ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria - DISI

Corso di Laurea in Informatica

Applicazione di computazione parallela per l'ottimizzazione di un servizio di Data Management

Relatore:
Fabio Vitali

Correlatore:
Francesco Giacomini

Presentata da:
Angelo Ruggieri

Sessione del 17 Dicembre 2025

Anno Accademico 2024/2025

Indice

Capitolo 1

Introduzione

La fisica delle alte energie è una disciplina che, per natura, spinge costantemente al limite le frontiere della tecnologia informatica. Gli esperimenti condotti tramite acceleratori di particelle, come il Large Hadron Collider (LHC) del CERN di Ginevra, producono quantità di dati grezzi senza precedenti, che devono essere archiviati, distribuiti e analizzati da una rete globale di centri di calcolo.

Attualmente, l'infrastruttura di calcolo mondiale (WLCG - Worldwide LHC Computing Grid) si trova di fronte a una transizione importante: l'avvento del **High-Luminosity LHC (HL-LHC)** [1]. Questo aggiornamento degli apparati dell'acceleratore mira ad aumentare la *luminosità* — parametro che indica il numero di collisioni potenziali per unità di tempo e area — di un fattore dieci rispetto ai valori nominali attuali. Un aumento della luminosità si traduce direttamente in un incremento proporzionale della quantità di dati acquisiti dagli esperimenti.

Le stime attuali per l'era HL-LHC prevedono un traffico di rete aggregato globale necessario di circa 9.6 Tbps [17]. In questo scenario, il **CNAF** (Centro Nazionale Tecnologie Informatiche) dell'INFN a Bologna, che opera come uno dei centri di livello *Tier-1* della rete WLCG (oggi situato presso il Tecnopolo), dovrà sostenere circa il 10% di tale carico, gestendo flussi nell'ordine di 1 Tbps. Attualmente, il centro gestisce un archivio di circa 100 PB su disco e ben **200 PB su nastro magnetico**.

È proprio in questo contesto che si inserisce il problema affrontato in questa tesi. Mentre i dati su disco sono immediatamente accessibili, una frazione rilevante e crescente dei dati grezzi risiede su nastro (*tape*), un supporto economico e capiente ma caratterizzato da latenze di accesso elevate. L'operazione di recupero dati da nastro, detta *staging*, è un processo critico che coinvolge interazioni complesse tra file system distribuiti (come GPFS) e database di gestione. Le attuali soluzioni software per la gestione dello staging, basate spesso su architetture *single-threaded*

o con concorrenza limitata, rischiano di diventare un serio collo di bottiglia. Senza un intervento preventivo sull'efficienza del software di gestione, l'infrastruttura di storage potrebbe non riuscire a rendere disponibili i dati ai ricercatori con le tempistiche richieste dai nuovi esperimenti.

Obiettivi e Contributo

Questa tesi, svolta in collaborazione con l'INFN, si pone l'obiettivo di evolvere l'architettura di **StoRM Tape**, il servizio responsabile delle operazioni di staging presso il Tier-1 del CNAF. Lo scopo è superare i limiti delle implementazioni attuali valutando l'adozione di paradigmi di programmazione parallela e concorrente.

La soluzione proposta si basa su due pilastri tecnologici:

- L'impiego degli strumenti di concorrenza moderni offerti dallo standard **C++20**, per trasformare un'applicazione sequenziale in un sistema capace di gestire richieste concorrenti voluminose senza bloccare le risorse di calcolo.
- L'integrazione di strategie di ottimizzazione dell'accesso ai dati persistenti (database SQLite), identificati come punto critico nelle architetture ad alto throughput.

Valutazione Sperimentale

Per validare l'approccio, è stata progettata una suite di *load testing* personalizzata basata sul framework **Locust**. Questo ha permesso di simulare pattern di traffico realistici, stressando il sistema con payload complessi in proporzione simili a quelli previsti durante le "Data Challenge" del WLCG [17].

I risultati sperimentali hanno offerto spunti significativi e, in parte, controintuitivi. Sebbene l'introduzione del parallelismo offra vantaggi teorici evidenti, l'analisi ha dimostrato che in un contesto fortemente *I/O bound* — dove la velocità è dettata dalla risposta del file system e del disco — l'aumento indiscriminato dei thread può essere controproducente. I benchmark evidenziano come l'incremento prestazionale più netto e stabile sia stato ottenuto combinando una concorrenza controllata con l'ottimizzazione mirata del sottosistema di database (specificamente l'adozione della modalità WAL, *Write-Ahead Logging*).

Il risultato di questo lavoro è un sistema più robusto, prevedibile e pronto a scalare per sostenere le sfide imposte dalla prossima generazione di esperimenti di fisica delle alte energie.

Struttura della disserazione di tesi

Il presente elaborato è organizzato come segue:

- Il **Capitolo 2** descrive il contesto scientifico del WLCG, il ruolo del Tier-1 del CNAF e le sfide poste da HL-LHC.
- Il **Capitolo 3** introduce le tecnologie abilitanti utilizzate (C++ Moderno, SQLite, REST API e GPFS).
- Il **Capitolo 4** analizza l'architettura software di StoRM Tape e le modifiche architetturali proposte.
- Il **Capitolo 5** dettaglia l'implementazione, la metodologia di test con Locust e l'analisi critica dei benchmark effettuati.
- Il **Capitolo 6** trae le conclusioni e delinea gli sviluppi futuri per la messa in produzione del servizio.

Capitolo 2

Panoramica sul servizio di data management StoRM Tape

La gestione dei dati su scala Petabyte rappresenta una delle sfide più ardue per l'infrastruttura di calcolo scientifico contemporanea. Questo capitolo analizza lo stato dell'arte dei sistemi di storage nel contesto della fisica delle alte energie, evidenziando i limiti implementativi delle soluzioni precedenti di fronte ai requisiti del prossimo High-Luminosity LHC [1]. Successivamente, vengono introdotti i paradigmi di programmazione concorrente e le tecnologie di persistenza (SQLite WAL [14]) e testing (Locust) che costituiscono i blocchi fondanti per il superamento di tali limitazioni.

2.1 Lo Storage Tiering nel WLCG

Il modello di calcolo del CERN, organizzato nella Worldwide LHC Computing Grid (WLCG)[1], si basa su una gerarchia di centri di calcolo. I centri Tier-1[2], come il CNAF dell'INFN, hanno la responsabilità primaria della custodia a lungo termine dei dati grezzi (RAW data[3]).

Data la mole di informazioni, è economicamente e tecnologicamente insostenibile mantenere tutti i dati su dischi ad accesso rapido (HDD/SSD). Si adotta

[1] Il progetto Worldwide LHC Computing Grid (WLCG) è una collaborazione globale di circa 160 centri di calcolo in più di 40 paesi, che collega infrastrutture di rete nazionali e internazionali.

[2] Nell'architettura WLCG, il Tier-0 (CERN) acquisisce i dati dai rivelatori. I Tier-1 (centri nazionali come il CNAF) custodiscono una copia dei dati grezzi e forniscono capacità di riprocessamento. I Tier-2 (università) sono dedicati all'analisi e alla simulazione.

[3] I RAW data sono le informazioni digitali prodotte direttamente dall'elettronica dei rivelatori, non ancora ricostruite in oggetti fisici (elettroni, muoni, ecc.).

quindi un modello HSM (*Hierarchical Storage Management*)[4] che prevede due livelli di qualità del servizio:

- **Disk Pool (Online):** Cache ad alte prestazioni per i dati in uso corrente.
- **Tape Library (Custodial):** Librerie di nastri magnetici per l'archiviazione a lungo termine, caratterizzate da costi ridotti ma latenze di accesso elevate (minuti o ore).

2.1.1 La transizione a HTTP/REST

Storicamente, l'interazione con questi sistemi era mediata dal protocollo SRM (*Storage Resource Manager*)[5], che si è però rivelato pesante, complesso da mantenere e non allineato con gli standard web moderni. La comunità WLCG ha quindi avviato una transizione verso interfacce standard HTTP/WebDAV [18]. Questo cambio di paradigma richiede che i servizi di storage espongano API RESTful[6] capaci di gestire operazioni complesse (come il *Recall*[7] da nastro) attraverso verbi HTTP standard (POST, GET, DELETE).

2.2 Limiti degli approcci tradizionali

In un'architettura server classica (ad esempio, un server web che gestisce una richiesta per thread), quando arriva una richiesta che necessita di interagire con il filesystem o con il sistema a nastro, il thread si blocca in attesa della risposta (stato *Wait*). Questo fenomeno è noto come Blocking I/O[8]. Nei sistemi HSM, le operazioni di verifica di esistenza file o di interrogazione dello stato del nastro possono richiedere tempi significativi, specialmente se il filesystem sottostante è sotto carico.

[4]HSM è una tecnica di virtualizzazione dello storage che sposta automaticamente i dati tra supporti ad alto costo/alta velocità (dischi) e supporti a basso costo/bassa velocità (nastri) in base a policy di accesso e frequenza di utilizzo.

[5]Protocollo Grid standardizzato per gestire lo spazio di storage distribuito e negoziare trasferimenti di file, spesso implementato tramite SOAP/XML.

[6]REST (Representational State Transfer) è uno stile architetturale per sistemi distribuiti che utilizza i verbi HTTP (GET, POST, PUT, DELETE) per manipolare risorse identificate da URI.

[7]Il Recall dal nastro è processo di recupero o ripristino di dati archiviati su supporti a nastro magnetico.

[8]Nel Blocking I/O, il thread chiamante viene sospeso dal sistema operativo finché l'operazione di input/output non è completata. Durante questo tempo, il thread non può eseguire altre istruzioni, sprecando cicli CPU potenziali o occupando risorse di memoria.

Un approccio sequenziale o scarsamente parallelizzato porta rapidamente a due fenomeni degradanti nel caso il collo di bottiglia sia l'implementazione del software e non il file system a cui si affida:

1. **Resource Starvation:** Tutti i thread del server sono bloccati in attesa di I/O, rendendo il servizio irraggiungibile anche per richieste semplici.

2. **Increased Tail Latency:** La latenza percepita dagli utenti non segue una distribuzione normale, ma presenta una "coda lunga" (alti percentili P95[9] o P99).

2.3 Paradigmi e Tecnologie Abilitanti

Per superare i limiti di scalabilità sopra descritti, è necessario adottare tecnologie che permettano di disaccoppiare la logica di controllo (ricezione richieste) dalla logica di I/O (esecuzione su disco/DB).

2.3.1 Modern C++ e Parallel Algorithms

Il linguaggio C++ ha subito una profonda evoluzione con l'introduzione dello standard C++17 [6], che ha rivoluzionato il modo di scrivere codice concorrente integrando la parallelizzazione direttamente nella libreria standard (STL).

La novità più significativa è l'introduzione delle **Execution Policies** nel header `<execution>`. Questo permette di trasformare algoritmi classici come `std::for_each`, `std::transform` o `std::reduce` in operazioni parallele semplicemente aggiungendo un parametro. L'utilizzo della policy `std::execution::par` [2] istruisce il compilatore a distribuire il carico di lavoro su più thread disponibili, sfruttando le capacità multi-core delle moderne CPU senza che lo sviluppatore debba gestire manualmente la creazione e il corretto assegnamento dei processi. Questo approccio dichiarativo ("cosa fare" invece di "come farlo") riduce la complessità del codice e minimizza gli errori comuni della programmazione concorrente, come le *race condition*[10], delegando l'ottimizzazione del thread-pool alla libreria sottostante.

[9] Il percentile P95 indica il valore sotto il quale ricade il 95% delle osservazioni. Nel networking, è una metrica critica per valutare la stabilità del servizio escludendo gli outlier più estremi.

[10] Race Condition: anomalia in cui il risultato di un programma dipende dalla sequenza temporale o dall'ordine di esecuzione di thread o processi non controllabili, portando spesso a bug difficili da riprodurre.

2.3.2 Crow: Microframework C++ per il Web

Per l'esposizione delle API REST, la scelta tecnologica ricade su **Crow** [3]. A differenza di framework storici più pesanti, Crow è un microframework *header-only*[11] ispirato a Flask (Python) ma scritto in C++ moderno. La sua caratteristica fondamentale è l'architettura asincrona basata su **Boost.Asio** [9]. Questo permette di gestire migliaia di connessioni concorrenti utilizzando un numero limitato di thread, delegando le operazioni di rete al sistema operativo (tramite `epoll`[12] su Linux). Tuttavia, Crow da solo non risolve il problema del blocking I/O sul filesystem; è necessario integrarlo con una logica applicativa capace di fare offloading dei compiti pesanti.

2.3.3 SQLite e il Write-Ahead Logging (WAL)

La persistenza dello stato delle richieste (quali file sono stati richiesti, il loro stato di avanzamento) è affidata a **SQLite** [13]. Tradizionalmente, SQLite è considerato un database inadatto ad alta concorrenza a causa del suo meccanismo di locking a livello di file, che inibisce le letture durante una scrittura.

Tuttavia, l'introduzione della modalità **WAL (Write-Ahead Logging)** ha cambiato radicalmente questo scenario.

In modalità WAL:

- Le modifiche non vengono scritte immediatamente nel file database principale, ma in un file di log separato (WAL file).
- Questo permette ai lettori di accedere al database principale mentre uno scrittore sta aggiungendo dati al WAL.
- La concorrenza aumenta drasticamente (*Readers do not block Writers, Writers do not block Readers*).

Per sfruttare questa caratteristica in un'applicazione multithread C++, è necessario un design attento della gestione delle connessioni, evitando la condivisione di handle non thread-safe.

[11] Libreria composta esclusivamente da file header (.h o .hpp), che non richiede una compilazione separata in file oggetto (.o o .lib). Il codice viene compilato direttamente nell'unità di traduzione che lo include.

[12] `epoll` è una syscall di Linux per il monitoraggio scalabile di molteplici descrittori di file. È molto più efficiente di `select` o `s` quando si gestisce un alto numero di connessioni simultanee.

2.4 Metodologia di Validazione: Il Load Testing

La validazione di sistemi di data management complessi come StoRM Tape richiede un approccio metodologico che vada oltre il semplice test di raggiungibilità delle API. In particolare, quando l'obiettivo è misurare l'efficacia di algoritmi paralleli interni, il test deve essere in grado di generare carichi di lavoro che stressino specificamente le componenti reingegnerizzate.

2.4.1 Il limite dei Benchmark "Stateless"

Gli strumenti di benchmarking tradizionali, come *Apache Benchmark* (ab) o *wrk*, sono progettati per misurare le prestazioni pure del server web (RPS - Requests Per Second) inviando richieste identiche e indipendenti tra loro ("Stateless").

Tuttavia, questo approccio è inadeguato per protocolli asincroni come quello di gestione del nastro. Il protocollo Tape REST prevede un flusso logico sequenziale e "Stateful":

1. Il client invia una richiesta di lavoro (*Stage*).
2. Il server risponde con un identificativo univoco (*Request ID*).
3. Il client deve usare quell'ID per interrogare periodicamente lo stato (*Status*).

Uno strumento che si limita a "bombardare" l'endpoint di sottomissioni (*Stage requests*) senza seguire il flusso logico creerebbe solo migliaia di job orfani nel database, senza mai testare la capacità del sistema di portarli a termine, falsando completamente la misurazione delle prestazioni reali.

2.4.2 Testing Comportamentale con Locust

Per superare questi limiti, in questo lavoro è stato selezionato il framework **Locust** [10]. Sebbene Locust sia famoso per la sua capacità di simulare migliaia di utenti, la caratteristica determinante per questo progetto è la sua natura **Code-Driven**.

A differenza di altri tool basati su configurazioni XML o interfacce grafiche, in Locust il comportamento dell'utente virtuale è definito interamente tramite codice Python. Ciò permette di modellare scenari complessi che richiedono logica condizionale e gestione della memoria a breve termine.

Nel contesto specifico della validazione del parallelismo su file system, Locust permette di:

- **Gestire lo Stato:** Memorizzare l'ID ricevuto nella prima risposta e utilizzarlo nelle richieste successive.
- **Simulare Batch Variabili:** Invece di aumentare il numero di utenti, è possibile programmare un singolo utente che invia richieste con payload di dimensioni crescenti (da decine a migliaia di file), isolando così il tempo di elaborazione interno del server dalle latenze di gestione delle connessioni di rete.
- **Validazione Semantica:** Verificare non solo che il server risponda con codice HTTP 200, ma che il contenuto JSON della risposta sia coerente con le aspettative (es. che tutti i file richiesti siano effettivamente passati allo stato "DONE").

L'adozione di questo strumento sposta il focus dal semplice "stress test" di rete alla **validazione prestazionale della logica applicativa**, permettendo di quantificare con precisione il guadagno ottenuto grazie alla parallelizzazione degli algoritmi di backend.

Capitolo 3

Parallelizzazione di StoRM Tape e configurazione dei test di carico

L'obiettivo di questo lavoro di tesi è stato superare i limiti strutturali del servizio **StoRM Tape**, un componente critico nell'infrastruttura di storage del Tier-1 del CNAF. Per rispondere ai requisiti di scalabilità imposti dallo scenario High-Luminosity LHC, non è stato sufficiente ottimizzare il codice esistente, ma è stato necessario riprogettare il modello di concorrenza del servizio.

L'intervento si è concentrato sulla rimozione di un "doppio collo di bottiglia": quello computazionale, legato all'elaborazione sequenziale dei file, e quello legato alla persistenza, dovuto all'accesso serializzato al database. Parallelamente, è stato ideato un framework di validazione scientifica basato su *Locust*, essenziale per misurare l'efficacia delle modifiche in scenari realistici.

In questo capitolo viene descritta l'architettura logica della soluzione **StoRM Tape Parallelo**, analizzando come il cambio di paradigma verso il parallelismo e la gestione efficiente delle connessioni possa potenzialmente migliorare le capacità del sistema.

3.1 Evoluzione del Core Engine di StoRM Tape

StoRM Tape agisce come intermediario tra le richieste HTTP (REST) degli utenti e le risorse di storage sottostanti (GPFS [5] e Tape Library).

Nella versione *Legacy*, il servizio gestiva le richieste seguendo un modello strettamente seriale. Sebbene il server web potesse accettare connessioni multiple, la logica interna processava le liste di file ("bulk") iterando su un elemento alla volta. Questo approccio rendeva il tempo di risposta dipendente linearmente dalla dimensione della richiesta e dalla latenza istantanea del sottosistema di storage.

Il contributo di questa tesi introduce un'architettura che interviene su due livelli logici distinti ma interdipendenti: il piano di esecuzione e il piano dei dati.

3.1.1 Da Modello Sequenziale a Parallelo

La prima area di intervento riguarda la strategia con cui vengono processate le richieste massive. Le operazioni di *Data Management* su nastro coinvolgono spesso migliaia di file per singola transazione.

Nel vecchio modello architetturale, il sistema operava come un esecutore singolo: ogni operazione di I/O (lettura metadati, verifica esistenza) bloccava il flusso principale fino al suo completamento. In un contesto di filesystem distribuito, dove la latenza di rete è intrinseca, questo comportava lunghi periodi di inattività della CPU ("Idle Time") in attesa delle risposte dello storage.

La nuova architettura inverte questo paradigma adottando un modello **Scatter-Gather**:

1. **Decomposizione (Scatter):** La richiesta massiva in ingresso viene immediatamente scomposta in unità di lavoro atomiche (task), ciascuna relativa a un singolo file.

2. **Esecuzione Concorrente:** Queste unità vengono affidate a un pool di risorse di calcolo che le esegue simultaneamente, saturando la capacità di I/O del sistema sottostante.

3. **Aggregazione (Gather):** I risultati parziali vengono raccolti e ricomposti in un'unica risposta per l'utente.

Questo approccio disaccoppia il tempo di elaborazione dal numero di file, rendendo il sistema resiliente ai rallentamenti dei singoli componenti hardware.

3.1.2 Riprogettazione dell'Accesso al Database

L'introduzione del parallelismo computazionale ha reso evidente un secondo limite strutturale: la gestione della persistenza. Avere molteplici processi pronti a lavorare è inutile se tutti devono passare per un unico punto di accesso ai dati.

Per supportare la nuova architettura parallela, il layer di persistenza è stato ridisegnato abbandonando il modello a "connessione singola" in favore del pattern architetturale del **Connection Pooling**.

Concettualmente, il sistema non vede più il database come una risorsa esclusiva da bloccare per ogni operazione, ma come un servizio condiviso accessibile attraverso canali multipli. Il *Pool* agisce come un gestore di risorse che:

- Mantiene un insieme di connessioni al database sempre aperte e pronte all'uso.
- Assegna temporaneamente una connessione esclusiva a un thread richiedente.
- Gestisce il ciclo di vita delle transazioni in modo isolato, evitando che un'operazione di scrittura lunga blocchi le operazioni di lettura degli altri task.

Questa evoluzione permette al throughput del database di scalare orizzontalmente in funzione delle risorse di calcolo disponibili, rimuovendo il collo di bottiglia che storicamente limitava le prestazioni dei sistemi basati su SQLite in scenari multi-thread.

3.2 Analisi delle Funzionalità Ottimizzate

Le modifiche al motore impattano direttamente l'efficienza dei tre principali *use case* del protocollo Tape REST [18].

3.2.1 UC1: Stage (Recall Massivo)

L'operazione di *Stage* rappresenta l'inizio del flusso di recupero dati. Richiede la validazione dei percorsi logici e la creazione dello stato iniziale nel sistema.

Evoluzione del flusso: Invece di processare la richiesta come una lunga transazione sequenziale, il sistema verifica la validità dei percorsi in parallelo in base alle decisioni dell'algoritmo di pooling. Questo approccio fa sperare che il sistema restituisca l'identificativo della richiesta (*Request ID*) più velocemente.

3.2.2 UC2: Monitoraggio (Status Polling)

Dopo la sottomissione, i client interrogano ripetutamente il server (`GET`) per sapere se i file sono pronti.

Evoluzione del flusso: Lo status è un'operazione frequente in cui il client richiede lo stato corrente di un insieme di file corrispondente ad un *Request ID*. Nel design precedente, il sistema verificava lo stato di ogni file sequenzialmente,

interrogando il filesystem per ciascuno di essi per confermarne la presenza o l'avvenuto richiamo. Adesso il sistema lancia le verifiche sul filesystem per tutti i file della richiesta in modo concorrente abbattendo il tempo di latenza necessario per costruire la risposta.

3.3 Il Framework di Validazione e Benchmark

Per validare scientificamente l'efficacia dell'architettura proposta, è stato necessario sviluppare un ecosistema di test *ad hoc*. Gli strumenti generici di benchmark HTTP non erano in grado di replicare la complessità "a stati" del protocollo Tape REST.

È stato quindi realizzato un framework basato su **Locust**, composto da due elementi software originali sviluppati per questa tesi.

3.3.1 Il Client Simulato (Locustfile)

Il file `locustfile.py` definisce il comportamento di un utente virtuale che non si limita a "bombardare" il server, ma simula il ciclo di vita reale di un trasferimento dati:

- **Autenticazione Reale:** Gestisce token JWT (*JSON Web Token*) [8] validi per operare in ambienti autenticati.
- **Macchina a Stati:** Implementa la logica *Submit* $\rightarrow$ *Wait* $\rightarrow$ *Status*. L'utente invia una richiesta, attende il *Request ID* e poi entra in un ciclo di polling intelligente, interrogando il server finché il lavoro non è concluso.
- **Carico Misto:** Alterna operazioni di scrittura (Stage) a operazioni di lettura (ArchiveInfo) per stressare contemporaneamente tutti i sottosistemi (DB in scrittura e Filesystem in lettura).

3.3.2 L'Orchestratore di Benchmark (Benchmark Suite)

Per garantire la riproducibilità dei risultati, è stato sviluppato lo script di orchestrazione `benchmark.py`. Questo strumento automatizza l'esecuzione di intere campagne di test parametriche, variando in modo controllato il numero di file per richiesta (10, 50, 100, 200) e isolando le variabili.

Al termine dell'esecuzione, l'orchestratore acquisisce i dati grezzi prodotti da Locust ed elabora le statistiche utilizzando *pandas* [16], separando le metriche per

tipologia di endpoint (distinguendo la latenza dello staging da quella dello status). In particolare inserisce in un file di riassunto le seguenti statistiche:

- Richieste al secondo (*throughtput*) e quantità di fallimenti (numero di risposte con codice maggiore di 400)
- Latenza media, tempo minimo e il 95 percentile per le richieste di stage, per quelle del primo giro di status e per quelle del secondo.

Vengono infine prodotti automaticamente dei grafici di tendenza tramite *matplotlib* [4], fornendo l'evidenza empirica necessaria per l'analisi prestazionale discussa nel Capitolo 5.

Capitolo 4

Dettagli implementativi

In questo capitolo vengono approfonditi gli aspetti tecnici del contributo, scendendo al livello del codice sorgente. L'obiettivo è fornire una documentazione tecnica delle modifiche apportate al core C++ di **StoRM Tape** e dell'implementazione della suite di test in Python.

Questa sezione è pensata per fornire le informazioni necessarie a uno sviluppatore che intenda mantenere, estendere o replicare le funzionalità introdotte [12].

4.1 Reingegnerizzazione di StoRM Tape (C++)

Il servizio StoRM Tape è scritto in C++ moderno (standard C++20 [7]). Le modifiche principali hanno riguardato due aree critiche: la gestione della concorrenza nell'accesso al database e la parallelizzazione degli algoritmi di business logic.

4.1.1 Database Connection Pooling con SOCI

Come discusso nel capitolo precedente, l'uso di una singola sessione SQLite rappresentava un punto di serializzazione. Per risolvere il problema, è stata introdotta la classe `soci::connection_pool`.

Modifiche alla classe SociDatabase

Il costruttore della classe `SociDatabase` è stato modificato per accettare un riferimento a un pool di connessioni invece che a una singola sessione.

```
1 class SociDatabase : public Database {
2   soci::connection_pool& m_pool; // Nuova implementazione
3   // ...
```

```
}
```

Gestione del Lease delle Connessioni

Per ogni operazione che richiede accesso al DB (sia in lettura che in scrittura), è stato introdotto un helper statico `get_session` che gestisce il ciclo di vita della connessione per il thread corrente.

```
static soci::session& get_session(soci::connection_pool& pool) {
  // pool.lease() blocca il thread finche' non c'e' una connessione
    libera
  // Restituisce l'indice della connessione nel pool
  size_t index = pool.lease();

  // Ottiene la sessione associata all'indice
  static thread_local soci::session& session{pool.at(index)};
  return session;
}
```

Listing 4.1: Helper per l'acquisizione della sessione dal pool

Ogni metodo della classe (es. `insert`, `update`, `find`) è stato riscritto per utilizzare questo pattern. Invece di usare `m_sql` direttamente (nella versione precedente veniva aperta una sola connessione che veniva mantenuta in quella variabile, ora sostituita da un riferimento all'intero connection_pool), il codice in questa versione deve invocare `get_session` per acquisire una sessione locale:

```
bool SociDatabase::insert(StageId const& id, StageRequest const& stage)
    {
    // Acquisizione di una connessione dedicata al thread
    auto& sql = get_session(m_pool);
    soci::transaction tr{sql};

    // Esecuzione della query sulla connessione isolata
    sql << "INSERT INTO Stage VALUES ...", soci::use(s_entity);

    // La transazione viene committata automaticamente alla distruzione
    di 'tr'
    // La connessione viene rilasciata automaticamente al pool
}
```

Listing 4.2: Funzione per l'inserimento di una stage request nel database, utilizzando la nuova connection pool

Questa modifica permette a N thread di eseguire operazioni sul DB contemporaneamente, dove N è la dimensione del pool configurata all'avvio in `main.cpp`.

4.1.2 Algoritmi Paralleli in Tape Service

Il cuore della parallelizzazione logica risiede nel file header `tape_service_utils.hpp`. Qui, oltre ad esservi degli estratti di alcune funzionalità presenti altrove, ma necessarie per il corretto funzionamento del programma, sono state riscritte le funzioni che interagiscono col file system. A queste funzioni sono stati sostituiti i cicli sequenziali con, in base all'evenienza, gli algoritmi `std::transform` o `std::for_each`, configurati con la policy di esecuzione parallela (`std::execution::par`).

Di seguito analizziamo i dettagli delle tre implementazioni principali aggiornate.

1. Extend Paths with Localities

Questa funzione associa ad ogni percorso fisico le informazioni sulla sua località (es. *tape*, *disk*, *lost*).

Nella modalità parallela, il primo passo fondamentale è la pre-allocazione del vettore di destinazione. Poiché l'inserimento dinamico in un `std::vector` (tramite `push_back`) non è *thread-safe*, è necessario dimensionare il vettore a priori.

```
1 inline auto extend_paths_with_localities(PhysicalPaths&& paths, Storage
     & storage) {
2   // Resize preventivo: permette ai thread di scrivere
3   // in posizioni di memoria distinte senza data race.
4   path_localities.resize(paths.size());
```

Listing 4.3: Pre-allocazione per accesso parallelo

Successivamente, viene utilizzato l'algoritmo `std::transform` con policy parallela. Ogni thread lavora su un indice specifico, recuperando lo stato del file dallo storage e costruendo la coppia `PathLocality`.

```
1   std::transform(std::execution::par, paths.begin(), paths.end(),
2       path_localities.begin(), [&](PhysicalPath& path) {
3
4         // Lettura concorrente (thread-safe) dello stato
5         auto const locality =
6             ExtendedFileStatus{storage, path}.locality();
```

```
7
8         return PathLocality{std::move(path), locality};
9       });
10 }
```

Listing 4.4: Trasformazione parallela degli attributi

2. Resolve Paths

La funzione `resolve_paths` ha il compito di risolvere i percorsi logici in fisici e validare l'esistenza dei file.

L'operazione avviene *in-place* sugli oggetti `File`, modificando direttamente i loro campi interni. Per questo motivo viene utilizzato `std::for_each`.

```
1     std::for_each(std::execution::par, files.begin(), files.end(),
2         [&](auto& file) {
3       // Risoluzione del percorso (operazione CPU bound)
4       file.physical_path = resolve(file.logical_path);
5
6       std::error_code ec;
7       // System call fs::status (operazione I/O bound)
8       auto status = fs::status(file.physical_path, ec);
9
10      // Aggiornamento atomico dello stato del singolo file
11      if (ec || !fs::is_regular_file(status)) {
12        file.state       = File::State::failed;
13        file.started_at  = now;
14        file.finished_at = now;
15      }
16    });
17 }
```

Listing 4.5: Risoluzione e validazione parallela

L'utilizzo del parallelismo è qui particolarmente vantaggioso poiché la chiamata a `fs::status` coinvolge operazioni di I/O sul filesystem, che possono essere eseguite efficacemente in concorrenza.

3. Check File Status

La funzione `check_file_status` verifica se lo stato dei file sottomessi è cambiato (es. da `submitted` a `started`, o da `started` a `completed`).

A differenza delle funzioni precedenti, qui è necessario popolare un vettore di risultati (`files_to_update`) la cui dimensione non è nota a priori. Poiché

`emplace_back` non è sicuro se chiamato da più thread contemporaneamente, è stato introdotto un meccanismo di sincronizzazione.

Viene dichiarato un `std::mutex` locale prima del ciclo parallelo:

```
inline auto check_file_status(Files& files, Storage& storage, std::
    time_t now) {
     std::vector<std::pair<PhysicalPath, File::State>> files_to_update;
     std::mutex mx; // Mutex per proteggere files_to_update
```

Listing 4.6: Definizione del Mutex per la sincronizzazione

All'interno della lambda eseguita in parallelo, quando un thread rileva un cambio di stato e deve scrivere nel vettore condiviso, acquisisce il lock. L'uso di `std::scoped_lock` garantisce che il lock venga rilasciato automaticamente all'uscita dallo scope (RAII)[1], anche in caso di eccezioni.

```
  // ... logica di controllo stato ...

  // Se lo stato e' cambiato, acquisiamo il lock
  std::scoped_lock l{mx};

  // Sezione critica: scrittura sicura nel vettore
  files_to_update.emplace_back(file.physical_path, file.state);

  } // Il lock viene rilasciato automaticamente qui
```

Listing 4.7: Sezione critica protetta da Scoped Lock

Questo approccio ibrido permette di eseguire i controlli costosi (lettura xattr[2] e logica di business) in parallelo, serializzando solo la brevissima operazione di scrittura dei risultati.

4.2 La Suite di Load Testing (Python)

Per misurare efficacemente i benefici di queste modifiche, è stato necessario sviluppare un sistema di test capace di generare carichi di lavoro specifici (richieste con molti file) e di seguire il protocollo stateful di StoRM Tape.

[1] RAII (Resource Acquisition Is Initialization): è un approccio che lega la gestione delle risorse alla durata di vita di un oggetto in stack. La risorsa viene acquisita nel costruttore dell'oggetto e viene rilasciata automaticamente quando questo esce dallo scope.

[2] Extended Attributes (xattr): metadati arbitrari associati a un file, non interpretati dal sistema operativo ma usati qui per comunicare lo stato del nastro.

4.2.1 Il Client Simulato: locustfile.py

Il file `locustfile.py` definisce il comportamento dell'utente virtuale `StormTapeUser`.

Configurazione Dinamica

L'aspetto chiave di questo script è la sua configurabilità tramite variabili d'ambiente. Questo permette all'orchestratore esterno di modificare il numero di file per richiesta (`STORM_FILES_PER_REQ`) senza dover modificare il codice sorgente.

```
# Lettura parametri da variabili d'ambiente con valori di default
create_amount = int(os.getenv("STORM_FILES_PER_REQ", "10"))

token = os.getenv("AT")
# ... Errore nel caso non venga trovato
auth = {"Authorization": f"Bearer {token}"} if token != "" else {}
```

Listing 4.8: Lettura della configurazione da ambiente

Tra le variabili d'ambiente rilevate è presente anche l'*access token*, spesso necessario per autenticarsi ed interagire via API con servizi di data management come StoRM Tape.

Definizione dell'Utente Virtuale

La classe `StormTapeUser` modella l'agente di test. Una scelta progettuale importante è l'impostazione di `wait_time`. A differenza dei test utente classici che includono pause per simulare il tempo di lettura, qui vogliamo testare il throughput massimo del server, quindi il tempo di attesa è azzerato.

```
class StormTapeUser(HttpUser):
    # Tempo di attesa nullo per massimizzare il throughput
    wait_time = constant(0)
```

Listing 4.9: Classe User con pacing azzerato

Task Principale: Generazione Payload

Il metodo `do_stage_and_status` inizia costruendo dinamicamente il payload JSON. I percorsi vengono generati in modo pseudo-casuale per limitare il rischio che avvenga caching dei risultati ed hanno questo particolare formato perchè precedentemente sono stati creati 10000 files sul server che si occupa dell'hosting di StoRM Tape. I files sono stati suddivisi in 100 directories da 100 ognuna e configurati in modo tale che apparissero come salvati anche su nastro.

```
    @task(3)
    def do_stage_and_status(self):
        new_files = []
        for i in range(create_amount):
            # Genera percorsi del tipo /tape/dir055/file012
            dirtext = f"{random.randrange(1, 101):03d}"
            filetext = f"{random.randrange(1, 101):03d}"
            new_files.append({"path": f"/tape/dir{dirtext}/file{
    filetext}"})

        payload = {"files": new_files}
```

Listing 4.10: Generazione dinamica dei percorsi

Task Principale: Sottomissione (POST stage)

Il client invia la richiesta di stage. Se l'operazione ha successo (codice 200 o 201), è fondamentale estrarre il `requestId` dalla risposta JSON, poiché servirà per la fase successiva.

```
        with self.client.post("/api/v1/stage",
                headers=auth,
                json=payload,
                catch_response=True,
                name="stage") as resp:

            if resp.status_code in (200, 201):
                try:
                    j = resp.json()
                    # Salviamo l'ID per lo status successivo
                    self.request_id = j.get("requestId")
                except Exception as e:
                    resp.failure(f"JSON parse error: {e}")
            else:
                resp.failure(f"Stage failed: {resp.status_code}")
```

Listing 4.11: Invio POST e recupero Request ID

Task Principale: Status (GET stage)

Una volta ottenuto l'ID, l'utente consegue due volte uno status della richiesta. Queste due operazioni sono distinte e vengono misurate separatamente perchè StoRM Tape internamente segue un processo diverso dalla seconda status in poi (sulla stessa richiesta). La prima status in assoluto fa sì che avvenga:

1. Un primo accesso di interrogazione al database, che non saprà ancora indicare a che punto è il processo di recall dei files.
2. Un effettivo controllo della presenza di questi files nel sistema.
3. Un aggiornamento finale nel database con la loro posizione.

Dalla seconda status in poi StoRM Tape si ferma alla prima interrogazione del database, che questa volta avrà già pronta la risposta.

```
        if self.request_id:
            for i in range(2):
                r = self.client.get(f"/api/v1/stage/{self.request_id}",
                    headers=auth, verify=False,
                    name=f"get_stage{i+1}")
```

Listing 4.12: Ciclo di Status

4.2.2 L'Orchestratore: benchmark.py

Per automatizzare l'esecuzione di test con carichi crescenti, è stato creato uno script wrapper in Python.

Loop di Esecuzione

Lo script itera su una lista predefinita di carichi (`FILES_PER_REQUEST_LIST`). Per ogni iterazione, lancia un nuovo processo Locust impostando la variabile d'ambiente appropriata.

```
FILES_PER_REQUEST_LIST = [10, 50, 100, 200]

def run_benchmark_suite():
    for n_files in FILES_PER_REQUEST_LIST:
        # Passaggio del parametro di carico tramite ENV
        env = os.environ.copy()
        env["STORM_FILES_PER_REQ"] = str(n_files)

        cmd = [
            "locust", "-f", LOCUSTFILE,
            "--users", str(USERS),
            "--run-time", DURATION,
            "--headless",       # Esecuzione senza UI
            "--csv", prefix,    # Export dei dati grezzi
            "--only-summary"
        ]
```

```
17
18          subprocess.run(cmd, check=True, env=env)
```

Listing 4.13: Loop principale dell'orchestratore

Analisi Differenziata (Pandas)

Dopo ogni esecuzione, lo script legge il CSV prodotto da Locust. Un passaggio critico è la separazione delle metriche. Locust aggrega i tempi di risposta, ma per noi è vitale distinguere la latenza della `POST` (che scrive nel DB) da quella della `GET` (che legge).

```
1           df = pd.read_csv(stats_file)
2
3           # Filtriamo per nome della richiesta (tag 'name' in locustfile)
4           stage_row = df[df["Name"] == "stage"]       # POST
5           poll_row = df[df["Name"] == "get_stage1"]    # GET
6           poll2_row = df[df["Name"] == "get_stage2"]  # 2nd GET
7
8           # ... Raccolta precisa delle metriche in una struttura apposita
```

Listing 4.14: Filtraggio metriche con Pandas

Questi dati puntuali vengono poi aggregati in una struttura dati finale utilizzata per generare i grafici di confronto presentati nel capitolo dei Risultati.

Capitolo 5

Valutazione tramite uno strumento creato ad-hoc

In questo capitolo vengono presentati i risultati ottenuti dalla campagna di test effettuata sul sistema **StoRM Tape** parallelizzato. L'analisi è suddivisa in due macro-aree: una valutazione quantitativa dell'*efficienza*, basata sulle metriche di performance raccolte, e una valutazione qualitativa dell'*efficacia*, focalizzata sui miglioramenti funzionali e manutentivi del software.

5.1 Efficienza: Valutazione Quantitativa

L'obiettivo di questa sezione è misurare l'impatto che ha avuto l'attivazione della modalità WAL di SQLite, che ha permesso l'implementazione di una connection pool per gli accessi al suo database, sulle prestazioni del sistema, confrontando la versione originale (sequenziale) con la nuova (parallela).

5.1.1 Ambiente e Metodologia di Test

I test sono stati eseguiti su una macchina di produzione ospitata sui server dell'INFN, equipaggiata con il filesystem distribuito enterprise **GPFS** (IBM Spectrum Scale).

È importante notare che, in una fase preliminare, si è tentato di eseguire dei micro-benchmark interni per misurare puntualmente la durata delle singole funzioni parallelizzate tramite `std::execution::par`. Tuttavia, l'efficace meccanismo di caching implementato da GPFS ha reso i tempi di esecuzione delle system call estremamente volatili e difficili da isolare, invalidando di fatto misurazioni su scala microscopica.

Per tale motivo sono state tenute disabilitate le modifiche fatte alle funzioni sopracitate e la validazione è stata condotta esclusivamente tramite lo strumento di benchmarking esterno basato su *Locust* (descritto nei capitoli precedenti), che misura le prestazioni *end-to-end* dal punto di vista del client.

Protocollo di Test

Ogni sessione di test segue un protocollo incrementale standardizzato, diviso in 4 fasi da 10 secondi ciascuna, in cui varia il carico di lavoro per richiesta:

- **4 Fasi da 10s:** Il numero di files per richiesta varia da 10, 50, 100 e 200 file a seconda della fase.
- **Utenti Concorrenti:** Il numero di utenti virtuali concorrenti varia tra 1, 2, 4, 8 e 16 a seconda dello scenario.

Sono state confrontate 4 configurazioni principali del servizio:

1. **Sequenziale (Stato dell'arte):** La versione originale di StoRM Tape con modalità WAL disattivata, operante in modalità sequenziale su 1 thread dedicato.
2. **Sequenziale (Originale modificata):** La versione originale di StoRM Tape, ma con modalità WAL attivata, sempre operante in modalità sequenziale su 1 thread dedicato.
3. **Parallelo (Single-threaded):** La nuova versione con modalità WAL attiva e connection pool di accessi al database SQLite, limitata a 1 thread dedicato.
4. **Parallelo (Multi-threaded):** La nuova versione configurata per utilizzare 3 thread dedicati (sui 4 disponibili, riservandone uno al framework web Crow).

5.1.2 Analisi del Caso Base: StoRM Tape Sequenziale

Per stabilire una baseline, analizziamo il comportamento della versione originale con un singolo utente concorrente. Il resoconto seguente rappresenta l'output generato alla fine di ogni sessione di test.

Come evidenziato in Tabella 5.1 e nelle Figure 5.1 e 5.3, all'aumentare del numero di file per richiesta, il throughput (RPS) diminuisce drasticamente, passando da 12.36 a 5.81 req/s. Parallelamente, la latenza media dell'operazione di `Stage` (scrittura su DB e fs) cresce in modo lineare, indicando che il tempo di elaborazione

Tabella 5.1: Metriche prestazionali: StoRM Tape Sequenziale, 1 Utente Concorrente

Files/ Req	RPS	Fail	Stage Avg (ms)	Stage P95	Stage Min	Status 1 Avg (ms)	Status 1 P95	Status 1 Min	Status 2 Avg (ms)	Status 2 P95	Status 2 Min
10	12.36	0.00	93.87	100.00	85.46	80.38	85.00	71.88	67.72	74.00	62.81
50	11.53	0.00	101.05	140.00	85.78	86.66	91.00	55.43	71.13	76.00	67.41
100	9.53	0.00	179.75	230.00	141.28	67.28	95.00	43.18	66.66	74.00	27.86
200	5.81	0.00	357.90	540.00	289.33	75.00	100.00	56.71	70.42	74.00	47.11

è direttamente proporzionale al numero di file, un comportamento tipico dell'esecuzione sequenziale. Le operazioni di `Status` (lettura), invece, mantengono latenze relativamente stabili (Figura 5.2).

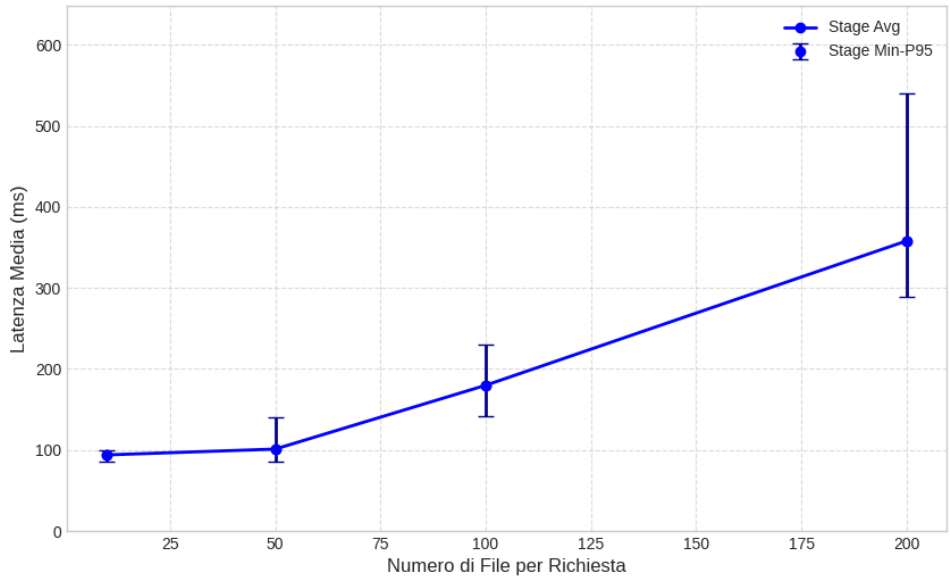

Figura 5.1: Latenza Stage (Sequenziale, 1 Utente)

5.1.3 Impatto della Parallelizzazione

Per valutare l'efficacia del refactoring, confrontiamo le prestazioni fissando un singolo utente concorrente nelle tre configurazioni, sia per la latenza dell'operazione Stage che per il numero di richieste al secondo totali (Figura 5.4 e Figura 5.5).

Nel caso della latenza dell'operazione Stage, il grafico non mostra apparenti cambiamenti delle performance, al contrario del grafico delle Richieste al secondo, che invece fa vedere un netto miglioramento delle prestazioni nella versione parallela rispetto a quella sequenziale. Tuttavia, in entrambi si osserva un fenomeno interessante: il miglioramento ottenuto passando dalla versione parallela *single-threaded* a quella *multi-threaded* (3 server threads) è marginale. Questo suggerisce

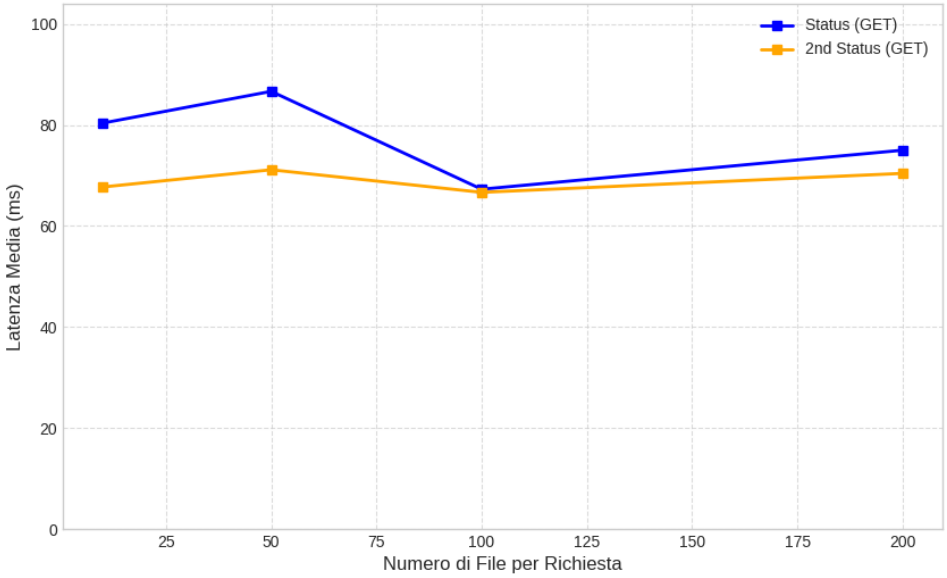

Figura 5.2: Latenza Status (Sequenziale, 1 Utente)

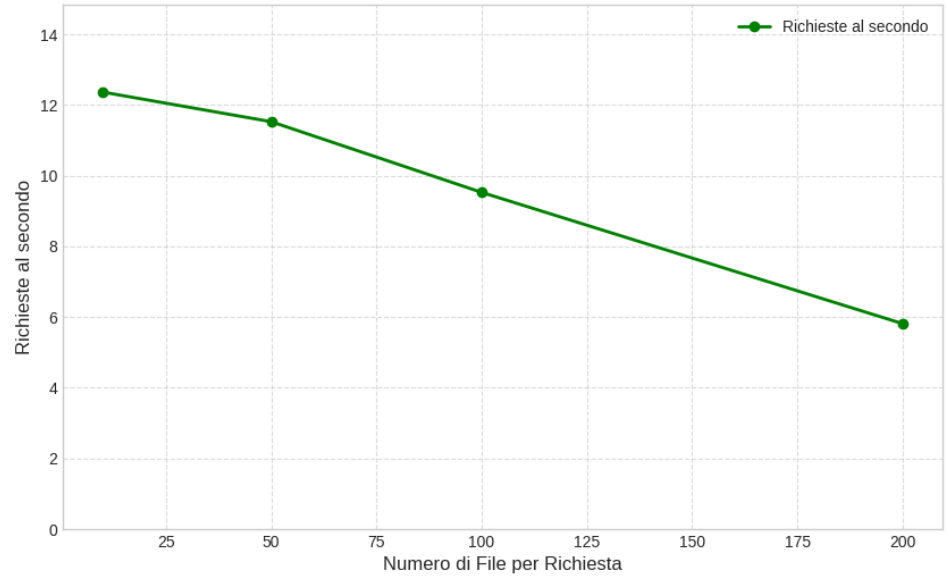

Figura 5.3: Variazione RPS al variare del carico (Sequenziale, 1 Utente)

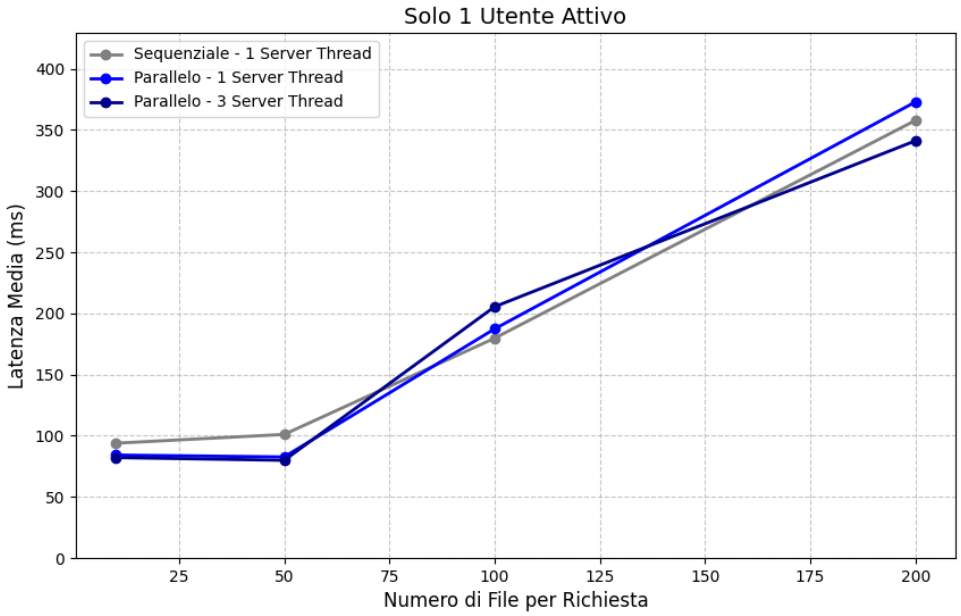

Figura 5.4: Confronto Latenza Stage: Sequenziale vs Parallelo (1 e 3 server threads)

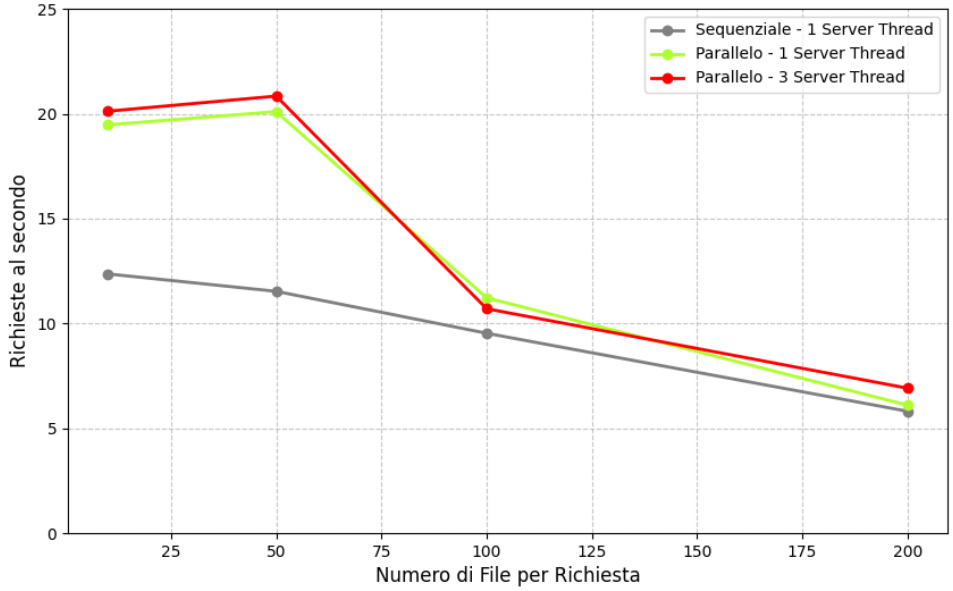

Figura 5.5: Confronto RPS: Sequenziale vs Parallelo (1 e 3 server threads)

che, per un singolo utente, il collo di bottiglia potrebbe essersi spostato dalla CPU all'I/O del file system o ai lock di sincronizzazione necessari per l'aggiornamento del database.

5.1.4 Scalabilità con Utenti Concorrenti

Aumentando il numero di utenti concorrenti (Figura 5.6), il sistema dimostra una buona capacità di scaling. Il throughput complessivo aumenta all'aumentare della concorrenza, indicando che il server riesce a gestire efficientemente richieste multiple parallele, saturando al meglio le risorse disponibili.

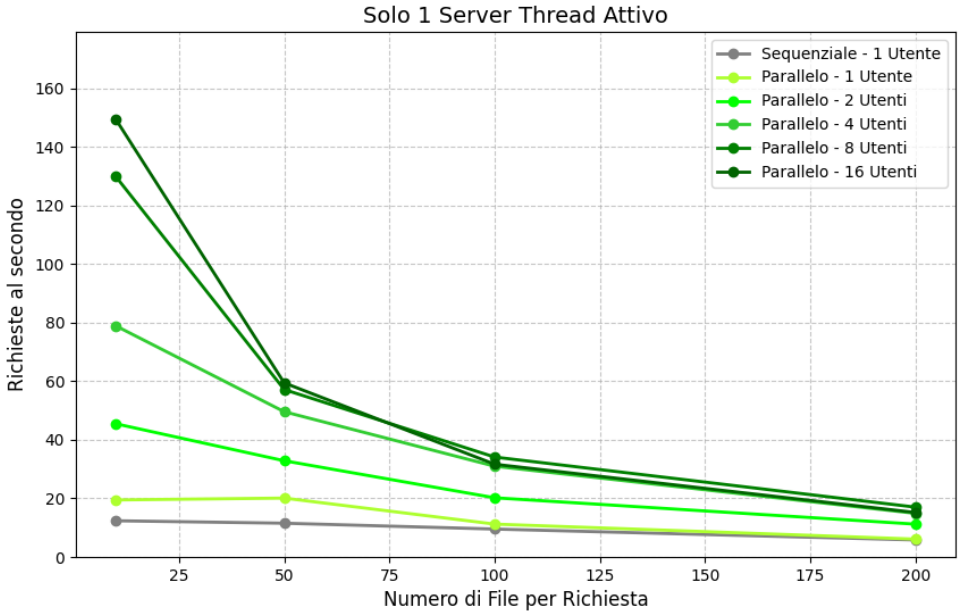

Figura 5.6: Scaling RPS al variare del carico per diversi utenti concorrenti

5.1.5 Stress Test Complessivo (200 Files/Request)

L'analisi finale si concentra sullo scenario più gravoso: richieste contenenti 200 file ciascuna. La Figura 5.7 riassume l'andamento del throughput per 4 categorie di esperimenti al variare degli utenti concorrenti. In questo confronto è stata tenuta in conto anche la configurazione del server con la versione sequenziale, ma con la modalità WAL attivata. Così facendo, abbiamo potuto avere una panoramica completa sull'impatto che ha sulle perfomance la connection pool per gli accessi al database, non implementata nelle configurazioni sequenziali.

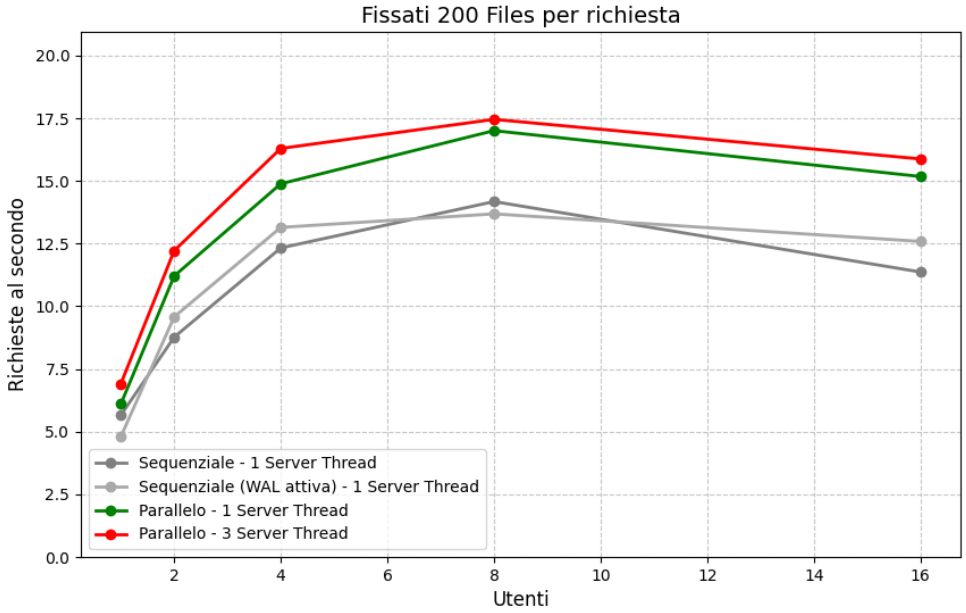

Figura 5.7: RPS Totali per categoria (Carico: 200 file/richiesta)

Si nota chiaramente il salto prestazionale delle due categorie parallele rispetto a quelle sequenziali. Tuttavia, superata la soglia degli 8 utenti concorrenti, si registra una flessione del throughput in tutte le configurazioni. Questo comportamento indica il raggiungimento di un punto di saturazione del sistema, probabilmente dovuto alla congestione del database. Inoltre, persiste il trend osservato precedentemente: lo scarto tra parallelo *single-threaded* e *multi-threaded* rimane contenuto, confermando che l'overhead di gestione dei thread o l'attesa su risorse condivise (mutex) limita il guadagno teorico del calcolo parallelo puro in questo specifico scenario.

Ad ogni modo, la Figura 5.8, anch'essa includendo la configurazione sequenziale con modalità WAL attiva, conferma il miglioramento dal punto di vista della latenza: la versione parallela mantiene tempi di risposta significativamente più bassi rispetto a quelle sequenziali anche sotto forte carico.

Questi ultimi confronti fanno capire che l'attivazione della modalità WAL in SQLite da sola non comporta un miglioramento sicuro delle perfomance, difatti le due configurazioni sequenziali differiscono di molto poco nei grafici. La scelta più conveniente, dunque, sembra essere sfruttarla al meglio, introducendo logiche di concorrenza negli accessi al database per trarne il più possibile i benefici.

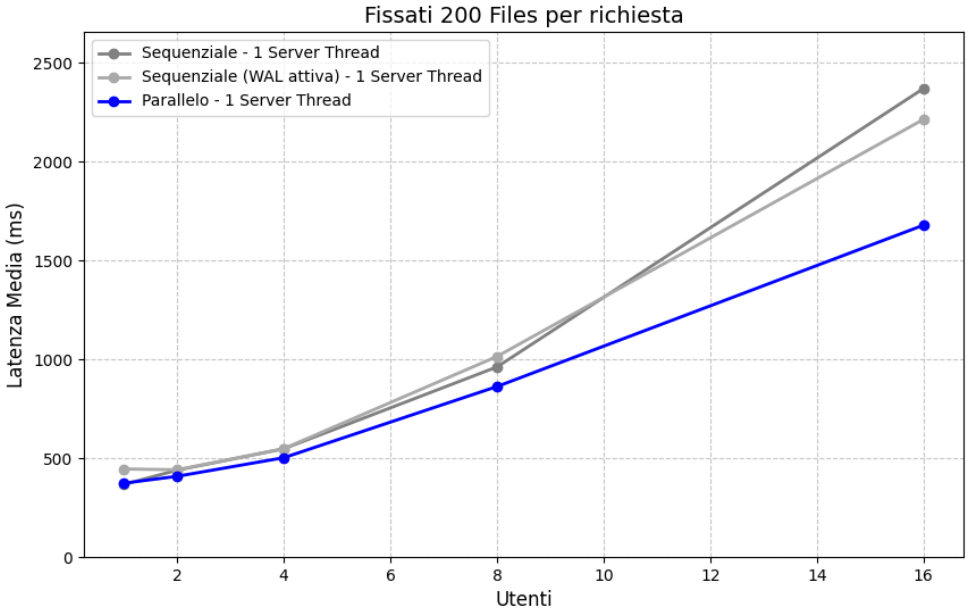

Figura 5.8: Confronto Latenza Media Stage al variare degli utenti (Sequenziale vs Parallelo)

5.2 Efficacia: Valutazione Qualitativa

Al di là delle metriche puramente numeriche, il lavoro svolto ha portato miglioramenti significativi nella qualità del software e nella sua manutenibilità.

- **Flessibilità:** L'architettura del software è ora ancor più predisposta al parallelismo essendo il passaggio da un'esecuzione sequenziale a una parallela finalmente implementato in maniera vera e propria.
- **Usabilità:** Dal punto di vista dell'utente finale (il client REST), l'aggiornamento è completamente trasparente. Le funzionalità e le interfacce API sono rimaste invariate, garantendo la totale retrocompatibilità, ma offrendo tempi di risposta ridotti per batch di grandi dimensioni.
- **Manutenzione:** La rimozione del lock globale sul database in favore di un *connection pool* rende il sistema più robusto e meno propenso a deadlock o colli di bottiglia artificiali, facilitando future espansioni o modifiche alla concorrenza.

In conclusione, sebbene i test di efficienza mostrino margini di miglioramento nell'utilizzo delle risorse multi-core, l'efficacia dell'intervento ha posto basi solide per l'evoluzione futura di StoRM Tape verso carichi di lavoro High-Throughput.

Capitolo 6

Conclusioni

6.1 Analisi critica del lavoro svolto

Il percorso di sviluppo descritto in questa tesi ha portato alla realizzazione di un sistema che, pur nelle sue evoluzioni e ripensamenti, ha raggiunto gli obiettivi prefissati di stabilità e performance. L'analisi finale del progetto ci permette di identificare chiaramente i punti di forza della soluzione e le aree in cui l'approccio teorico ha dovuto cedere il passo alla realtà empirica.

6.1.1 Motivi di orgoglio

Il principale motivo di orgoglio risiede nella robustezza architetturale raggiunta. Il sistema è in grado di gestire carichi di lavoro significativi mantenendo una reattività dell'interfaccia utente fluida, un risultato non scontato in applicazioni che interagiscono pesantemente con il disco.

In particolare, l'adozione di un design pattern che disaccoppia nettamente la logica di presentazione dalla logica di business e dall'accesso ai dati ha pagato dividendi in termini di manutenibilità. La scelta di utilizzare tecnologie moderne (C++ standard recente) ha garantito un codice pulito, espressivo e *Type-Safe*, riducendo drasticamente la categoria di bug legati alla gestione della memoria, come suggerito dalle **best practices** del C++ moderno [11, 15].

Un altro punto di eccellenza è stata la capacità di diagnosi e ottimizzazione delle performance. L'aver identificato nella modalità WAL (*Write-Ahead Logging*) di SQLite il vero "game changer" per le prestazioni dimostra una comprensione profonda non solo del codice scritto, ma dell'intero stack tecnologico su cui l'applicazione poggia. Questo ha permesso di ottenere accelerazioni nell'ordine di grandezza, superiori a qualsiasi micro-ottimizzazione algoritmica.

6.1.2 Considerazioni di modestia: il caso della parallelizzazione

La "modestia" tecnica di questo progetto emerge principalmente dall'analisi critica delle metodologie di ottimizzazione. Come descritto nel Capitolo 4, l'implementazione iniziale prevedeva l'uso di `std::execution::par` per parallelizzare le operazioni di elaborazione dati.

Tuttavia, i test empirici condotti sull'infrastruttura di destinazione hanno rivelato criticità impreviste. L'esecuzione parallela interagiva in modo erratico con il filesystem GPFS, in particolare a causa dei suoi complessi meccanismi di *caching* distribuito. Queste interazioni introducevano un livello di aleatorietà e rumore nelle misurazioni tale da rendere inaffidabili i benchmark, rischiando di confondere i risultati o falsare la valutazione delle altre ottimizzazioni (come l'adozione del WAL).

La decisione finale di disattivare tali ottimizzazioni parallele durante i test non è stata quindi dovuta a un limite intrinseco dell'algoritmo, ma a una scelta di rigore metodologico: era prioritario garantire la stabilità e la riproducibilità delle misurazioni in un ambiente di storage complesso. Riconoscere che la pulizia del dato sperimentale valeva più dell'applicazione cieca di pattern di concorrenza è stato un passo fondamentale verso la maturità ingegneristica del progetto.

6.2 Sviluppi futuri

Nonostante il sistema sia funzionale, lo sviluppo del software è un processo continuo e già nel breve termine si può individuare il punto principale su cui lavorare: l'effettiva inclusione delle ottimizzazioni con `std::execution::par`.

Sebbene siano state disattivate per le criticità sopra esposte, è fondamentale che si trovi un metodo per riuscire a misurare precisamente quelle singole funzioni aggirando il caching del file system. Così facendo, in caso di esiti positivi sul miglioramento delle performance, queste funzioni parallelizzate si potranno andare ad abilitare realmente nel codice di produzione, contribuendo all'ottimizzazione del servizio.

Bibliografia

[1] CERN. *High-Luminosity LHC*. CERN. 2025. URL: https://home.cern/science/accelerators/high-luminosity-lhc.

[2] CppReference. *std::execution::par*. 2025. URL: https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t.

[3] CrowCpp. *Crow: A Fast and Easy to use microframework for the Web*. 2025. URL: https://crowcpp.org/.

[4] John D. Hunter. *Matplotlib: Visualization with Python*. 2025. URL: https://matplotlib.org/.

[5] IBM. *IBM Spectrum Scale (GPFS)*. Filesystem parallelo ad alte prestazioni. 2024. URL: https://www.ibm.com/products/spectrum-scale.

[6] *ISO/IEC 14882:2017 Programming languages — C++*. International Organization for Standardization. 2017. URL: https://www.iso.org/standard/68564.html.

[7] *ISO/IEC 14882:2020 Programming languages — C++*. International Organization for Standardization. 2020.

[8] M. Jones, J. Bradley e N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. Mag. 2015. URL: https://tools.ietf.org/html/rfc7519.

[9] Christopher M. Kohlhoff. *Boost.Asio*. Libreria C++ per I/O asincrono e networking. 2025. URL: https://www.boost.org/doc/libs/release/doc/html/boost_asio.html.

[10] *Locust - An open source load testing tool*. 2025. URL: https://locust.io/.

[11] Scott Meyers. *Effective Modern C++*. O'Reilly Media, 2014.

[12] *Repository ufficiale con documentazione interna del progetto*. Materiale interno pubblicato. 2025. URL: https://github.com/glxcee/Stormtape-Optimization.

[13] *SQLite Database Engine*. 2025. URL: https://www.sqlite.org/.

[14] SQLite Development Team. *Write-Ahead Logging*. Documentazione ufficiale SQLite. 2025. URL: https://www.sqlite.org/wal.html.

[15] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley, 2013.

[16] The pandas development team. *pandas - Python Data Analysis Library*. 2025. URL: https://pandas.pydata.org/.

[17] WLCG. *WLCG Data Challenge 2024*. Zenodo Record. 2024. URL: https://zenodo.org/records/11444180.

[18] Worldwide LHC Computing Grid. *WLCG Tape REST API Specification*. Repository ufficiale. 2023. URL: https://github.com/wlcg-storage/wlcg-tape-rest-api.