

Università Degli Studi di Fisciano

Dipartimento di Informatica

Corso di Laurea in Informatica

Modello di simulazione basato su agenti con Rust

Relatore: Alberto Negro
Relatore: Carmine Spagnuolo
Autore: Daniele De Vinco
Matricola: 0512104573
Anno Accademico 2018-19

1 Introduzione

Al giorno d'oggi analizzare sistemi reali e fenomeni complessi sta diventando una sfida essenziale per riuscire ad ottenere modelli predittivi efficaci. In molti campi, dall'economia all'ecologia, è necessario riuscire a collezionare dei dati affidabili che possano aiutare la ricerca a progredire più velocemente. Per questo negli ultimi anni le comunità scientifiche stanno esplorando la possibilità di ricreare questi fenomeni con dei modelli di simulazione basati su agenti, in gergo ABM, ovvero degli strumenti innovativi che possono modellare sistemi complessi reali, grazie anche alla potenzialità che il mondo tecnologico riesce ad offrire in questo momento. Questo nuovo metodo è molto più semplice ed efficiente dei tradizionali metodi di simulazione. Usare questo processo per effettuare simulazioni in maniera efficace può ridurre enormemente il carico computazionale e risultare più semplice da utilizzare per simulare sistemi complessi su larga scala.
Questi modelli sono composti solitamente da:

1. Agenti
2. Relazioni
3. Regole

1. Gli agenti definiscono la popolazione che deve essere analizzata. Possono essere indipendenti o relazionarsi tra di loro, eseguendo funzioni complesse con semplici interazioni.
2. Le relazioni definiscono quelle che possono essere le possibili interazioni tra gli agenti.
3. Le regole descrivono il comportamento che gli agenti devono avere come risultato di un'interazione.

La maggior parte di questi ABM sono implementati tramite software che effettuano le simulazioni in maniera concorrente o parallela. Il problema di questi modelli è che spesso le tecnologie con cui sono realizzate presentano molti inconvenienti quando si tratta di computare dati globali in maniera efficiente. Nell'ottica di ottimizzare e migliorare questi modelli di simulazione, la tesi illustra la possibilità di utilizzare Rust come nuovo linguaggio di programmazione per poter effettuare queste computazioni in maniera efficiente rispetto ad altri competitor, evidenziandone le performance in questo specifico campo. In molti casi infatti, per ottenere dei buoni risultati, vengono adoperate soluzioni con il C o C++, che consentono al programmatore di effettuare operazioni a basso livello per gestire al meglio la memoria. E' difficile anche per un

programmatore esperto sviluppare codice pulito e ad alte performance con questi linguaggi. Il problema da risolvere è computare in maniera continua dei dati globali durante lo svolgimento di una simulazione che può avere un numero molto alto di agenti, evitando che ci siano delle inconsistenze tali da falsare i risultati.
Da dove nasce e perché abbiamo deciso di utilizzare Rust?
Rust è un progetto Open Source, inizialmente sviluppato da Mozilla Foundation, creato con lo scopo di sviluppare sistemi sicuri, affidabili e ad alte prestazioni. Rust è stato scelto per numerosi progetti fin dalla sua creazione. Principalmente, per i suoi ideatori, è il cuore di Servo, il parallel browser engine di Mozilla, unico nel suo genere per la capacità di gestire in maniera efficiente e sicura processi concorrenti per il rendering. Dalla sua prima release stabile nel 2015, l'utilizzo di Rust è cresciuto notevolmente nel panorama scientifico - è il linguaggio più amato da parte degli sviluppatori dal 2016[1] - riuscendo a trovare applicazione nei campi più disparati grazie alle sue particolari proprietà, dallo sviluppo di sistemi operativi(Redox, BlogOS, intermezzOS, QuiltOS, RustOS, Rux, Tefflin, and Tock) all'impiego nel mondo videoludico(Piston, Amethyst, and GGEZ). L'interesse non è nato solamente per gli sviluppatori curiosi di scoprire un nuovo linguaggio, ma soprattutto per le aziende. L'elenco delle compagnie che utilizzano Rust è in crescita e si possono trovare nomi di spicco come Samsung o Cloudflare[2]. La maggior parte di esse sta mirando a sfruttare le peculiarità di Rust per rimpiazzare linguaggi come C in settori critici dove la sicurezza, l'accesso e la memorizzazione dei dati è importante.

Come si può notare dalla tabella 1, che esplica le performance di Rust in confronto a quelle del C in relazione all'implementazione di alcuni algoritmi, le prestazioni di Rust sono molto simili a quelle del C, in alcuni casi anche superiori[3].
Questo linguaggio lascia al programmatore la facoltà di scegliere la modalità di sviluppo del proprio software, ottenendo risultati migliori a seconda dell'ottimizzazione desiderata. Cargo, il package manager di Rust, permette di svolgere tutte queste funzioni e richiamare il compilatore rustc una volta effettuate le varie ottimizzazioni.

[1] https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted

[2] https://prev.rust-lang.org/en-US/friends.html

[3] https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html

reverse-complement

source	secs	mem	gz	cpu	cpu load
Rust	**1.61**	994,796	1376	2.74	28% 24% 96% 24%
C gcc	1.76	994,488	1438	3.98	97% 43% 44% 46%

binary-trees

source	secs	mem	gz	cpu	cpu load
Rust	**3.50**	134,208	754	12.73	89% 89% 99% 89%
C gcc	3.73	116,600	836	11.74	76% 71% 98% 72%

spectral-norm

source	secs	mem	gz	cpu	cpu load
Rust	**1.97**	2,312	1126	7.86	100% 100% 99% 99%
C gcc	1.98	1,168	1139	7.87	100% 99% 100% 99%

Tabella 1

Dalla tabella 2 si possono notare le grandi differenze di prestazioni tra Rust e Java[4]. E' da tenere in considerazione che Rust eredita alcune caratteristiche del paradigma ad oggetti, e in alcuni casi specifici è molto più conveniente utilizzare quest'ultimo che un linguaggio puro orientato ad oggetti. Nonostante ci siano delle evidenti limitazioni, che verranno evidenziate più approfonditamente nei prossimi capitoli, dovute alla struttura particolare di Rust, la possibilità di poter sfruttare alcune peculiarità del paradigma ad oggetti con così alte prestazioni, rende questo linguaggio molto appetibile in numerosi settori.

[4] https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-java.html

regex-redux

source	secs	mem	gz	cpu	cpu load
Rust	**2.51**	191,144	765	3.93	16% 84% 58% 0%
Java	10.50	571,736	929	31.63	71% 73% 66% 92%

n-body

source	secs	mem	gz	cpu	cpu load
Rust	**5.72**	916	1753	5.72	0% 100% 1% 0%
Java	22.00	32,272	1489	22.07	1% 0% 0% 100%

mandelbrot

source	secs	mem	gz	cpu	cpu load
Rust	**2.00**	11,740	1007	7.93	99% 100% 99% 99%
Java	6.96	76,316	796	27.06	97% 97% 97% 97%

Tabella 2

In questa tesi, nella sezione 2 analizzeremo quelle che sono le proprietà specifiche del linguaggio e metteremo a confronto Rust con altri linguaggi di programmazione, mostrando le differenze sostanziali nell'architettura. Nella sezione 3 verrà presentata la versione Alpha 1.0 della libreria implementata per effettuare simulazioni di modelli basati su agenti. Nella sezione 4 verrà mostrato un caso d'uso e i risultati ottenuti. Nella sezione 5 si faranno le conclusioni riguardo questo lavoro e progetti futuri.

2 Caratteristiche di Rust

Rust implementa diversi paradigmi - imperativo, funzionale, object oriented - accogliendo dentro di se numerose funzionalità che ne rendono l'utilizzo versatile ed efficiente.
Questo linguaggio si vanta di essere Memory Safe, ovvero non è possibile avere problemi di data race o dangling pointers, anche in ambiente multithread. Questo vuol dire che Rust si assicura a tempo di compilazione che non vi siano accessi a porzioni di memoria non allocata o non inizializzata, e che la memoria utilizzata venga liberata appena sia possibile, nonostante non esista un garbage collector.
Non tutti i controlli riescono ad essere gestiti a tempo di compilazione, come la verifica di un accesso corretto all'interno di un array o l'apertura di un file, ma il runtime resta incredibilmente basso.
La capacità del linguaggio di essere Memory Safe deriva da una combinazione di regole che devono essere rispettate da un programma:

1. Inizializzazione forzata
2. Ownership
3. Restricted aliasing

1. Qualsiasi variabile all'interno del programma deve essere inizializzata prima di poter essere usata. Rust non concede la possibilità di accedere ad una tale porzione di memoria;
2. Ogni valore ha una variabile che è chiamata proprietario. Ci può essere un solo proprietario alla volta, ed ogni volta che il proprietario va fuori scope, Rust si occupa di liberare la memoria occupata. Questo approccio consente di avere delle limitazioni sull'accesso in scrittura di una cella di memoria;
3. I riferimenti consentono di puntare al valore di una variabile senza prendere la proprietà di quest'ultima. Il restricted aliasing ci consente di avere infiniti riferimenti immutabili ad una variabile o un solo riferimento mutabile ad una variabile. Questo perché è lecito accedere ad una variabile in sola lettura, mentre se si vuole sovrascrivere il contenuto di una cella di memoria, l'accesso deve essere unico. In Rust non esistono meccanismi che gestiscono la sincronizzazione di accesso ai dati. Ogni riferimento in Rust ha un lifetime, ovvero lo scope nel quale il riferimento è valido. Nella maggior parte dei casi -e con il rilascio di nuove versioni- il lifetime viene inferito, ma è ancora necessario annotarlo in casi come la dichiarazione di funzioni. Il principale obiettivo di questo strumento è

evitare che ci siano *dangling pointers*, ovvero riferimenti a punti di memoria non più accessibili.

Rust introduce anche delle caratteristiche del paradigma object-oriented:

1. Oggetti
2. Polimorfismo
3. Incapsulamento

1. In Rust è possibile dichiarare oggetti mediante Enums, Structs e blocchi di implementazione. Gli Enums definiscono un tipo enumerando tutti i possibili valori che può assumere. Una Struct è un custom data type che aggrega dei dati che hanno una relazione tra di loro. Questi tipi possono avere fra i propri valori anche dei generics. I blocchi imp definiscono il comportamento che questi oggetti devono avere;
2. In Rust il polimorfismo è implementato mediante i Traits. Un trait consente a differenti oggetti di condividere degli specifici comportamenti, lasciando a l'oggetto l'onere di implementare il metodo;
3. In Rust ogni dettaglio implementato è privato. E' compito del programmatore decidere quale tipo, metodo o modulo deve essere pubblico, utilizzando la keyword pub.

Nel linguaggio non è prevista l'ereditarietà, ovvero un oggetto non può ereditare metodi o campi dal genitore. Questa mancanza nel linguaggio genera non pochi problemi quando si tratta di generalizzare il codice ma rende la struttura più robusta.

Alle regole sulla sicurezza della memoria è possibile trovare una scappatoia usando quello che viene definito Unsafe Rust. La filosofia dietro questa modalità è che interagire con hardware a basso livello non è molto sicuro, quindi è sconsigliato e praticamente impossibile con le regole fissate dal linguaggio. Ma lavorare con sistemi a basso livello, o programmare un proprio sistema operativo necessita l'accesso a queste funzionalità, quindi Rust dà la possibilità di effettuare 4 azioni, chiamate unsafe superpowers:

1. Dereferenziare un raw pointer
2. Chiamare una funzione o un metodo unsafe
3. Accedere o modificare una variabile statica mutabile
4. Implementare un trait unsafe

Usare dei blocchi unsafe non consente di evitare le stringenti regole di ownership e sicurezza di Rust, ma lasciano più libertà al programmatore che però deve assicurare che alla fine del blocco tutte le regole siano state rispettate.

1. E' consentito ignorare le regole di borrowing con i raw pointers, potendo avere nello stesso momento più riferimenti mutabili e immutabili nello stesso momento. Non è assicurato che i raw pointers puntino a indirizzi validi di memoria o che abbiano valori non nulli. Non è implementato nessun sistema di pulizia automatico;
2. Chiamare o definire una funzione unsafe consente di inserire nei programmi anche chiamate ad altri linguaggi, lasciando al programmatore il compito di assicurarsi che il codice rispecchi tutte le regole di Rust;
3. Gestire una variabile globale in Rust può essere complesso, in quanto due threads potrebbe cercare di accedere alla variabile nello stesso momento, quindi c'è bisogno di usare un blocco unsafe quando non si può ricorrere alla tecniche di concorrenza fornite dal linguaggio;
4. Un trait è unsafe quando uno dei suoi metodi non è verificabile dal compilatore.

Nonostante Rust sia un linguaggio giovane, è oggetto di studio da parte di molti ricercatori. Nella tabella 1, presa da uno studio riguardante l'utilizzo di Rust nell'ambito dell'Internet of Things[5], si evidenziano numerosi concetti relativi all'architettura di svariati linguaggi di programmazione. Le celle in verde indicano la migliore scelta possibile. Le gialle indicano che un altro linguaggio implementa una soluzione migliore. Le blu indicano che è possibile ottenere quella specifica proprietà, ma bisogna apportare delle modifiche al compilatore, estendere il linguaggio o avere estrema competenza. Le rosse indicano una mancanza del linguaggio rispetto alla proprietà.
Le features evidenziate in tabella mostrano le capacità di un linguaggio rispetto a sicurezza, thread safety, consumo energetico, astrazioni, ecc ecc.
La riga riguardante Rust è molto incoraggiante. La capacità di controllare gli errori a tempo di compilazione influisce molto positivamente, limitando le chiamate a runtime e rendendolo veloce quanto C e C++, nonostante una semantica più ad alto livello.

[5] Tunc¸ Uzlu, Ediz Saykol On Utilizing Rust Programming Language for Internet of Things

	MC	IL	FSL	IO	GN	SBT	SMM	PM	TS	PA	NOS	ZCA	RMM
Rust	yes	LLVM	partial	fast FFI	yes	yes	yes	yes	yes	cross compile	Core	yes	Unsafe
C	yes	LLVM	no	yes	no	no	no	no	no	yes	partial	yes	yes
C++	yes	LLVM	no	yes	yes	no	no	no	no	yes	partial	yes	yes
Haskell	yes	type safe	pure	expensive	yes	partial	garbage collector	yes	yes	cross compile	complex	no	complex
Lisp	yes	C	partial	expensive	no	no	garbage collector	varies	no	special builds	complex	no	complex
Python	no	C	no	CFFI	no	no	garbage collector	yes	no	CPython	no	no	no
Go	yes	GCCGO	no	CGO	no	no	garbage collector	3rd party	no	cross compile	no	no	no
Swift	yes	LLVM	no	extension	yes	no	ARC	3rd party	yes	no	no	no	no

Tabella 1 Rust compared to other programming languages. MC=compiled to machine architecture, IL=intermediate language, FSL=functional standard library, IO=interoperable with de-facto systems languages, GN=generics, SBT=safety between threads, SMM=safe memory model, PM=package manager, TS=thread safety, PA=platform agnosticity, NOS=operable in no-OS environment, ZCA=zero cost abstractions, RMM=raw memory management (e.g. volatile or memory fence).

MC(compiled to machine architecture) e ZCA(zero cost abstraction) si riferiscono alle performance. Essere capaci di compilare direttamente sull'hardware desiderato è la chiave per esecuzioni ad alta velocità. Per esempio Python deve essere compilato in C prima di poter produrre un eseguibile.
La possibilità di usare generics e richiami di programmazione a oggetti favorisce la possibilità di progettare software riutilizzabile.
Il SMM(Safety Memory Model*)* e il *TS(*Type System*)* di Rust consentono al linguaggio di essere l'unico linguaggio ad avere un ottima stabilità e prevenire errori di memorizzazione dei dati. Il sistema di ownership e i meccanismi di gestione dinamica dei riferimenti rendono Rust sicuro anche in ambiente multithreading.
IO (interoperable with de-facto systems languages), PA (platform agnosticity), NOS (operable in no-OS environment) e RMM (raw memory management) si riferiscono all'accessibilità. Rust consente di richiamare funzioni esterne con l'utilizzo di blocchi unsafe, rendendo disponibile praticamente ogni risorsa del C.

3 Rust-AB

Rust-AB è una libreria pensata per essere ready-to-use. L'architettura scelta è difatti ispirata dal design di MASON[6] (MultiAgent Simulation Toolkit), libreria pensata per effettuare la simulazione di eventi discreti multi-agenti. La libreria MASON definisce un architettura standard e modulare basandosi su quelli che sono i principi del Model-View-Controller design pattern.
La simulazione effettuata con MASON è composta principalmente da 3 fattori:

1. gli agenti di simulazione(definiti da un'interfaccia Steppable);
2. uno scheduler per la simulazione(definito dall'oggetto Scheduler);
3. lo stato della simulazione(definito dall'oggetto SimState).

Per semplificare l'utilizzo e la curva di apprendimento, l'implementazione di questa libreria ha rispecchiato quella di MASON, nonostante sono stati effettuati dei cambiamenti per poter sfruttare a pieno tutte le potenzialità di Rust.
La versione 1.0 di Rust-AB è completamente funzionante ed è rilasciata sotto licenza MIT, su una repository GitHub pubblica[7].
La libreria è divisa in questi 6 file:

1. Agent
2. AgentImpl
3. Field2D
4. Location
5. Priority
6. Schedule

Agent racchiude il concetto più importante della libreria. Agent è un trait, e secondo le specifiche di Rust, questo permette alle strutture di condividere un determinato comportamento, ricordando le peculiarità dei linguaggi orientati agli oggetti e il design di MASON. Ogni struttura ha la possibilità di diventare un agente, a patto che venga implementato il metodo contenuto nel trait, ovvero *step,* che dovrà racchiudere la logica dell'agente preso in esame.

```rust
pub trait Agent {
    fn step(&mut self);
}
```

Img 1 Trait Agent

[6] https://cs.gmu.edu/~eclab/projects/mason/

[7] https://github.com/ spagnuolocarmine/abm

AgentImpl è una struttura che agisce come un wrapper sull'agente, fornendogli un id univoco e un boolean per poter scegliere fra le due opzioni possibili di schedulazione.

```rust
#[derive(Clone, Debug)]
pub struct AgentImpl<A: Agent + Clone> {
    pub id: u32,
    pub agent: A,
    pub repeating: bool,
}
```

Img 2 AgentImpl Struct

Priority definisce una struttura per gestire la priorità di un agente. Quando vengono effettuati i passi di simulazione, lo scheduler deve conoscere qual è l'ordine con cui gli agenti devono essere estratti dalla coda a priorità. Questa struttura fornisce delle operazioni di confronto per poter stabilire l'ordinamento.

```rust
#[derive(Clone, Debug)]
pub struct Priority{
    pub time:f64,
    pub ordering:i64
}
```

Img 3 Priority Struct

```rust
impl Ord for Priority {
    fn cmp(&self, other: &Priority) -> Ordering {
        if self.time < other.time {return Ordering::Greater;}
        if self.time > other.time {return Ordering::Less;}
        if self.ordering < other.ordering {return Ordering::Greater;}
        if self.ordering > other.ordering {return Ordering::Less;}
        Ordering::Equal
    }
}
```

Img 4 Priority Methods

Lo scheduler è invece il cuore della libreria. L'oggetto Schedule fornisce tutte le funzionalità per gestire la simulazione di eventi discreti. Ha la medesima interfaccia dello scheduler di MASON. La simulazione avviene schedulando time-by-time gli agenti. Un agente è schedulabile se è una struttura che implementa il trait Clone, che permette allo scheduler di clonare gli agenti per rispettare gli standard di sicurezza della memoria di Rust. Lo scheduler funziona come una coda a priorità -First in, first out- dove gli agenti sono ordinati secondo il tempo di schedulazione e il valore di priorità.

```
pub struct Schedule<A:'static + Agent + Clone + Send>{
    pub step: usize,
    pub time: f64,
    pub events: PriorityQueue<AgentImpl<A>,Priority>,
}
```

Img 5 Schedule

Il tempo di simulazione, rappresentato da un numero reale, inizia con la schedulazione del primo agente. Ad ogni passo di simulazione, tutti gli agenti schedulati effettuano la funzione step, in accordo con la loro priorità di schedulazione.
Nella versione Alpha 1.0, Rust-AB fornisce due opzioni per lo scheduling:

1. schedule_once
2. schedule_repeating

```
pub fn schedule_once(&mut self, agent: AgentImpl<A>,the_time:f64, the_ordering:i64) {
    self.events.push(agent, Priority{time: the_time, ordering: the_ordering});
}

pub fn schedule_repeating(&mut self, agent: A, the_time:f64, the_ordering:i64) {
    let mut a = AgentImpl::new(agent);
    a.repeating = true;
    let pr = Priority::new(the_time, the_ordering);
    self.events.push(a, pr);
}
```

Img 6 Scheduling options

1. schedule_once, inserisce un agente nello scheduler ad uno specifico passo di simulazione. Il tempo di schedulazione è passato come parametro. Un secondo parametro, la priorità, è usato per ottenere l'ordinamento tra tutti gli agenti schedulati allo stesso tempo di simulazione.
2. schedule_repeating, agisce come il primo metodo, ma l'agente verrà schedulato anche per tutti i passi di simulazione successivi.

Location2D è un trait, che permette ad una struttura di localizzarsi all'interno di un campo 2D. Ogni struttura che implementa questo trait può essere inserita in un campo 2D e permette al programmatore di modellare facilmente tutte le possibili interazioni all'interno della simulazione. Devono essere implementate le due funzioni, get_location e set_location, che permettono di inserire o recuperare le posizioni all'interno del campo.

```
pub trait Location2D {
    fn get_location(self) -> Real2D;
    fn set_location(&mut self, loc: Real2D);
}
```

Img 7 Location2D

Field2D è una struttura a matrice spara che permette al programmatore di modellare le interazioni fra agenti in un campo 2D. Questa struttura è parametrizzata sul trait Location2D, che permette alle strutture che lo implementano di definire dei metodi per inizializzare o recuperare la posizione nel campo. Il Field2D prevede inoltre che vengano implementati i traits Clone, Hash, ed Eq, senza i quali alcune operazione non possono essere eseguite. Questo permette al programmatore di poter facilmente usufruire di questo campo per qualsiasi implementazione, in quanto una volta rispettati i vincoli imposti dall'architettura, la versatilità offerta è molto ampia.

```rust
#[derive(Clone)]
pub struct Field2D<A: Location2D + Clone + Hash + Eq + Display + Copy> {
    pub findex: HashMap<A, Int2D>,
    pub fbag: HashMap<Int2D, Vec<A>>,
    pub fpos: HashMap<A, Real2D>,
    pub width: f64,
    pub heigth: f64,
    pub discretization: f64,
    pub toroidal: bool,
}
```

Img8 Field2D

Il field contiene tre hashmap, ognuna con uno scopo preciso. *findex* memorizza gli agenti con la loro posizione discretizzata, *fpos* memorizza gli agenti con la loro posizione reale, mentre *fbag* memorizza in un vettore tutti agenti presenti ad una determinata posizione discretizzata. Il campo viene diviso in queste bag per poter effettuare la ricerca di elementi vicini in maniera più veloce e precisa.

```rust
pub fn set_object_location(&mut self, object: A, pos: Real2D) {…
}
pub fn get_neighbors_within_distance(&self, pos: Real2D, dist: f64) -> Vec<A> {…

}
pub fn get_objects_at_location(&self, pos: Real2D) -> Vec<&A>{…
}
pub fn num_objects(&self) -> usize {…
}
pub fn num_objects_at_location(&self, pos: Real2D) -> usize {…
}
pub fn get_object_location(&self, obj: A) -> Option<&Real2D> {…
}
```

Img 7 Field2D

La struttura offre diverse funzionalità:

1. set_object_location
2. get_neighbours_within_distance
3. get_object_location
4. get_objects_at_location
5. num_objects
6. num_objects_at_location

1. La funzione inserisce o aggiorna un oggetto nel campo passando come input una determinata posizione;
2. La funzione restituisce un vettore di agenti presenti nel cerchio avente come centro una posizione fissata e come raggio la distanza passata come parametro. Questa ricerca è effettuata per ottenere tutto il vicinato rispetto ad un agente;
3. La funzione restituisce la posizione di un determinato oggetto nel campo;
4. La funzione restituisce un vettore contenente tutti agenti presenti ad una determinata posizione;
5. La funzione restituisce il numero di agenti totali presenti nel campo;
6. La funzione restituisce il numero di agenti presenti ad una determinata posizione passata come parametro.

La Simulation State è lo stato della simulazione Rust-AB. Essa è composta dalla definizione di un agente, un'istanza dello scheduler Rust-AB, ed un insieme di campi e variabili. La logica della simulazione è sviluppata all'interno della funzione step implementata dagli agenti, e per questo motivo l'ambiente di programmazione deve fornire un meccanismo per accedere allo stato di simulazione dalla funzione.
Lo stato di simulazione è definito usando una struttura che contiene tutti i campi e la variabili che dovranno essere usati nella simulazione. Viene dichiarato quindi un riferimento statico che verrà inizializzato a run-time. E' necessario accedere a questa struttura usando dei lock che garantiscono l'accesso sicuro alla memoria.

Le limitazioni di questa libreria sono principalmente dovute al modello a oggetti fornito da Rust e dai vincoli che derivano dal suo Safe Memory Model. La prima limitazione riguarda l'impossibilità di definire multipli agenti, sebbene sia possibile implementare un modello multiagente definendo differimenti comportamenti all'interno della definizione di agente. La seconda limitazione consiste nel fatto che un field può contenere solamente oggetti dello stesso tipo, il che significa che per modellare interazioni tra oggetti di diverso tipo è necessario usare molteplici campi.

4 Simulazione di Boids

Per analizzare l'efficacia di questa libreria, abbiamo preso in esame il noto modello di simulazione di Boids, ed abbiamo eseguito numerosi benchmark variandone i parametri. Abbiamo quindi comparato i risultati con quelli ottenuti dal toolkit MASON e da NetLogo eseguendo lo stesso modello.
Il modello di Boids[8] sviluppato da C. Raynolds descrive il comportamento di uno stormo di uccelli mediante la simulazione di agenti indipendenti.
Il comportamento degli agenti è una combinazione di tre regole indipendenti:

1. Distacco
2. Allineamento
3. Coesione

1. Il distacco modella i movimenti che un agente deve tenere per evitare la collisione con altri agenti;
2. L'allineamento considera i movimenti degli altri agenti intorno in modo da ottenere una rotta simile ai vicini;
3. La coesione cerca di correggere la rotta dell'agente in modo da orientarsi sempre verso la posizione media dei vicini.

Il modello di Boids implementato per Rust-AB segue la stessa strategia utilizzata da MASON per la simulazione di Flockers.
Abbiamo quindi definito il nostro agente, che non è altro che una struttura contenente tutti i dati locali dell'agente, come Bird*,* con lo scopo di emulare il concetto di un uccello in uno stormo. Bird, come si può osservare nell'immagine 1.1, tramite la macro di Rust *#derive*, che segnala al compilatore di computare automaticamente determinate funzioni, implementa Clone e Copy, e implementa il trait Location2D per poter essere inserito nel Field2D. Un agente deve inoltre implementare i trait Agent, Eq e Hash per come è stata definita l'architettura della libreria.

[8] Reynolds, C.W.: Flocks, Herds, and Schools: A Distributed Behavioral Model. Computer Graphics (ACM) (1987)

```rust
#[derive(Clone, Copy)]
pub struct Bird{
    pub id: u128,
    pub pos: Real2D,
    pub last_d: Real2D,
}
```

1.1 Rust-AB Agent struct

```rust
impl Agent for Bird {
    fn step(&mut self) {

        let vec = GLOBAL_STATE.lock().unwrap().field1.get_neighbors_within_distance(self.pos, 10.0);

        let avoid = self.avoidance(&vec);
        let cohe = self.cohesion(&vec);
        let rand = self.randomness();
        let cons = self.consistency(&vec);
        let mom = self.last_d;

        let mut dx = COHESION*cohe.x + AVOIDANCE*avoid.x + CONSISTENCY*cons.x + RANDOMNESS*rand.x + MOMENTUM*mom.x;
        let mut dy = COHESION*cohe.y + AVOIDANCE*avoid.y + CONSISTENCY*cons.y + RANDOMNESS*rand.y + MOMENTUM*mom.y;

        let dis = (dx*dx + dy*dy).sqrt();
        if dis > 0.0 {
            dx = dx/dis*JUMP;
            dy = dy/dis*JUMP;

        }

        let _lastd = Real2D {x: dx, y: dy};
        let loc_x = toroidal_transform(self.pos.x + dx, WIDTH);
        let loc_y = toroidal_transform(self.pos.y + dy, HEIGTH);
        self.last_d = _lastd;
        self.pos = Real2D{x: loc_x + dx, y: loc_y + dy};

        GLOBAL_STATE.lock().unwrap().field1.set_object_location(*self, self.pos);

    }
}
```

1.2 Rust-Ab Agent step

Nell'immagine 1.2 si può osservare l'implementazione della logica presente all'interno del metodo step legato alla struttura Bird. Dopo aver effettuato l'accesso alla struttura che mantiene lo stato della simulazione ed aver individuato i vicini in una certa area, l'agente richiama le funzioni per modellare il proprio comportamento e aggiornare la propria posizione nel campo, utilizzando sempre un lock per poter accedere al riferimento statico dello State..

```rust
pub struct State{
    pub field1: Field2D<Bird>,
}

impl State {
    pub fn new(w: f64, h: f64, d: f64, t: bool) -> State {
        State {
            field1: Field2D::new(w, h, d, t),
        }
    }
}
```

1.3 Rust-AB State

Nell'immagine 1.3 si può osservare l'implementazione dello stato della simulazione. All'interno della struttura State possono essere inseriti i campi e le variabili che devono essere prese in oggetto nella simulazione.
Nel caso di Boids, abbiamo definito l'ambiente di simulazione con solamente un Field2D.
Uno dei problemi di Rust, come già accennato in precedenza, risiede nel fatto che

```rust
lazy_static! {
    static ref GLOBAL_STATE: Mutex<State> = Mutex::new(State::new(WIDTH, HEIGTH, DISCRETIZATION, TOROIDAL));
}
```

1.4 Rust-AB State global

non è possibile condividere dati attraverso multiple funzioni nello stesso momento, quindi è stato necessario dichiarare l'istanza di State come una variabile globale, come è possibile notare dall'immagine 1.4.

L'immagine 1.5 mostra invece la funzione di simulazione vera e propria. Viene inizializzato un oggetto Schedule e un generatore di numeri casuali. Si definiscono in maniera casuale un numero parametrizzato di agenti ed ognuno di essi viene inserito nel Field2D e schedulati con il metodo schedule_repeating. Infine viene lanciato lo scheduling per un numero fissato di step.

```rust
fn main() {
    let mut rng = rand::thread_rng();
    let mut schedule: Schedule<Bird> = Schedule::new();

    for bird_id in 0..NUM_AGENT{
        let r1: f64 = rng.gen();
        let r2: f64 = rng.gen();
        let last_d = Real2D {x: 0.0, y: 0.0};
        let bird = Bird::new(bird_id, Real2D{x: WIDTH*r1, y: HEIGTH*r2}, last_d);
        GLOBAL_STATE.lock().unwrap().field1.set_object_location(bird, bird.pos);
        schedule.schedule_repeating(bird, 0.0, 0);
    }
    for _ in 1..STEP{
        schedule.step();
    }
}
```

1.5 Rust-AB Main Simulation function

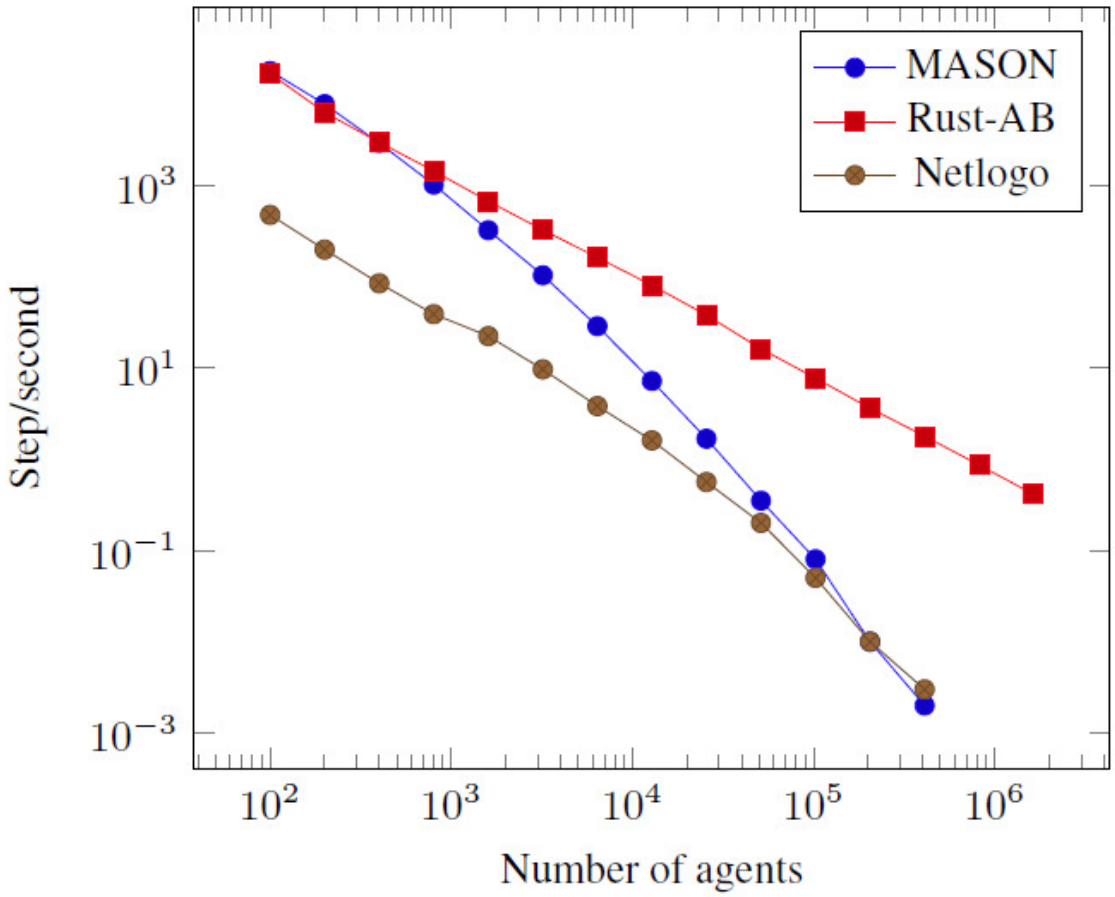
Abbiamo quindi analizzato questo modello con molteplici test per misurare le sue effettive capacità in confronto con i risultati ottenuti da MASON e NetLogo sulla simulazione di Flockers. Tutti gli esperimenti sono stati effettuati su una macchina avente queste caratteristiche:

1. 1 x CPU i7-8700T12 x 2.40 MHz;
2. 16 GB RAM;
3. Ubuntu Linux 18.04 LTS;
4. Oracle Java Virtual Machine 1.7;
5. Rust 1.31;

Abbiamo quindi definito diverse configurazioni della simulazione, cambiando l'ambiente di simulazione ed il numero di agenti:

1. Agenti a densità costante;
2. Campo a dimensioni fisse;
3. Numero di agenti costante.

La densità degli agenti in un campo di simulazione può essere facilmente computata come $\frac{wxh}{A}$, dove w e h rappresentano rispettivamente la larghezza e l'altezza del campo e A denota il numero di agenti.

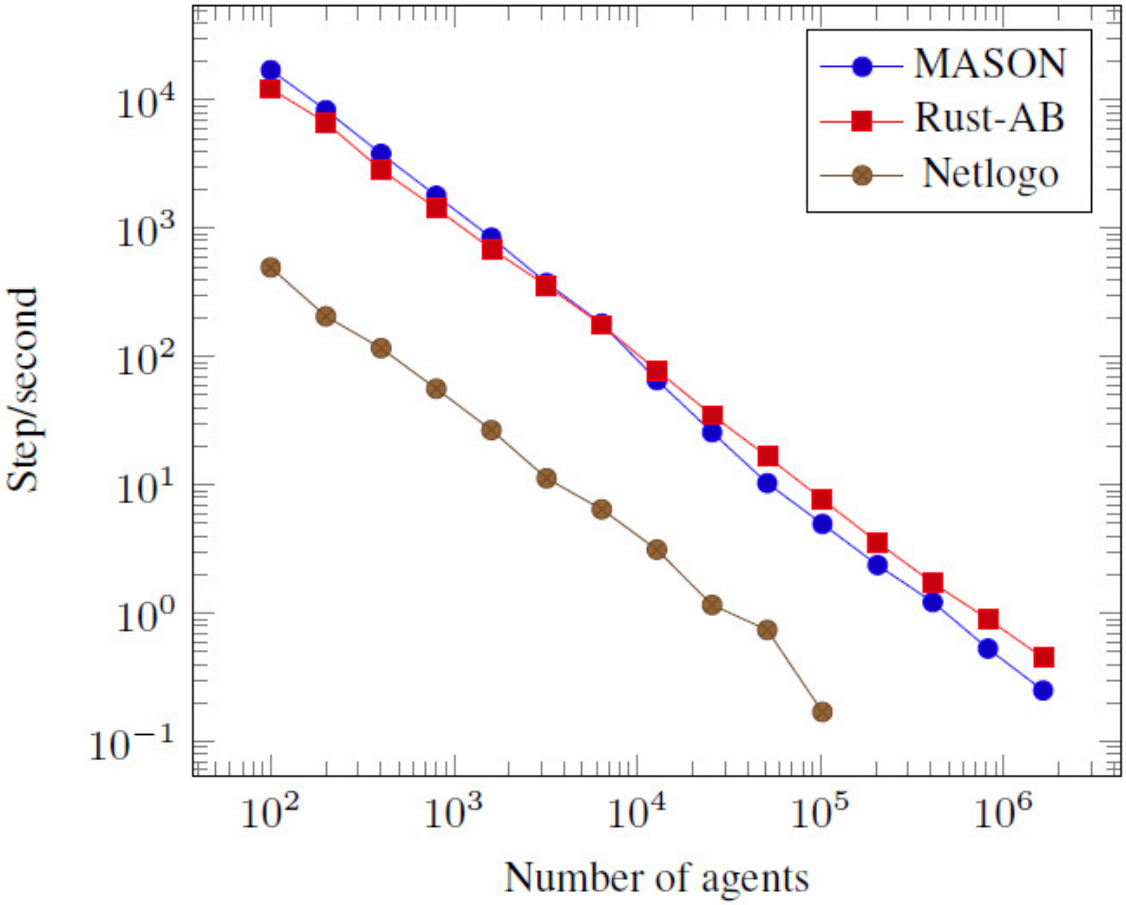


2.2 Rust-AB performances

1. Il primo esperimento mira a misura l'efficienza di Rust-AB quando variano siano il numero di agenti che le grandezze del campo, ma mantenendo le stesse proporzioni. I risultati sono mostrati nell'immagine 2.1. L'asse delle ascisse mostra il numero di agenti, misurato in un range di valori da 100 a 1638400, mentre l'asse delle ordinate mostra le performance ottenute in termini di step al secondo. Come mostrato in figura, quando la densità degli agenti rimane costante, le performance di Rust-AB sono simili a quelle di MASON, le quali sono molto superiori a quelle ottenute da NetLogo.
2. Il secondo esperimento valuta le performance di Rust-AB quando il il numero di agenti incrementa su un campo di dimensioni costanti(per questa simulazione un campo 200x200). In questo caso, incrementando il numero di agenti in un campo così piccolo, aumenta il carico computazionale nel trovare i vicini e la nuova posizione degli agenti. La figura 2.2 mostra i risultati ottenuti. Rust-AB con un basso numero di agenti ottiene le stesse performance di MASON, ma con il crescere del numero di agenti, e quindi consequenzialmente con il carico computazionale che aumenta, riesce a scalare molto meglio dei due competitor. E'

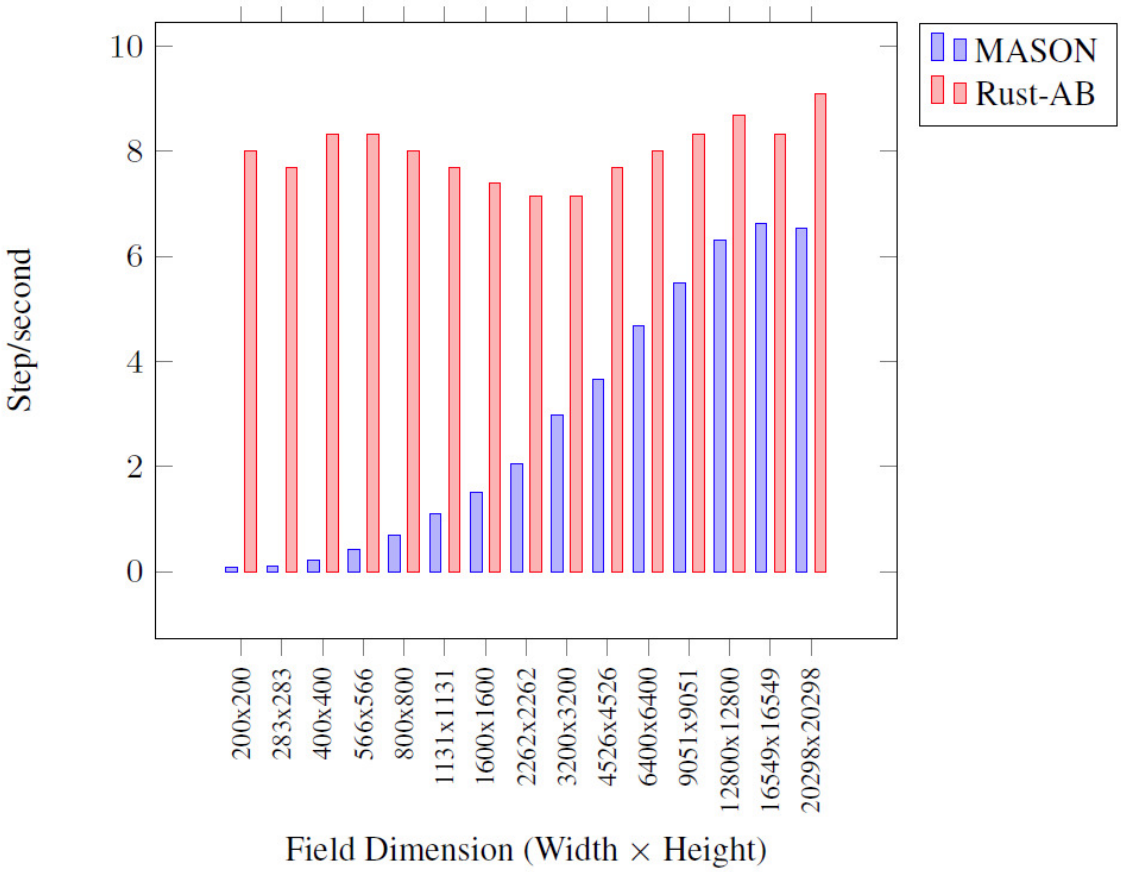
nota da tenere in mente il fatto che sia MASON che NetLogo non sono riusciti ad eseguire le ultime due configurazioni di simulazione.

3. Il terzo esperimento ha valutato le performance di Rust-AB in relazione alla capacità di simulare un numero costante di agenti (102400) variando la



2.1 Rust-AB performance

dimensione del campo. La figura 2.3 mostra i risultati ottenuti. Sull'asse delle ascisse sono mostrate le dimensioni del campo, che variano nel range da 200x200 a 20298x20298, mentre sull'asse delle ordinate sono mostrate le performance ottenute in termini di step al secondo. Come si può notare, Rust-AB performa molto meglio di MASON fino a che la densità del campo non diventa abbastanza piccola da diminuire il carico di lavoro. Quando la densità del campo di simulazione è piccola, le performance delle librerie sono molto simili. Non sono mostrate le performance di NetLogo in quanto la libreria non è riuscita ad eseguire le configurazioni della simulazione.



2.3 Rust-AB performance

5 Conclusioni

Questa tesi ha illustrato tutte le potenzialità di Rust e un suo possibile utilizzo nel campo dei modelli di simulazione basati su agenti. Rust-AB, nella sua versione Alpha 1.0, è riuscita ad ottenere delle ottime prestazioni in confronto ad alcune delle librerie maggiormente utilizzate nel campo della simulazione. In futuro, lo sviluppo di questa libreria cercherà di sfruttare a pieno quelle che sono le potenzialità del linguaggio, cercando di ottenere il massimo delle prestazioni da un'architettura così solida e incentrata sulla sicurezza e consistenza dei dati. Altri possibili miglioramenti riguarderanno il motore della libreria, ovvero si cercherà di migliorare la computazione concorrente, e altri possibili casi d'utilizzo che non utilizzino solamente interazioni locali.

References

1. https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted
2. https://prev.rust-lang.org/en-US/friends.html
3. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html
4. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-java.html
5. Tunc¸ Uzlu, Ediz Saykol On Utilizing Rust Programming Language for Internet of Things
6. https://cs.gmu.edu/~eclab/projects/mason/
7. https://github.com/spagnuolocarmine/abm
8. Reynolds, C.W.: Flocks, Herds, and Schools: A Distributed Behavioral Model.

Indice Sommario

1 Introduzione 2
2 Caratteristiche di Rust 6
3 Rust-AB 10
4 Simulazione di Boids 16
5 Conclusioni 23
References 25
Indice Sommario 26