Il Multi-threading nel linguaggio Julia: Principi e Applicazioni

Biagio Grimolizzi

Anno Accademico 2024/2025

Indice

Capitolo 1

Introduzione

Il linguaggio di programmazione Julia è stato progettato per rispondere alle esigenze del calcolo scientifico, dei big data e dell'intelligenza artificiale [1], combinando alte prestazioni con un'elevata produttività. Julia si distingue per la sua capacità di unire la velocità tipica dei linguaggi compilati, come C o Fortran, con la semplicità e la leggibilità dei linguaggi interpretati, come Python.

Negli ultimi anni, la programmazione parallela è diventata sempre più importante nello sviluppo del software moderno, poiché le applicazioni richiedono prestazioni elevate e una maggiore efficienza di elaborazione su sistemi multi-core. Con la fine della "*free lunch*" della Legge di Moore[2], il miglioramento della potenza computazionale non può più essere attribuito solo all'aumento della frequenza di clock dei processori. Questo ha spinto gli sviluppatori a ottimizzare l'uso delle architetture multi-core e a implementare paradigmi di programmazione parallela come il *multi-threading*.

In questo contesto, Julia, un linguaggio progettato specificamente per il calcolo tecnico e scientifico [3], ha introdotto un sistema innovativo per il *multi-threading*, caratterizzato da un'architettura componibile e altamente performante. Julia non solo semplifica l'implementazione del parallelismo, ma offre anche strumenti che permettono agli sviluppatori di scrivere codice parallelo scalabile senza compromettere la leggibilità e la manutenibilità.

Il nuovo modello, ispirato a paradigmi come Cilk e Intel Threading Building Blocks[6], consente di marcare porzioni di codice per l'esecuzione parallela tramite costrutti come `@spawn`. Questi task vengono gestiti automaticamente da un pianificatore dinamico, che distribuisce il lavoro tra i thread disponibili, rendendo l'interfaccia semplice e potente.

Questa tesina esplorerà nel dettaglio il sistema di *multi-threading* di Julia, mettendolo a confronto con altre tecnologie e mostrando esempi pratici di implementazioni. In particolare, l'obiettivo è analizzare:

- I principi fondamentali del *multi-threading* e le sue applicazioni.

- Le caratteristiche del sistema di *multi-threading* di Julia, incluso il modello di pianificazione dei task e la sincronizzazione tra thread.
- Casi pratici e benchmark che dimostrano l'efficacia e le prestazioni del *multi-threading* in Julia.
- Le sfide attuali e le prospettive future del *multi-threading* nel linguaggio Julia.

Capitolo 2

Fondamenti del Multi-threading

2.1 Il concetto di Multi-threading

Il *multi-threading* è una tecnica di programmazione parallela che consente l'esecuzione simultanea di più flussi di lavoro, detti *thread*, all'interno dello stesso programma. Questo approccio sfrutta al meglio l'architettura multi-core dei processori moderni, consentendo di ridurre i tempi di calcolo e migliorare le prestazioni complessive di un'applicazione.

Un esempio classico di utilizzo del *multi-threading* è l'elaborazione di grandi dataset, dove i dati possono essere suddivisi in blocchi e elaborati contemporaneamente su core diversi. Analogamente, nella grafica computerizzata, il rendering di immagini e video può beneficiare notevolmente dell'esecuzione parallela [4].

Ma non è sempre giusto o facile implementare il multi-threading: di questo parleremo nel capitolo 5, dove verranno analizzate le sfide e le difficoltà che accompagnano da sempre tutte le applicazioni che fanno uso di multi-threading.

2.2 Vantaggi del Multi-threading

I principali vantaggi del *multi-threading* includono:

- **Riduzione dei tempi di esecuzione**: Grazie alla distribuzione del carico di lavoro su più core, le applicazioni multi-threaded possono completare operazioni complesse in meno tempo.
- **Maggiore utilizzo delle risorse hardware**: I processori moderni sono progettati per supportare più core e thread. Il *multi-threading* garantisce che queste risorse non rimangano inutilizzate.

- **Reattività delle applicazioni**: In contesti interattivi, come le interfacce utente, il *multi-threading* consente di gestire attività pesanti in background senza bloccare l'interfaccia principale.

2.3 Confronto tra Multi-threading e Programmazione Asincrona

Il *multi-threading* è spesso confuso con la programmazione asincrona, ma i due approcci hanno obiettivi e metodologie differenti:

- **Multi-threading**: Consente l'esecuzione simultanea di calcoli intensivi su core multipli. È ideale per attività che richiedono un uso intensivo della CPU, come algoritmi di calcolo numerico o simulazioni fisiche.
- **Programmazione asincrona**: Si concentra sulla gestione non bloccante delle operazioni di I/O, come la lettura di file o la comunicazione di rete. Questo approccio utilizza un singolo thread per gestire molteplici eventi, evitando che il programma rimanga inattivo durante l'attesa.

Un esempio di linguaggio che adotta principalmente la programmazione asincrona è Node.js, progettato per applicazioni web ad alta intensità di I/O. Julia, invece, è ottimizzato per il *multi-threading*, poiché si rivolge principalmente a domini scientifici e tecnici dove i calcoli intensivi sono centrali.

Capitolo 3

Tecniche e API Multi-threading di Julia

Il linguaggio Julia offre molti strumenti per sfruttare il multi-threading, fornendo agli sviluppatori un controllo granulare sulle risorse dei processori moderni.

3.1 Configurazione dei Thread

Julia avvia, per impostazione predefinita, un singolo thread di esecuzione. Per abilitare l'uso di più thread, è necessario configurare l'ambiente in uno dei seguenti modi:

- Utilizzando il parametro `-t` o `-threads` nella riga di comando:

```
julia --threads=auto

```

  Questo comando attiva un numero di thread pari ai core fisici disponibili sulla macchina.

- Definendo la variabile d'ambiente `JULIA_NUM_THREADS` prima dell'avvio di Julia:

```
export JULIA_NUM_THREADS=4

```

Il numero di thread attivi può essere verificato all'interno di Julia con il comando:

```
Threads.nthreads()
```

3.2 Macro Fondamentali per il Multi-threading

3.2.1 `@threads`: Parallelizzazione di Loop

La macro `@threads` permette di distribuire automaticamente le iterazioni di un ciclo for tra i thread disponibili. Ad esempio:

```
using Base.Threads

a = zeros(Int, 10)

Threads.@threads for i = 1:10
    a[i] = Threads.threadid()  # ritorna l'id del thread che sta eseguendo il
    codice
end

println(a)
```

Un possibile output di quest'esempio, eseguito su 8 threads, potrebbe essere il seguente:

```
    [3.0, 3.0, 4.0, 4.0, 1.0, 2.0, 7.0, 8.0, 6.0, 5.0]
```

Possiamo vedere come l'accesso all'array da parte dei threads è totalmente casuale ed è per questo che bisogna far uso degli strumenti di sincronizzazione che Julia mette a disposizione e di cui parleremo nel paragrafo successivo 3.3.

3.2.2 `@spawn`: Creazione di Task Paralleli

La macro `@spawn` consente di creare task che possono essere schedulati dinamicamente su qualsiasi thread disponibile:

```
function sum_single(a)
    s = 0
    for i in a
        s += i
    end
    s
end


function sum_multi(a)
    chunks = Iterators.partition(a, length(a) / Threads.nthreads())
```

```
12      tasks = map(chunks) do chunk
13          Threads.@spawn sum_single(chunk)
14      end
15      chunk_sums = fetch.(tasks)
16      return sum_single(chunk_sums)
17  end
18
19
20  println(sum_multi(1:1_000_000))
```

Spiegazione del codice

- `function sum_single(a)` calcola la somma di tutti gli elementi nell'array passato come parametro a; Usa un semplice ciclo for per iterare sugli elementi.
- `function sum_multi(a)` Divide l'array a in più blocchi (o chunk) di dimensione uguale o simile; Per ogni blocco (chunk), crea un task parallelo usando la macro `@spawn` che calcola su un thread separato la somma degli elementi del suo blocco.
- `Iterators.partition()` Questa funzione in Julia viene utilizzata per dividere, in questo caso, un'array in blocchi più piccoli di dimensioni uguali, risultando particolarmente utile per l'elaborazione parallela.
- `chunk_sums = fetch.(tasks)` infine attende la fine di ogni task e combina i risultati parziali (le somme di ciascun blocco) calcolandone la somma finale.

3.2.3 Threadpools

In Julia, i thread pool offrono un meccanismo per gestire l'esecuzione parallela dei task su gruppi separati di thread, migliorando l'efficienza e la gestione delle risorse computazionali. Attualmente, Julia supporta due tipi principali di pool di thread: il pool predefinito (***default pool***) e il pool interattivo (***interactive pool***).

Il **pool predefinito** è utilizzato per eseguire task generici e operazioni di calcolo intensivo. È ideale per operazioni che non richiedono interazione immediata con l'utente, come calcoli numerici complessi o simulazioni.

Il **pool interattivo** è progettato per operazioni che richiedono reattività immediata, come interazioni con l'utente o risposte rapide a input dinamici. È utile per gestire attività che devono essere completate rapidamente e non possono attendere il completamento di operazioni lunghe in esecuzione nel pool predefinito. Il pool interattivo ha la priorità su quello predefinito.

Julia fornisce diversi modi per gestire i pool:

- `@spawn :interactive f()` per assegnare direttamente una funzione f al pool interattivo (interactive).
- `julia -threads 3,1` all'avvio di julia puoi riservare uno o più thread al pool interattivo, in questo caso 3 per il predefinito e 1 per l'interattivo.
- `export JULIA_NUM_THREADS=3,1` la variabile d'ambiente vista anche in precedenza `JULIA_NUM_THREADS` può essere usata anche diversamente come, in questo caso, per riservare 1 thread al pool interattivo.

3.2.4 `@threadcall`: Esecuzione di Funzioni Non-Julia su Thread Specifici

La macro `@threadcall` è utile per eseguire funzioni scritte in altri linguaggi (ad esempio, C o Fortran) su un thread specifico. Questo è particolarmente importante per chiamate che bloccano, come operazioni di I/O:

```
Threads.@threadcall :default ccall((:sleep, "libc"), Cvoid, (UInt32,), 1)
```

In questo esempio, la chiamata `ccall` a una funzione di libreria viene eseguita in modo non bloccante, sul threadpool di default, permettendo agli altri thread di continuare l'elaborazione.

3.3 Sincronizzazione e Concorrenza

Quando più thread accedono simultaneamente a risorse condivise, come variabili globali o strutture dati, possono verificarsi problemi di concorrenza. Julia offre diversi strumenti per gestire questi scenari:

3.3.1 Lock (ReentrantLock)

Un *lock* impedisce che più thread accedano contemporaneamente a una sezione critica del codice. Julia fornisce `ReentrantLock`, che consente a un thread di ottenere un lock senza preoccuparsi di deadlock in caso di ricorsione:

```
lock = ReentrantLock()

@lock lock begin
    println("Inizio sezione critica")
    sleep(1)
    println("Fine sezione critica")
end
```

- Solo un thread può entrare nella sezione protetta alla volta.

- L'uso di `@lock` semplifica la sintassi e riduce il rischio di dimenticare di rilasciare manualmente il lock.

3.3.2 Operazioni Atomiche

Le operazioni atomiche consentono a più thread di accedere e modificare una variabile condivisa senza bisogno di lock espliciti. Una variabile atomica si può modificare solo con operazioni atomiche e le operazioni atomiche possono essere eseguite solo su variabili atomiche. Ad esempio:

```
acc = Threads.Atomic{Int}(0)

Threads.@threads for i in 1:100
    Threads.atomic_add!(acc, i)
end

println("Somma totale: $(acc[])")
```

- `acc = Threads.Atomic{Int}(0)` inizializza una variabile atomica a 0 di tipo Int
- `Threads.atomic_add!` incrementa la variabile assicurandosi che l'incremento venga eseguito in modo sicuro.
- L'operatore `[]` restituisce il valore attuale della variabile atomica.

3.3.3 Channel per la Comunicazione

I *Channel* offrono un modo thread-safe per scambiare dati tra thread. Sono particolarmente utili per implementare code di lavoro (possono essere visti come le pipe per i processi in C):

```
ch = Channel{String}(10)

Threads.@spawn for i in 1:5
    put!(ch, "Messaggio dal thread $i")
end

Threads.@spawn begin
    while isopen(ch)
        println(take!(ch))
    end
end
```

- `put!` inserisce un messaggio nel canale.
- `take!` rimuove e restituisce un messaggio dal canale.
- `isopen` verifica se il canale è ancora aperto.

3.3.4 Semaphore

I *semaphore* limitano il numero di thread che possono accedere contemporaneamente a una risorsa condivisa. Julia fornisce `Semaphore` per questo scopo:

```
using Base

sem = Base.Semaphore(2)  # Permette l'accesso a massimo 2 thread
    contemporaneamente

Threads.@threads for i in 1:5
    Base.acquire(sem)
    println("Thread $i entra nella sezione critica")
    sleep(1)
    println("Thread $i esce dalla sezione critica")
    Base.release(sem)
end
```

- `acquire` decrementa il contatore del semaforo e blocca il thread se il valore è zero.
- `release` incrementa il contatore, consentendo ad altri thread di accedere.

Questo meccanismo è utile per gestire pool di risorse limitate, come connessioni a un database.

3.3.5 Differenze e Scenari di Utilizzo

Tecnica	**Quando Usarla**	**Pro**	**Contro**
Lock	Risorse complesse o sezioni critiche non atomiche	Facile da implementare	Rischio di deadlock, overhead
Operazioni Atomiche	Operazioni semplici su variabili condivise	Alte prestazioni, privo di deadlock	Non adatte a operazioni complesse
Channel	Comunicazione tra thread o pipeline	Thread-safe, flessibile	Maggiore complessità
Semaphore	Risorse limitate con accesso simultaneo controllato	Controllo preciso delle risorse	Più complesso da gestire

Tabella 3.1: Confronto tra le tecniche di sincronizzazione in Julia.

3.4 Avvertenze sul Multi-threading in Julia

Al momento, la maggior parte delle operazioni della runtime e delle librerie standard possono essere utilizzate in modo "*thread-safe*", purché il codice sia

privo di data race. Tuttavia, ci sono ancora alcune aree in cui il supporto ai thread è in fase di stabilizzazione. La programmazione multi-threading presenta molte difficoltà intrinseche e, se un programma che utilizza i thread mostra comportamenti insoliti o indesiderati (ad esempio, crash o risultati inaspettati), è giusto sospettare che ci siano interazioni problematiche tra i thread.

Ecco alcune limitazioni e avvertenze specifiche da tenere a mente quando si utilizzano i thread in Julia:

- I tipi di collezione di base richiedono un blocco manuale (lock) se utilizzati contemporaneamente da più thread, soprattutto quando almeno uno di essi modifica la collezione. Esempi comuni includono l'uso di `push!` sugli array o l'inserimento di elementi in un `Dict`.
- La pianificazione (schedule) dei task creati con `@spawn` è non deterministica e non dovrebbe essere considerata prevedibile.
- I task che richiedono calcoli intensivi e non effettuano allocazioni di memoria possono impedire l'esecuzione del garbage collector su altri thread che stanno allocando memoria. In questi casi, potrebbe essere necessario inserire manualmente una chiamata a `GC.safepoint()` per consentire l'esecuzione del garbage collector. Questa limitazione, dice la documentazione di Julia, verrà rimossa in futuro [5].
- È meglio evitare di eseguire operazioni di alto livello in parallelo, come `include` o `eval` per definire tipi, metodi e moduli.

3.4.1 Migrazione dei Task

Dopo che un task inizia l'esecuzione su un determinato thread, potrebbe essere spostato su un thread diverso se il task esegue un'operazione di yield (rilascio del controllo).

Tali task possono essere stati avviati con `@spawn` o `@threads`, sebbene l'opzione di schedulazione `:static` per `@threads` blocchi l'assegnazione al thread specifico.

Questo significa che, nella maggior parte dei casi, la funzione `threadid()` non dovrebbe essere considerata come un valore costante all'interno di un task e, di conseguenza, non dovrebbe essere utilizzata per indicizzare un vettore di buffer o oggetti con stato.

Capitolo 4

Casi Pratici e Benchmark

In questo capitolo, presentiamo nuovi esempi pratici di utilizzo del *multi-threading* in Julia, insieme a benchmark che dimostrano i miglioramenti di prestazioni ottenibili. Gli esempi sono stati testati su di un MacBookPro M4PRO 12 core. Inoltre per tutti i benchmark si è usato `@btime`, una macro della libreria BenchmarkTools di Julia che automaticamente esegue più volte la funzione da testare e mostra il valore medio.

4.1 Benchmark di Riempimento di un Vettore: Multi-threaded vs Single-threaded

Un esempio semplice, ma dai risultati sorprendentemente efficaci per dimostrare la potenza del multi-threading, è il popolamento di un vettore di grandi dimensioni.

Implementazione

```
using BenchmarkTools

n = 1_000_000_000

# Multi-threaded
myvector = Vector{Int8}(undef, n)
@btime Threads.@threads for i in 1:n
    myvector[i] = Int8(i % 128)
end

# Single-threaded
myvector2 = Vector{Int8}(undef, n)
@btime for i in 1:n
    myvector2[i] = Int8(i % 128)
end
```

Spiegazione

In questo esempio è stato utilizzato un valore molto grande per `n` (1 miliardo di elementi) al fine di evidenziare le differenze di prestazioni tra l'approccio multi-threaded e quello single-threaded. Per ottimizzare l'utilizzo della memoria, ogni elemento del vettore è stato dichiarato di tipo `Int8`, che occupa solo 1 byte.

L'approccio multi-threaded sfrutta la parallelizzazione fornita da `Threads.@threads`, suddividendo il lavoro tra i thread disponibili per velocizzare l'inizializzazione del vettore. Al contrario, l'approccio single-threaded esegue lo stesso compito sequenzialmente, risultando significativamente più lento per un numero elevato di elementi.

Benchmark

Eseguendo i due cicli `for`, il primo utilizzando sei e poi dodici thread, e il secondo in modalità single-threaded, sono stati registrati i seguenti tempi di esecuzione:

- **Singolo thread**: 14.4 secondi.
- **Sei thread**: 3.8 secondi.
- **Dodici thread**: 2.8 secondi.

Come si può osservare, l'approccio multi-threaded riduce drasticamente il tempo di esecuzione rispetto al single-threading. L'aumento del numero di thread migliora ulteriormente le prestazioni, sebbene con un beneficio marginale quando si passa da sei a dodici thread, a causa di overhead aggiuntivi e limitazioni nell'accesso alla memoria condivisa.

4.2 Somma di tutti gli elementi in un'array

Vediamo ora un altro esempio in cui andiamo a calcolare la somma di tutti gli elementi in un array utilizzando il multi-threading con varie configurazioni.

Implementazione

```
using Base.Threads
using BenchmarkTools

function sum_single(a)
    s = 0
    for x in a
        s += x
    end
```

```
    return s
end

function sum_multi(a)
    chunks = Iterators.partition(a, length(a) / Threads.nthreads())
    tasks = map(chunks) do chunk
        Threads.@spawn sum_single(chunk)
    end
    chunk_sums = fetch.(tasks)
    return sum_single(chunk_sums)
end

@btime sum_multi(1:999_999_999_999_999_999)
```

Spiegazione

Questo esempio mostra come calcolare la somma di tutti gli elementi in un array utilizzando il multi-threading.

- **Divisione dell'array:** L'array viene suddiviso in blocchi più piccoli utilizzando `Iterators.partition`, assegnando ogni blocco a un thread disponibile.
- **Parallelizzazione con @spawn:** Per ogni blocco, viene avviato un task parallelo che calcola la somma parziale degli elementi tramite la funzione `sum_single`.
- **Raccolta dei risultati:** I risultati parziali calcolati da ciascun task vengono recuperati con `fetch` e poi sommati insieme per ottenere il risultato finale.

Questo approccio sfrutta il multi-threading per suddividere il carico di lavoro e velocizzare il calcolo della somma totale.

Benchmark

- **Singolo thread**: 10.1 microsecondi.
- **Due thread**: 3.1 microsecondi.
- **Quattro thread**: 1.4 microsecondi.
- **Sei thread**: 3.5 microsecondi.
- **Dodici thread**: 7.8 microsecondi.

Possiamo notare come a causa dell'overhead, passati i quattro thread si va incontro ad un peggioramento delle prestazioni.

4.3 Benchmark con costrutti di sincronizzazione

In questa sezione vengono presentati due esempi pratici che mostrano l'utilizzo di costrutti di sincronizzazione nel contesto del multi-threading in Julia. I test sono stati eseguiti per confrontare le prestazioni di due approcci diversi: l'uso di una variabile atomica e l'impiego di un lock esplicito. Entrambi i metodi servono a garantire la correttezza dell'esecuzione in ambienti concorrenti, prevenendo le condizioni di race. Tuttavia, come mostrano i benchmark, il loro comportamento in termini di prestazioni varia sensibilmente a seconda del numero di thread impiegati e della natura dell'operazione eseguita.

4.3.1 Implementazione con variabile atomica

```
using Base.Threads
using BenchmarkTools

function threaded_sum(arr)
    s = Atomic{Int}(0)  # Variabile atomica per evitare data races
    Threads.@threads for i in arr
        atomic_add!(s, i)
    end
    return s[]
end

arr = collect(1:10^6)

@btime threaded_sum(arr)
```

Spiegazione

Questo esempio utilizza una variabile atomica per accumulare in modo sicuro la somma degli elementi di un array.

- **Variabile Atomica:** La variabile `s` è dichiarata come `Atomic{Int}(0)`, garantendo che ogni operazione di incremento sia atomica, ovvero eseguita senza interruzioni da parte di altri thread.
- **Parallelizzazione con @threads:** Il ciclo `@threads for` divide automaticamente il lavoro tra i thread disponibili.
- **Operazione atomica:** La funzione `atomic_add!` incrementa il valore della variabile atomica in modo sicuro, evitando conflitti tra i thread.

Benchmark

- **Singolo thread**: 1.5 millisecondi.

- **Sei thread**: 8.5 millisecondi.
- **Dodici thread**: 31.4 millisecondi.

4.3.2 Implementazione con `@lock`

```
using Base.Threads
using BenchmarkTools

function increment_with_lock(n)
    my_lock = ReentrantLock()
    counter = 0

    Threads.@threads for _ in 1:n
        lock(my_lock) do
            counter += 1
        end
    end

    return counter
end

n = 10^6
@btime result = increment_with_lock(n)
```

Spiegazione

Questo esempio utilizza un lock per garantire che l'accesso alla variabile condivisa `counter` sia sicuro.

- **ReentrantLock:** Viene creato un oggetto lock con `ReentrantLock()`, che consente di proteggere sezioni critiche del codice.
- **Blocco Protetto:** Ogni incremento della variabile `counter` avviene all'interno di un blocco protetto dalla funzione `lock`, garantendo che solo un thread alla volta possa modificarla.

Benchmark

- **Singolo thread**: 16.9 millisecondi.
- **Sei thread**: 124.8 millisecondi.
- **Dodici thread**: 152.3 millisecondi.

4.3.3 Overhead e Prestazioni nei Costrutti di Sincronizzazione

Entrambi gli approcci testati evidenziano un significativo overhead legato alla sincronizzazione, che aumenta con il numero di thread impiegati.

- **Variabile Atomica**: Sebbene sia generalmente più veloce rispetto ai lock, l'accesso concorrente a una variabile atomica introduce latenza a causa della gestione delle operazioni atomiche, rendendo le prestazioni meno scalabili all'aumentare del numero di thread.
- **Lock Espliciti**: L'uso di lock è adatto per sezioni critiche complesse, ma introduce un overhead ancora maggiore poiché ogni thread deve attendere il rilascio del lock. Questo porta a un degrado significativo delle prestazioni man mano che il numero di thread aumenta.

In generale, l'overhead della sincronizzazione nei task multi-threaded può limitare l'efficacia del parallelismo, specialmente quando si utilizzano operazioni semplici come incrementi o somme. È quindi essenziale valutare attentamente il tipo di sincronizzazione da adottare in base al carico di lavoro e al numero di thread disponibili.

Capitolo 5

Sfide e Prospettive

Nonostante i numerosi vantaggi offerti dal *multi-threading*, l'implementazione di applicazioni parallele in Julia non è priva di difficoltà. In questo capitolo analizzeremo alcune delle principali sfide tecniche che gli sviluppatori affrontano nell'utilizzo del *multi-threading*, oltre a elencare le prospettive future per migliorare ulteriormente questo linguaggio.

5.1 Sfide Tecniche

5.1.1 Sincronizzazione e Concorrenza

La sincronizzazione tra thread è uno dei problemi più complessi del *multi-threading*. Quando più thread accedono simultaneamente a risorse condivise, possono verificarsi conflitti che portano a risultati imprevedibili o a comportamenti errati come:

- **Race Condition**: Quando due o più thread accedono contemporaneamente a una risorsa condivisa e almeno uno di essi la modifica, causando risultati imprevedibili o errati. Questo accade perché l'ordine di esecuzione dei thread non è deterministico, portando a conflitti durante la lettura o scrittura dei dati.
- **Deadlock**: Quando due o più thread rimangono bloccati in attesa di risorse detenute reciprocamente.
- **Starvation**: Quando un thread non riesce mai a ottenere l'accesso a una risorsa condivisa.

Gli sviluppatori devono pianificare con attenzione l'uso delle risorse condivise per evitare tali situazioni. Ed è per questo che Julia fornisce strumenti come `ReentrantLock`, `Threads.Atomic`, `Semaphore` per gestire la sincronizzazione come abbiamo visto nel capitolo 3 delle API.

5.1.2 Overhead di Gestione

Il *multi-threading* introduce un overhead computazionale, come abbiamo potuto vedere dai test fatti nel capitolo precedente 4, dovuto al coordinamento dei thread e alla gestione dei task. In alcune applicazioni, specialmente quelle con carichi di lavoro relativamente piccoli, questo overhead può ridurre i vantaggi delle esecuzioni parallele ed è quindi importante capire se è effettivamente vantaggioso usare il multi-threading in alcuni contesti.

5.2 Prospettive Future

Julia è in continua evoluzione e la comunità di sviluppatori sta lavorando attivamente per affrontare le sfide esistenti e migliorare il supporto al *multi-threading*. Di seguito sono elencate alcune delle principali direzioni di sviluppo [6].

5.2.1 Alcune aree di ricerca

- Ottimizzazione delle prestazioni per il cambio di task e la latenza di I/O.
- Aggiunta di parallelismo alla libreria standard. Molte operazioni comuni, come l'ordinamento e il broadcasting sugli array, potrebbero ora utilizzare più thread internamente.
- Fornire più operazioni atomiche a livello di Julia.
- Utilizzare più thread nel compilatore.
- Loop e riduzioni paralleli più performanti, con più opzioni di *scheduling*.
- Permettere l'aggiunta di ulteriori thread durante il tempo di esecuzione.
- Strumenti di debug migliorati.
- Libreria standard di strutture dati thread-safe.
- Fornire *schedulers* alternativi.
- Esplorare l'integrazione con il TAPIR parallel IR [7].

Capitolo 6

Conclusioni

Nonostante le sfide tecniche, il *multi-threading* in Julia rappresenta una tecnologia avanzata che ha già dimostrato di essere efficace in una vasta gamma di applicazioni.

Julia continua a crescere grazie al contributo della sua comunità di sviluppatori e utenti. L'implementazione di nuove funzionalità, combinate con il perfezionamento di quelle esistenti, garantirà che Julia rimanga all'avanguardia nel campo del calcolo parallelo e scientifico.

In conclusione, il *multi-threading* in Julia rappresenta un esempio brillante di come un linguaggio di programmazione possa rendere accessibile una tecnologia complessa, senza compromettere le prestazioni.

L'approfondimento delle nuove funzionalità di Julia e l'applicazione del *multi-threading* in scenari reali potrebbero costituire interessanti direzioni per future ricerche.

Julia dimostra che il *multi-threading* non deve essere complicato: può essere potente, elegante e alla portata di tutti, rendendolo una scelta valida per affrontare le sfide del calcolo moderno.

Bibliografia

[1] Matheus S. Serpa, Arthur M. Krause, Eduardo H.M. Cruz, Philippe Olivier Alexandre Navaux, Marcelo Pasin, and Pascal Felber. Optimizing machine learning algorithms on multi-core and many-core architectures using thread and data mapping. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 329–333, 2018.

[2] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm/, 2005.

[3] Mohamad Bitar. Rust and Julia for Scientific Computing . *Computing in Science & Engineering*, 26(01):72–76, January 2024.

[4] Eric Debts and Fulvio Moschetti. Multithreading for video processing applications running on pc workstations. In *2000 10th European Signal Processing Conference*, pages 1–4, 2000.

[5] Julia 1.11 documentation multi threading. https://docs.julialang.org/en/v1/manual/multi-threading/#man-multithreading, 2019.

[6] Kiran Pamnany Jeff Bezanson, Jameson Nash. Announcing composable multi-threaded parallelism in julia. https://julialang.org/blog/2019/07/multithreading/, 2019.

[7] Valentin Churavy. Experimental tapir support. https://github.com/JuliaLang/julia/pull/31086, 2019.