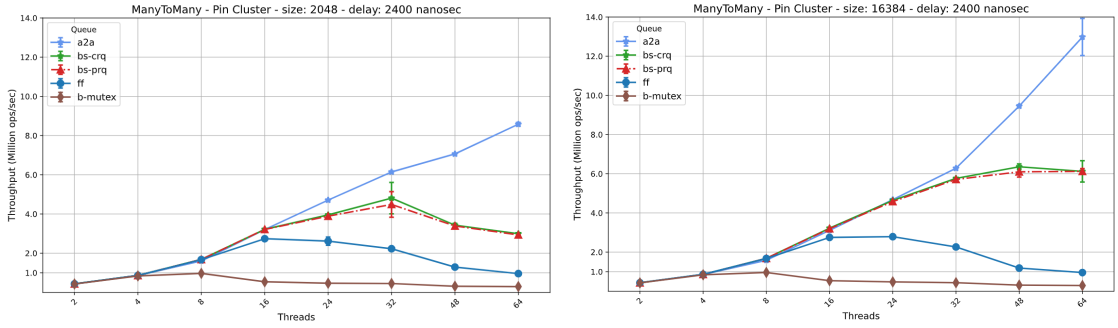
Nonostante un approccio All2All non si possa sostituire universalmente alla necessità di un canale di comunicazione condiviso, è utile analizzare le prestazioni per valutare i casi specifici.

Nei seguenti risultati si comparano le prestazioni di All2All contro tutti i casi già trattati di code bounded. Dato che nel modello All2All i buffer sono distribuiti tra le coppie produttore consumatore, per confrontare le capacità delle code è necessario definire una metrica comune. In questo caso, si considera la somma degli slot di tutti i buffer allocati come una lunghezza equivalente ai casi MPMC. Questa scelta è dettata dalla necessità di confrontare una struttura dati distribuita con code tradizionali che possiedono una singola capacità massima. Al variare del parallelismo, varia di conseguenza il numero di buffer allocati nel modello All2All. Pertanto, dati un numero di slot fissato n che rappresenta la capacità totale da "simulare", e un numero di thread produttori e consumatori p e c rispettivamente, la lunghezza di ciascun buffer SPSC nella corrispondente implementazione All2All sarà di $\left\lceil \frac{n}{pc} \right\rceil$. Tutti i risultati riportati utilizzano random work (Sezione 3.2.1)



(a) buffer size: 2^{10} (b) buffer size: 2^{16}

Figura 3.11: Caso ManyToMay con scheduling OS



(a) buffer size: 2^{10} (b) buffer size: 2^{16}

Figura 3.12: Caso ManyToMay con pinning attivo

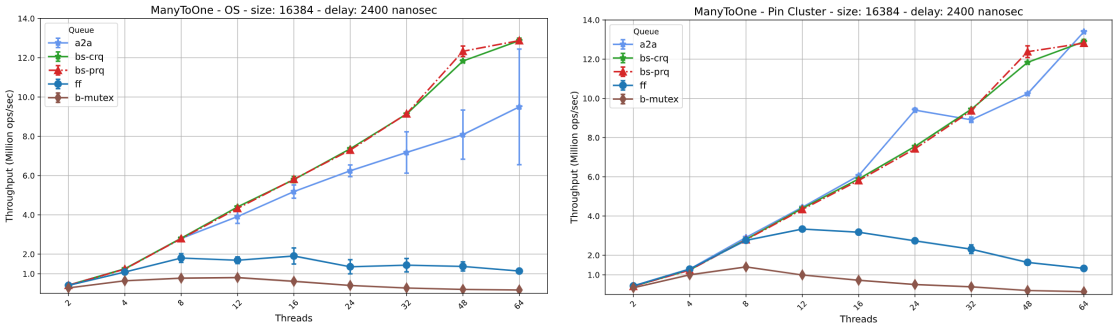
Nelle Figure 3.11 e 3.12 è mostrato il confronto tra le prestazioni dell'implementazione All2All e delle code MPMC nei contesti di OS scheduling e pinning. Dai grafici si osserva che All2All riesce a eguagliare o superare le performance delle code MPMC, con un vantaggio particolarmente evidente nel caso di pinning attivo.

La differenza nelle prestazioni tra OS scheduling e pinning nell'implementazione All2All è principalmente dovuta alla sensibilità al cache thrashing. Il punto di forza di All2All sta nella minimizzazione della condivisione delle risorse: mentre nelle code MPMC esiste una coda condivisa tra tutti i produttori e i consumatori, in All2All ogni produttore ha un canale di comunicazione dedicato con ogni consumatore.

In scenari con workload bilanciato, dove i buffer tra produttori e consumatori non vengono saturati frequentemente, i produttori tendono a inviare dati sempre allo stesso consumatore. Questo comportamento riduce significativamente il cache thrashing, poiché la probabilità che i dati vengano scritti su linee di cache diverse (e quindi vengano invalidate frequentemente) è notevolmente ridotta. Questo meccanismo di comunicazione diretta consente di ottenere prestazioni più scalabili, in particolare con il pinning attivo, che riduce ulteriormente la frequen-

za di invalidazione delle linee di cache. Questo risulta molto importante data l'architettura NUMA del sistema dove il *cache coherent traffic* è ancora più critico.

Al contrario, nelle code MPMC, la condivisione della coda tra produttori e consumatori rende il sistema più resistente al pinning disattivato, poiché la distribuzione dei carichi di lavoro tra i core è più uniforme, e il cache thrashing avviene indipendentemente da fenomeni di *core switch* data la condivisione della coda tra tutti i thread.

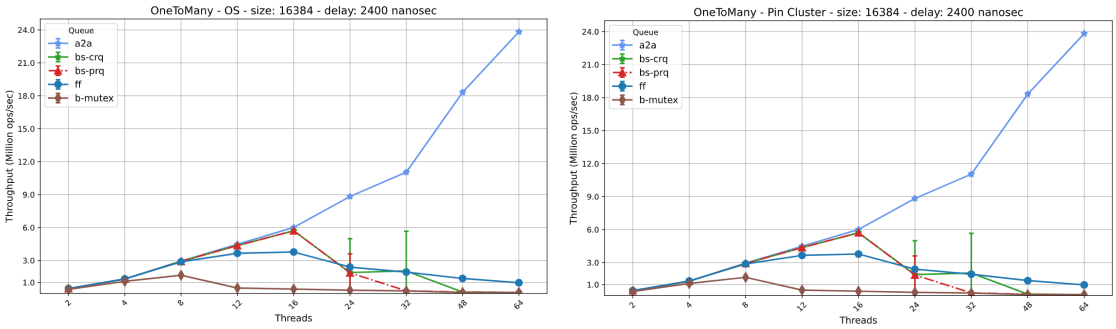


(a) OS scheduling (b) pinning attivo

Figura 3.13: Caso ManyToOne: confronto tra pinning e OS scheduling

In Figura 3.13 sono mostrati i risultati relativi a uno scenario Many to One, con buffer di lunghezza pari a 2^{16} per ogni implementazione, confrontando i casi di OS scheduling e pinning attivo. Nei grafici relativi a OS scheduling, si nota che le prestazioni di All2All mostrano una crescente deviazione standard all'aumentare del parallelismo. Questo comportamento è dovuto al fatto che, in presenza di uno scheduler *non-cache-aware*, l'unico consumatore potrebbe essere soggetto a frequenti *core switches*, causando l'invalidazione dei riferimenti ai canali di comunicazione. Di conseguenza, il sistema soffre di *cache thrashing* che riduce le prestazioni, in quanto le linee di cache vengono invalidate ogni volta che il consumatore è migrato su un nuovo core.

Quando il pinning è attivo, le prestazioni di All2All migliorano notevolmente. Il pinning impedisce il core switch, riducendo la frequenza di invalidazione delle linee di cache e migliorando l'efficienza dell'accesso ai buffer. Questo approccio, soprattutto in scenari con elevato parallelismo, permette di mantenere l'*affinity* tra il consumatore e il core su cui è eseguito, migliorando le prestazioni del sistema. In questo caso le prestazioni risultano paragonabili alle implementazioni `F&A`-based che si dimostrano estremamente scalabili in questo scenario nonostante la condivisione globale della coda.



(a) OS scheduling (b) pinning attivo

Figura 3.14: Caso OneToMany: confronto tra pinning e OS scheduling

In Figura 3.14, vengono mostrati i risultati relativi a uno scenario One to Many, sempre con buffer di lunghezza pari 2^{16}. In questo caso, All2All mostra prestazioni significativamente migliori rispetto al caso Many to One (Figura 3.13), sia con OS scheduling che con pinning attivo. La differenza nelle prestazioni è dovuta a un bilanciamento più efficiente dei carichi di lavoro. Nel modello One to Many, c'è un solo produttore che invia i dati a molti consumatori, il che consente una distribuzione più equa dei dati tra i consumatori. In questo modo, ogni consumatore riceve dati da un singolo produttore e può elaborare i dati in modo indipendente senza il rischio di diventare un collo di bottiglia, come nel caso di Many to One.

Anche in presenza di OS scheduling, le prestazioni non sono influenzate significativamente dalla disattivazione del pinning, poiché la comunicazione tra produttori e consumatori avviene attraverso canali dedicati, riducendo il rischio di cache thrashing. Inoltre, il fatto che ciascun consumatore lavori in modo indipendente su un flusso dedicato di dati riduce la competizione per le risorse e minimizza gli effetti negativi dovuti a cambi di core. Quando il pinning è attivo, le prestazioni di All2All migliorano ulteriormente grazie alla riduzione del traffico di cache coherence (caché incoerente), che potrebbe altrimenti compromettere l'efficienza in scenari di elevato parallelismo.

Allo stesso scenario le code MPMC proposte non beneficiano della stessa efficienza. Mentre l'implementazione `CAS`-loop è interessata da fenomeni di *hotspot*, le implementazioni `F&A` based necessitano di un riciclo continuo dei segmenti per garantire progresso in possibili situazioni di livelock dato il carico altamente sbilanciato.

In sintesi, lo scenario One to Many si dimostra altamente scalabile, con All2All che riesce a sfruttare al meglio le risorse di sistema in presenza di un solo produttore e più consumatori. La separazione dei canali di comunicazione e il minor carico sul consumatore contribuiscono a ottenere prestazioni superiori rispetto a Many to One, dove il consumatore singolo deve gestire tutti i dati provenienti da più produttori, creando un collo di bottiglia.

3.4 Considerazioni

In questo capitolo abbiamo presentato una valutazione comparativa delle prestazioni di code FIFO, sia in versione bounded che unbounded. Le code sono state realizzate attraverso estensioni che impiegano buffer circolari, generando strutture dati a capacità dinamica tramite un modello a lista concatenata descritto nel dettaglio nella sezione 2.2. Anche le implementazioni bounded sono state costruite utilizzando una variante del modello precedente, descritto in Sezione 2.3 dato che l'utilizzo di buffer circolari `F&A`-based non è sufficiente per contrastare fenomeni di stallo attivo (livelock), discusso in Sezione 1.8. L'analisi ha contemplato il testing con carichi di lavoro equilibrati e sbilanciati tra i vari thread produttori e consumatori.

Per ampliare la prospettiva, sono state incluse comparazioni prestazionali tra code implementate utilizzando segmenti lock-free basati su operazioni `F&A` (Fetch and Add), `CAS` loop e approcci basati su mutex.

Le analisi condotte evidenziano come, nella stragrande maggioranza, tutte le soluzioni lock-free superino significativamente le alternative basate su mutua esclusione. In contesti di carico bilanciato o di sovrapproduzione, le implementazioni basate su `F&A` mostrano una scalabilità notevolmente superiore rispetto alle altre realizzazioni, grazie alla distribuzione dell'overhead associato alle operazioni `CAS` che garantisce bassa contesa nelle operazioni di inserimento ed estrazione. Tuttavia, queste ultime richiedono un'allocazione continua di nuovi buffer in scenari di starvation dei produttori, al fine di assicurare una garanzia di progresso che prevenga il livelock interno ai segmenti. In situazioni estreme e fortemente asimmetriche (Sezione 3.3.3), tali soluzioni non si dimostrano pienamente affidabili e, nonostante la presenza di una garanzia di progresso, la velocità di elaborazione delle operazioni (throughput) risente di questa condizione.

Nella maggioranza dei test effettuati, emerge chiaramente che le code bounded offrono prestazioni comparabili alle versioni unbounded, evidenziando come le restrizioni necessarie per ottenere strutture a capacità fissa introducano un overhead trascurabile.

Abbiamo inoltre analizzato gli effetti del pinning dei thread per valutare la sensibilità al cache thrashing in relazione alla gestione del piazzamento dei thread offerta dal sistema operativo. Dai test svolti è emerso che le implementazioni unbounded risultano più vulnerabili a questo fenomeno, a causa dei pattern di allocazione continua della memoria che impediscono di sfruttare a pieno il caching dell'allocatore di memoria. Al contrario, le implementazioni bounded si sono dimostrate molto resilienti, tanto che le prestazioni in scenari con pinning attivo risultano comparabili a quelle ottenute con il normale scheduling del sistema operativo.

Infine, sono state eseguite ulteriori analisi che comparano le prestazioni delle implementazioni proposte con una versione alternativa basata sul pattern All2All (Sezione 3.3.4) che utilizza un network di buffer singolo produttore singolo con-

sumatore (SPSC) per emulare le funzionalità di una coda MPMC. Sebbene questa soluzione imponga scarsa flessibilità in casi di numero di processi variabili[5], mostra una scalabilità comparabile o migliore alle implementazioni `F&A`-based in diversi casi. Questo si ritiene sia dovuto al minore *sharing di risorse* e conseguente miglioramente delle prestazioni della gerarchia di memoria.

[5] ogni coppia di produttori e consumatori necessita di un buffer dedicato

Conclusioni

Questa tesi mi ha offerto l'opportunità di approfondire il mondo degli algoritmi concorrenti, con un focus particolare sulle strutture dati non bloccanti, permettendomi di apprezzarne le potenzialità in un contesto in cui la scalabilità e l'efficienza computazionale sono cruciali. Lo studio condotto ha fornito evidenze empiriche su come l'adozione di strutture dati non bloccanti, in alternativa ai tradizionali meccanismi di sincronizzazione, rappresenti un passo significativo verso il miglioramento delle prestazioni nei sistemi ad alta concorrenza. La riduzione dell'overhead dovuto alla sincronizzazione e la capacità di sfruttare al meglio i sistemi multi-core si traducono in un incremento dell'efficienza complessiva delle applicazioni concorrenti.

Il lavoro svolto propone una metodologia per implementare code FIFO bounded lock-free, traendo ispirazione dalle tecniche allo stato dell'arte adottate per le code lock-free unbounded. Il risultato è stato la possibilità di ottenere code con capacità predeterminata e consumo di memoria limitato, mantenendo in molti casi i benefici prestazionali delle soluzioni unbounded. I test svolti su una piattaforma multicore NUMA messa a disposizione dall'Università di Pisa, hanno evidenziato come le code lock-free superino significativamente quelle basate su mutua esclusione in termini di scalabilità e throughput. In particolare, le implementazioni basate su operazioni `F&A` evidenziano una distribuzione più equilibrata dell'overhead e una minore contesa, garantendo prestazioni elevate soprattutto in scenari in cui la velocità di elaborazione dei consumatori non è signifcativamente più elevata di quella dei produttori. Tuttavia, il continuo riciclo della memoria richiesto da tali implementazioni non solo riduce l'efficienza complessiva, ma le espone anche a una maggiore inaffidabilità quando i consumatori operano a velocità nettamente superiori rispetto ai produttori.

Abbiamo analizzato l'impatto del pinning dei thread rispetto all'allocazione automatica dei thread da parte del Sistema Operativo, al fine di valutare il fenomeno del cache thrashing. I test evidenziano che le code unbounded sono più vulnerabili a questo problema a causa della continua allocazione di memoria, che ostacola il caching dell'allocatore di memoria. Al contrario, le implementazioni bounded si sono dimostrate più resilienti, mantenendo prestazioni stabili.

Un possibile sviluppo futuro riguarda l'analisi approfondita di nuove tecniche che permetto di evitare l'allocazione dinamica della memoria, ed essere resilienti a fenomeni di *hotspot*, fornendo maggiori garanzie in termini di sicurezza ed

efficienza.

Questa esperienza di ricerca mi ha fornito una visione concreta delle sfide legate alla progettazione di strutture dati concorrenti e mi ha offerto spunti per approfondimenti futuri, con l'obiettivo di migliorare ulteriormente la robustezza e l'efficienza degli algoritmi non bloccanti in scenari reali.

Affrontare queste sfide mi ha permesso di avvicinarmi al mondo della ricerca, comprendendone le dinamiche e le metodologie. Formulare ipotesi, condurre esperimenti e analizzare i risultati mi ha fatto apprezzare il valore del metodo scientifico nel contesto della progettazione di algoritmi efficienti. Non è stata solo una semplice esercitazione teorica, ma un'opportunità concreta per sviluppare un approccio più strutturato e consapevole nell'affrontare le problematiche della programmazione concorrente e dell'ottimizzazione delle prestazioni software.

Bibliografia

[1] Maurice Herlihy, Nir Shavit, Victor Luchangco e Michael Spear. *The Art of Multiprocessor Programming*. 2nd. Morgan Kaufmann, 2020. ISBN: 9780124159501.

[2] Thomas Anderson e Michael Dahlin. *Operating Systems: Principles and Practice*. 2nd. Recursive books, 2014. ISBN: 0985673524.

[3] David Dice, Virendra J Marathe e Nir Shavit. «Lock cohorting: a general technique for designing NUMA locks». In: *ACM SIGPLAN Notices* 47.8 (2012), pp. 247–256.

[4] Brijesh Dongol. «Formalising progress properties of non-blocking programs». In: *International Conference on Formal Engineering Methods*. Springer. 2006, pp. 284–303.

[5] Shahar Timnat e Erez Petrank. «A practical wait-free simulation for lock-free data structures». In: *ACM SIGPLAN Notices* 49.8 (2014), pp. 357–368.

[6] Maurice Herlihy, Victor Luchangco e Mark Moir. «Obstruction-free synchronization: Double-ended queues as an example». In: *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* IEEE. 2003, pp. 522–529.

[7] A Intel. *Intel® Architecture Software Developer's Manual, Volume 1: Basic Architecture, IA-32 Intel Architecture Software Developer's Manuals*. 2004.

[8] Rachid Guerraoui, Alex Kogan, Virendra J Marathe e Igor Zablotchi. «Efficient multi-word compare and swap». In: *arXiv preprint arXiv:2008.02527* (2020).

[9] Raed Romanov e Nikita Koval. «The state-of-the-art LCRQ concurrent queue algorithm does NOT require CAS2». In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 2023, pp. 14–26.

[10] Damian Dechev, Peter Pirkelbauer e Bjarne Stroustrup. «Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs». In: *2010 13th IEEE international symposium on object/component/service-oriented real-time distributed computing*. IEEE. 2010, pp. 185–192.

[11] William K Reinholtz. «Atomic reference counting pointers». In: *CC PLUS PLUS USERS JOURNAL* 22 (2004), pp. 40–45.

[12] Jim Kukunas. *Power and performance: Software analysis and optimization.* Morgan Kaufmann, 2015.

[13] David Mosberger. «Memory consistency models». In: *ACM SIGOPS Operating Systems Review* 27.1 (1993), pp. 18–26.

[14] Vijay Nagarajan, Daniel J Sorin, Mark D Hill e David A Wood. «Total Store Order and the x86 Memory Model». In: *A Primer on Memory Consistency and Cache Coherence*. Springer, 2020, pp. 39–53.

[15] Leslie Lamport. «How to make a correct multiprocess program execute correctly on a multiprocessor». In: *IEEE Transactions on Computers* 46.7 (1997), pp. 779–782.

[16] Hans-J Boehm e Sarita V Adve. «Foundations of the C++ concurrency memory model». In: *ACM SIGPLAN Notices* 43.6 (2008), pp. 68–78.

[17] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar e Peter Sewell. «Clarifying and compiling C/C++ concurrency: from C++ 11 to POWER». In: *ACM SIGPLAN Notices* 47.1 (2012), pp. 509–520.

[18] Richard Jones e Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., 1996.

[19] Andrew W Appel. «Garbage collection». In: *Topics in Advanced Language Implementation* (1991), pp. 89–100.

[20] Maurice P Herlihy e J Eliot B Moss. «Lock-free garbage collection for multiprocessors». In: *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*. 1991, pp. 229–236.

[21] Maged M Michael. «Hazard pointers: Safe memory reclamation for lock-free objects». In: *IEEE Transactions on Parallel and Distributed Systems* 15.6 (2004), pp. 491–504.

[22] Pedro Ramalhete e Andreia Correia. «Brief announcement: Hazard eras-non-blocking memory reclamation». In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 2017, pp. 367–369.

[23] Maurice P. Herlihy e Jeannette M. Wing. «Linearizability: a correctness condition for concurrent objects». In: *ACM Trans. Program. Lang. Syst.* 12.3 (lug. 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972.

[24] Maged M Michael e Michael L Scott. «Simple, fast, and practical non-blocking and blocking concurrent queue algorithms». In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. 1996, pp. 267–275.

[25] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick e Massimo Torquati. «Fastflow: High-Level and Efficient Streaming on Multicore». In: *Programming multi-core and many-core computing systems* (2017), pp. 261–280.

[26] Adam Morrison e Yehuda Afek. «Fast concurrent queues for x86 processors». In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 103–112.

[27] Pedro Ramalhete. *FAAArrayQueue - MPMC lock-free queue*. `http://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html`. Accessed: 2025-03-03. 2016.

[28] Ruslan Nikolaev. «A scalable, portable, and memory-efficient lock-free FIFO queue». In: *arXiv preprint arXiv:1908.04511* (2019).

[29] Steven Feldman e Damian Dechev. «A wait-free multi-producer multi-consumer ring buffer». In: *ACM SIGAPP Applied Computing Review* 15.3 (2015), pp. 59–71.

[30] Leslie Lamport. «Concurrent reading and writing». In: *Communications of the ACM* 20.11 (1977), pp. 806–811.

[31] Lisa Higham e Jalal Kawash. «Critical sections and producer/consumer queues in weak memory systems». In: *Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97)*. IEEE. 1997, pp. 56–63.