

UNIVERSITÀ DEGLI STUDI DI
CASSINO E DEL LAZIO MERIDIONALE



DIPARTIMENTO DI INGEGNERIA ELETTRICA
E DELL'INFORMAZIONE "MAURIZIO SCARANO"

CORSO DI LAUREA IN INGEGNERIA INFORMATICA
E DELLE TELECOMUNICAZIONI

**Assessment delle performance di Elixir
nell'ambito IOT.**

Relatore:

Prof. Ciro D'Elia

Candidato:

Nico Fiorini

ANNO ACCADEMICO 2022/2023

Ai miei nonni,
che hanno contribuito
a plasmare la persona che sono oggi.

There's a lot of beauty in the ordinary things.
The Office - Pam Beesly

Abstract

L'industria del software è attualmente alle prese con la crescente esigenza di sviluppare applicazioni sempre più scalabili e performanti per gestire efficacemente l'aumento del numero di utenti e dei servizi che utilizzano tali applicazioni. In particolare, nel contesto dell'Internet of Things (IoT), un settore in costante sviluppo, la reattività, la tolleranza agli errori e la scalabilità emergono come fattori cruciali.

In questo contesto, Elixir, un linguaggio di programmazione funzionale e concorrente basato su Erlang, emerge come una scelta promettente per la costruzione di sistemi altamente affidabili e reattivi, nonché la sua capacità di gestire carichi di lavoro intensivi e flussi di dati in tempo reale tipici delle applicazioni IoT.

Questo studio si propone di analizzare le caratteristiche di Elixir che lo rendono un linguaggio degno di nota, e attraverso una serie di esperimenti empirici si misurano le performance di Elixir, anche mettendolo a confronto con altre soluzioni più diffuse.

I risultati di questa ricerca forniscono una base per valutare l'efficacia di Elixir, contribuendo così alla comprensione del ruolo che questo linguaggio può svolgere nel futuro dello sviluppo delle applicazioni IoT.

Indice

1	Introduzione	1
1.1	IoT - L'internet delle cose	2
1.1.1	Architettura IoT	2
1.1.2	Introduzione allo studio svolto	4
2	Caratteristiche di Elixir	6
2.1	Introduzione	6
2.2	Software utilizzati	7
2.2.1	Installazione software utilizzati	7
2.3	Il paradigma funzionale	9
2.3.1	Struttura di un progetto Elixir	10
2.3.2	Basi dichiarative	11
2.3.3	Transizione al funzionale	12
2.3.4	Conseguenze sulle prestazioni dell'immutabilità	13
2.4	Concorrenza	14
2.4.1	La concorrenza in Beam	14
2.4.2	Concorrenza basata su attori	16
2.5	OTP - Open Telecom Platform	18
2.5.1	GenServer	18
2.5.2	Supervisor	20
2.6	Sistemi distribuiti	22
2.6.1	Un esempio di comunicazione tra nodi	22
2.6.2	Considerazioni e sfide da considerare	23
2.7	Integrazione con codice esterno	24
2.7.1	Interoperabilità tramite NIF	25
2.7.2	Interoperabilità con Port	26
3	Performance - Test sperimentali	29
3.1	Introduzione	29
3.1.1	Hardware utilizzato	29
3.2	Test concorrenziale	30

3.2.1	Implementazione del Test	30
3.2.2	Esecuzione Test	32
3.2.3	Analisi Matlab	33
3.3	Test concorrentiale con IO	38
3.3.1	Implementazione del test	38
3.3.2	Esecuzione Test	41
3.3.3	Analisi Matlab	41
3.4	Test interoperabilità con C/C++	48
3.4.1	Definizione problema da risolvere	48
3.4.2	Funzione ricorsiva	49
3.4.3	Funzione ricorsiva ottimizzata	49
3.4.4	Funzione in C++ con Port	50
3.4.5	Funzione implementata in C++ tramite NIF	52
3.4.6	Esecuzione Test	54
3.5	Test Http Server	56
3.5.1	Implementazione dei Server	56
3.5.2	Esecuzione Test Http	58
Conclusioni e sviluppi futuri		61
Bibliografia e sitografia		62

Capitolo 1

Introduzione

Elixir è un linguaggio di programmazione dinamico e funzionale sviluppato nel 2012 da José Valim, con l'obiettivo di favorire uno sviluppo più agevole sulla Erlang Virtual Machine(VM) e affrontare la sfida legata a sistemi distribuiti, scalabili e affidabili mantenendo al contempo la compatibilità con l'ecosistema di Erlang. Elixir si è affermato come una promettente scelta nell'industria del software, specialmente in contesti dove è richiesta scalabilità, tolleranza agli errori e reattività grazie al suo approccio concorrenziale.

In particolare, Elixir può risultare vantaggioso nel campo dell'IoT per diversi motivi:

1. **Concorrenza:** Nell'ambito dell'IoT, la gestione simultanea di dispositivi è essenziale. Elixir, grazie alla concorrenza basata su processi leggeri, e tramite il modello basato su attori può consentire il monitoraggio e il controllo efficiente di numerosi dispositivi contemporaneamente.
2. **Fault Tolerance:** Data la natura degli ambienti IoT, dove i dispositivi possono guastarsi improvvisamente, Elixir offre strumenti per la supervisione e la gestione degli errori, garantendo la continuità delle operazioni anche in caso di fallimenti.
3. **Sviluppo Rapido e Manutenzione:** Elixir è un linguaggio moderno che offre una sintassi efficiente e snella, oltre a strumenti di sviluppo come Mix per la gestione delle dipendenze e l'ambiente interattivo iex. La presenza di un package manager (Hex)[10] e la possibilità di generare automaticamente la documentazione facilitano il processo di sviluppo e manutenzione del codice.

1.1 IoT - L'internet delle cose

L'Internet of Things (IoT), traducibile in italiano come Internet delle cose, si riferisce a un sistema di dispositivi interconnessi attraverso una rete, capaci di raccogliere e scambiare dati con sistemi esterni e di interagire con l'ambiente circostante.

Questi dispositivi possono essere dei sensori, telecamere, veicoli, elettrodomestici, o qualsiasi altro oggetto in grado di comunicare attraverso una rete. L'obiettivo principale dell'IoT è quello di automatizzare processi, migliorare l'efficienza, fornire informazioni in tempo reale e creare nuove opportunità di servizi e applicazioni.

Un sistema IoT ha una vasta gamma di applicazioni ed il suo uso sta crescendo velocemente nella società moderna, basta pensare alle molteplici applicazioni che spaziano dall'uso industriale per l'automazione, all'uso individuale con la domotica.

1.1.1 Architettura IoT

Un'architettura base per l'IoT può essere composta da quattro livelli come in figura 1.1

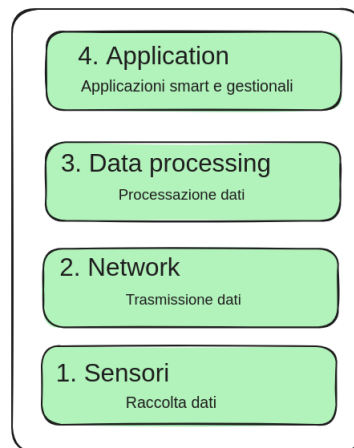


Figura 1.1: Architettura a 4 livelli

1. Il livello Sensori è responsabile di collezionare dati da diverse sorgenti, i dispositivi fisici possono essere potenzialmente migliaia. Tutti questi dispositivi hanno il compito di inviare o ricevere dati ai livelli superiori.

2. Il livello Network di un architettura IoT permette ai dispositivi di comunicare tra di loro o con il livello tre. Esempi di tecnologie di rete possono essere la rete WiFi, Zigbee e reti cellulari come il 5G.
3. Il livello Data Processing gestisce la raccolta, l'analisi e l'interpretazione dei dati provenienti dai dispositivi IoT. Include tecnologie come sistemi di gestione dei dati, piattaforme di analisi e algoritmi di machine learning.
4. Il livello delle applicazioni nell'architettura IoT è quello che interagisce direttamente con l'utente finale, fornendo interfacce user-friendly per controllare i dispositivi IoT. Include app mobili, portali web e altre interfacce utente, oltre a servizi middleware per la comunicazione tra dispositivi. Inoltre, offre funzionalità di analisi e elaborazione dati, come algoritmi di machine learning e strumenti di visualizzazione.

Nerves

Un framework open-source in Elixir è Nerves [15], consente di sviluppare software embedded basato su Erlang/OTP.

Offre agli sviluppatori un ambiente di sviluppo coerente e potente per la creazione di firmware e software per dispositivi IoT ed embedded. Nerves semplifica il processo di sviluppo e distribuzione di software per dispositivi embedded, consentendo agli sviluppatori di creare immagini firmware personalizzate e di integrare le funzionalità richieste dai dispositivi.

Un esempio di utilizzo di Nerves nell'ambito IoT si ha con l'azienda Spark-Meter [19], un'azienda con l'obiettivo di aumentare l'accesso all'elettricità offrendo soluzioni di gestione della rete che consentono alle aziende di servizi pubblici nei mercati emergenti di gestire sistemi finanziariamente sostenibili, efficienti e affidabili.

I loro prodotti sono contatori elettrici intelligenti e software di gestione della rete. Questi possono essere utilizzati per misurare il consumo di elettricità, raccogliere informazioni sulla salute di una rete elettrica e gestire la fatturazione.

Una panoramica della loro architettura è mostrata in figura 1.2, dove sono mostrati degli Smart Meters, dei sistemi embedded responsabili di collezionare misure del consumo dell'elettricità. Comunicano l'un l'altro tramite una rete mesh e comunicano con il Grid Edge management Unit, un altro sistema Embedded che riceve e processa dati da migliaia di smart meters. Il Grid Edge management unit comunica con il cloud che processa i dati così da essere mostrati da una User interface [4]. Sia il Grid Edge Management Unit che il server fanno uso di Elixir, infatti l'infrastruttura dei sistemi Embedded non

sono affidabili, la rete per la comunicazione con cui comunicano i dispositivi sono via radio, possono diventare irraggiungibili per vari fattori, come una mancanza di elettricità, guasto o mancanza di rete. Il sistema così ha bisogno di essere Fault-tolerant ed è uno dei cavalli di battaglia della piattaforma Erlang/OTP. Inoltre utilizzando il meccanismo Port di Elixir, vengono usati i vantaggi di Rust per processare i dati dagli Smart Meters e processare i dati in modo efficiente.

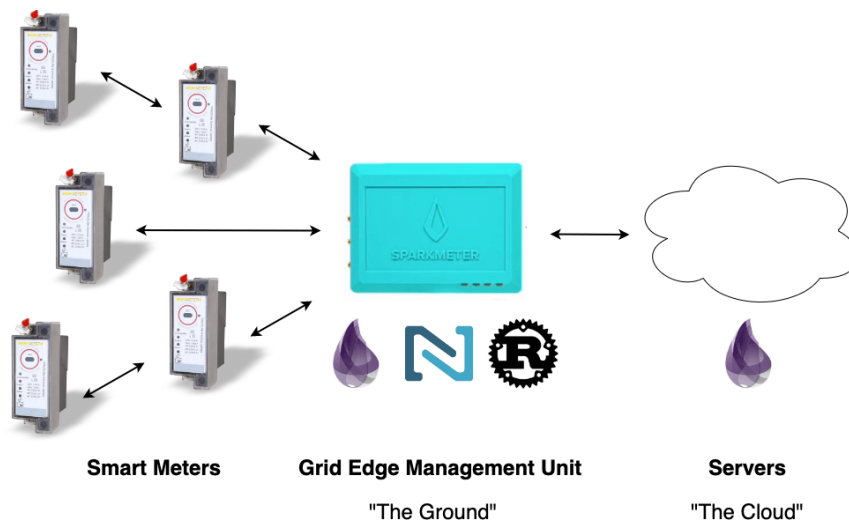


Figura 1.2: Un esempio di architettura IoT di SparkMeter[4]

1.1.2 Introduzione allo studio svolto

Il trattato esplora Elixir concentrandosi su due aspetti principali: la semplicità e le performance. Si analizzano i punti di forza di un linguaggio funzionale e come questi sono sfruttati in Elixir, con un focus sulla concorrenza. Nella scelta di un linguaggio, la semplicità è fondamentale e deve essere accessibile a tutti i programmatori. Tuttavia, l'efficienza è altrettanto importante, quindi vengono condotti test empirici per valutare le performance di Elixir.

In particolare il lavoro effettuato è così ripartito:

- Nel capitolo 2 si discute del linguaggio funzionale, esaminando le astrazioni offerte da Elixir per lo sviluppo di codice fault tolerant, si tratta la concorrenza e come la Erlang VM si occupa della gestione dei processi.
- Nel capitolo 3 si spiega il lavoro sperimentale svolto e i risultati ottenuti, in particolare vengono effettuati quattro test, i primi due per valutare

l'efficienza della concorrenza della VM, il terzo per sperimentare l'interoperabilità con il codice C/C++ confrontando le performance del codice C/C++ con quello di Elixir, infine, nel quarto test si misura il throughput di un server Http scritto in Elixir confrontandolo con dei semplici server scritti in Node e Python

Capitolo 2

Caratteristiche di Elixir

2.1 Introduzione

In questo capitolo, esamineremo le caratteristiche distintive di Elixir, un linguaggio di programmazione funzionale e concorrente che sfrutta appieno la potenza della piattaforma OTP (Open Telecom Platform). Non si vuole coprire ogni dettaglio del linguaggio, ma mettere in evidenza le caratteristiche fondamentali per iniziare a capire come pensare il codice con questo linguaggio e quali sono le caratteristiche che lo rendono un'opzione da considerare in determinati contesti.

Elixir, scritto in Erlang ed eseguito sulla macchina virtuale Erlang (BEAM), eredita gli obiettivi di Erlang, ma apporta miglioramenti significativi per rendere il linguaggio più appetibile e moderno.

Erlang, nato nel 1986, è stato progettato per semplificare lo sviluppo di software concorrente robusto e scalabile. Elixir si basa su queste fondamenta solide, offrendo un'API più pulita e astrazioni dell'OTP che consentono ai programmatori di ragionare a un livello più elevato, facilitando la scrittura di codice concorrente in modo intuitivo.

Una delle massime principali di Erlang e, di conseguenza, di Elixir, è "Let it crash" (Lascia che si schianti), che riflette l'approccio alla gestione degli errori nei sistemi concorrenti, incoraggiando la gestione degli errori tramite il rilancio e la supervisione anziché il blocco del processo.

Per capire come lavorare con questo linguaggio, bisogna affrontare un po' di questioni e farsi un po' di domande. Bisogna capire come la macchina virtuale Beam affronta la concorrenza, Elixir in particolare è un linguaggio orientato alla concorrenza e le astrazioni che fornisce sono proprio per far sì che si programmi in modo concorrenziale portando ad avere un codice responsivo. Gestendo in modo ottimale i processi attraverso il meccanismo di

Supervision, il software diventa fault-tolerant. Un altro punto che si affronta è l'immutabilità dei dati, un concetto chiave in Elixir ed Erlang, ed è proprio questa caratteristica che ci semplifica la programmazione concorrentiale.

2.2 Software utilizzati

I studi sulle performance riportati nel capitolo 3 sono stati effettuati utilizzando il sistema operativo Linux per scelta personale e per la varietà di software disponibile su Linux. Viene riportata l'installazione effettuata su Linux per i vari test e le relative configurazioni, in modo da poter testare gli esempi riportati in questo capitolo, l'installazione su Windows è immediata, infatti è disponibile sia l'installer per Erlang al seguente link: <https://www.erlang.org/downloads.html> e l'installer anche per elixir nel sito ufficiale <https://elixir-lang.org/install.html>, mentre su linux necessita qualche configurazione in più e viene riportata l'installazione tramite asdf (un tool version manager).

2.2.1 Installazione software utilizzati

Per avere l'ultima versione di Erlang ed Elixir su Linux bisogna affidarsi al Tool version manager **asdf** in quanto le versioni disponibili nei package manager della distro di riferimento non sono aggiornate all'ultima versione disponibile.

Installazione asdf

Asdf è un tool version manager open source che consente di installare, gestire e utilizzare diverse versioni di software senza dover installare o gestire separatamente ciascuna versione. È soprattutto utile quando si lavora su progetti che richiedono versioni specifiche del linguaggio. Nel nostro caso ci permette di scegliere l'ultima versione disponibile su qualunque distro linux indipendentemente dalla distro scelta.

Per l'installazione di asdf basta seguire le istruzioni della documentazione ufficiale del tool al seguente URL: <https://asdf-vm.com/guide/getting-started.html>

Le dipendenze necessarie da installare sono **curl** e **git**, una volta installate le dipendenze richieste basta clonare la repository. In una distro debian i comandi risultano i seguenti:

```
> sudo apt install curl git
> cd $HOME
> git clone https://github.com/asdf-vm/asdf.git ~/.asdf --branch v0.14.0
```

Inserire le seguenti righe nel file di configurazione `.bashrc`:

```
. "$HOME/.asdf/asdf.sh"  
. "$HOME/.asdf/completions/asdf.bash"
```

Assicurarsi che l'installazione sia avvenuta con successo:

```
> asdf --version  
  
v0.14.0
```

Installazione Erlang

Una volta installato asdf, bisogna installare Erlang seguendo le istruzioni della seguente repository: <https://github.com/asdf-vm/asdf-erlang>. Per una distro ubuntu si richiedono le seguenti dipendenze da installare con il package manager apt:

```
sudo apt-get -y install build-essential autoconf m4 libncurses5-dev  
libglu1-mesa-dev libpng-dev libssh-dev unixodbc-dev xsltproc  
fop libxml2-utils libncurses-dev openjdk-11-jdk
```

Per evitare warning nell'intellisense di VSCode, prima di procedere con i comandi di asdf, bisogna configurare le seguenti variabili d'ambiente per compilare la documentazione di Erlang necessaria per il corretto funzionamento dell'intellisense di VSCode:

```
> export KERL_BUILD_DOCS=yes  
> export KERL_INSTALL_HTMLDOCS=yes  
> export KERL_INSTALL_MANPAGES=yes
```

Facendo attenzione ad usare la stessa sessione del terminale in cui sono stati eseguiti i precedenti comandi eseguire:

```
> asdf plugin add erlang https://github.com/asdf-vm/asdf-erlang.git  
> asdf install erlang 26.2  
> asdf global erlang 26.2
```

Si può anche scegliere una versione differente di Erlang, fare attenzione alla versione dell'OTP che viene installata in quanto sarà necessaria per l'installazione di Elixir.

Se tutto è andato a buon fine si può testare l'avvenuta installazione tramite il comando `erl` che avvia l'ambiente interattivo di erlang.

Installazione Elixir

Per Elixir il procedimento con asdf è simile, bisogna solo fare attenzione alla versione OTP di Erlang installata, in quanto Elixir viene compilato con diverse versioni dell'OTP, fare riferimento al seguente link per il procedimento di installazione: <https://github.com/asdf-vm/asdf-elixir>

Per una distro debian le dipendenze richieste sono:

```
> sudo apt-get install unzip
```

Eeguire i seguenti comandi per procedere con l'installazione:

```
> asdf plugin-add elixir https://github.com/asdf-vm/asdf-elixir.git
> asdf install elixir 1.16.0-otp-26
> asdf global elixir 26.2
```

Se tutto è andato a buon fine si può testare elixir avviando l'ambiente interattivo di elixir **iex**.

Configurazione VsCode

VsCode supporta numerosi linguaggi di programmazione, è gratuito e open source, è un editor leggero che consente un'esperienza di sviluppo piacevole e produttiva con una vasta gamma di estensioni installabili. Per quanto riguarda Elixir, è possibile installare l'estensione **ElixirLS** disponibile al seguente link: <https://marketplace.visualstudio.com/items?itemName=JakeBecker.elixir-ls> Installabile anche direttamente dall'extension manager di VsCode.

2.3 Il paradigma funzionale

Come già accennato Elixir è un linguaggio funzionale, dove il concetto di funzione ricopre il ruolo di protagonista, i dati sono immutabili e il codice è dichiarativo.

Questo modo di vedere le cose deriva dal Lambda calcolo o λ -calcolo [13] un sistema formale definito da Alonzo Church nel 1936, sviluppato per definire formalmente le funzioni e il loro calcolo.

In un paradigma basato su stati come la programmazione ad oggetti spesso si hanno variabili condivise mutabili, di conseguenza più parti del codice possono riferirsi alla stessa variabile, complicando la programmazione multi-threading dovendosi preoccupare di meccanismi come il blocco sincronizzato o il locking per evitare le race condition tra più parti del codice. In questo modo non è immediato scrivere del codice concorrentiale sicuro aumentando la probabilità di errori e riscontrando possibili comportamenti indeterminati.

In un paradigma funzionale invece si prediligono le variabili immutabili che aggirano questo problema riducendo il rischio di scrivere codice concorrentiale non sicuro.

Cambiare paradigma non è immediato, un paradigma si può dire che definisce il modo di pensare al problema, nella programmazione ad oggetti

per esempio si definiscono le cosiddette classi, pensando al problema come ad oggetti che hanno un comportamento, e cambiano stato nel tempo. Questo può essere un modo di sviluppare intuitivo ma non sempre ottimale per la risoluzione di problemi concorrenziali. Nella programmazione funzionale si cambia prospettiva, ovvero si ha un input, si passa l'input alla funzione e si ottiene la trasformazione dell'input ottenendo l'output, quindi si trasformano dati e non si mutano.

In poche parole un linguaggio funzionale assume che scrivere un software complesso sia più facile nel momento in cui il codice ha queste proprietà:

- I dati sono immutabili
- Le funzioni sono pure, ovvero, il risultato di una funzione dipende soltanto dai suoi parametri in input.
- Le funzioni non generano effetti oltre il suo valore restituito.

Con queste proprietà si ha più controllo del flusso del programma, anche se non sempre possono essere soddisfatte anche in un linguaggio funzionale.

2.3.1 Struttura di un progetto Elixir

Elixir è un linguaggio moderno, e come ogni linguaggio moderno che si rispetti fornisce un tool per la creazione e configurazione di progetti, questo tool si chiama **Mix**.

Il tool Mix

È possibile creare un progetto con il comando:

```
mix new <nome-progetto>
```

Verrà creata una struttura per il progetto come nell'esempio 2.1, il codice sarà organizzato nella cartella **lib**, viene creata una cartella **test** per gli unit test, e il file per la configurazione del progetto **mix.exs**. Da notare che anche la configurazione del progetto in **mix.exs** avviene tramite funzioni.

```
.
>build
>deps
-->...
>lib
-->example.ex
mix.exs
README.md
>test
-->example.exs
```

Esempio 2.1: Struttura progetto

Come sappiamo Elixir fornisce anche un ambiente interattivo (**iex**) per testare il nostro codice, ed è consentito avviare questo ambiente nel contesto della nostra applicazione con il comando:

```
iex -S mix
```

Con Mix possiamo includere e scaricare facilmente anche librerie esterne attraverso il package manager definendo la dipendenza nel file di configurazione `mix.exs`.^[11]

Moduli

Elixir organizza il codice in Moduli, permettendo di definire le funzioni dentro dei namespace, così da separare le responsabilità delle funzioni.

Ci sono varie cose che si possono definire dentro un modulo, si possono definire delle **struct** ma cosa più importante si possono definire i cosiddetti **Behaviour**, un modo per definire un'interfaccia Api da poter riutilizzare per altri moduli. Elixir fornisce delle astrazioni proprio attraverso questi Behaviour, come il `GenServer` ed il `Supervisor`.

Ciò che si vuole evidenziare ora è che il progetto è definito in moduli, il modo che Elixir fornisce per organizzare il codice separando le funzioni in namespace e poter dare caratteristiche alle funzioni.

2.3.2 Basi dichiarative

Come già accennato, Elixir adotta un approccio dichiarativo nella definizione delle funzioni. Questo si contrappone all'approccio imperativo, che si concentra su "come posso risolvere questo problema?", mentre quello dichiarativo si pone la domanda "come posso definire un problema?".

Nell'esempio 2.2 è presentato un approccio imperativo al problema "somma dei primi n elementi" nel linguaggio C, mentre nell'esempio 2.3 è presentato l'approccio dichiarativo con Elixir.

```
1 int sum_first_n(n){
2     int sum=0;
3     for(int i=1;i++;i<=n){
4         sum+=i;
5     }
6     return sum;
7 }
```

Esempio 2.2: Somma N elementi

```
1 defmodule Sum do
2   def sum_recursive(0), do: 0
3   def sum_recursive(n), do: n + sum_recursive(n - 1)
4 end
```

Esempio 2.3: Somma N elementi

Pattern matching

L'approccio dichiarativo è rafforzato tramite il meccanismo del **Pattern Matching**, infatti in Elixir l'operatore `=`, non è un operatore di assegnazione come nei più comuni linguaggi, ma è comparabile all'equivalente algebrico. Quest'operatore ci permette di scrivere dell'equazioni che condizionano il flusso del codice, il controllo del flusso in questo linguaggio è pesantemente influenzato da questo meccanismo, infatti si vedono i costrutti *if - else* molto raramente. Questo è molto utile per l'approccio dichiarativo, infatti possiamo definire più corpi della stessa funzione, ed Elixir capisce quale funzione invocare in base al valore dei suoi parametri attraverso questo meccanismo, è già stato utilizzato implicitamente nell'esempio 2.3, dove viene invocata la funzione `sum_recursive(0)` quando il valore dell'argomento vale 0, altrimenti chiamerà quello con l'argomento `"n"` assegnando il valore dell'argomento alla variabile `n`.

Elixir cerca di assegnare il valore a destra dell'equazione al valore di sinistra cercando di risolvere l'equazione, tenta di fare un assegnazione dove possibile per risolvere l'equazione. Questo ci permette di usare quest'operatore per poter scomporre un dato, viene mostrato nell'esempio 2.4 tramite l'ambiente interattivo *iex*.

```
1 # Lists
2 iex> list = [1, 2, 3]
3 [1, 2, 3]
4 iex> [1, 2, 3] = list
5 [1, 2, 3]
6 iex> [] = list
7 ** (MatchError) no match of right hand side value: [1, 2, 3]
8 iex> [1 | tail] = list
9 [1,2,3]
10 iex> tail
11 [2,3]
```

Esempio 2.4: Pattern Matching

2.3.3 Transizione al funzionale

Cambiare paradigma può essere difficoltoso, bisogna cambiare prospettiva, ma con le giuste intuizioni può risultare semplice. Per fare una transizione

mentale al funzionale bisogna capire soprattutto qual'è la differenza rispetto ad un linguaggio basato a stati.

Elixir non ha oggetti, il linguaggio ha un forte focus sull'immutabilità. In Elixir trasformiamo dati piuttosto che mutarli, infatti da questo punto di vista probabilmente è più naturale il paradigma funzionale piuttosto che un procedurale, ad esempio in un linguaggio procedurale possiamo scrivere:

```
1 my_array = [1,2,3]
2 do_something_with_array(my_array)
3 print(my_array)
```

Ci potremmo aspettare che la stampa dell'array sia [1,2,3] quando in realtà l'output dipende da cosa fa la funzione, in molti linguaggi un array viene passato per riferimento. In Elixir non è implicito, ma l'assegnazione deve essere sempre esplicita come segue:

```
1 my_list = [1,2,3]
2 my_list = do_something_with_array(my_list)
3 IO.inspect(my_list)
```

Se *my_list* non viene riassegnata, *my_list* non cambia.

Il problema di questo approccio arriva nel momento in cui ci serve uno stato da condividere, infatti una semplice variabile globale non esiste in Elixir, e per mantenere uno stato dobbiamo affidarci ad un altro processo che lo mantiene per noi, ed Elixir per questo ci da delle astrazioni a cui fare affidamento come il Modulo **Agent** o il **GenServer** che verranno trattati in seguito.

2.3.4 Conseguenze sulle prestazioni dell'immutabilità

L'immutabilità ci consente di avere un buon controllo sul flusso del codice, si può pensare che può essere inefficiente e dovendo sempre copiare dati, inoltre si può pensare che si creano molti dati non più utilizzati da liberare in memoria con il Garbage collector portando anche il garbage collector ad essere inefficiente. Questo è vero in parte, per alcune tipologie di problemi come algoritmi che fanno uso di matrici, ma Elixir comunque sfrutta la poprietà dell'immutabilità per limitare l'inefficienza rendendo altri problemi più efficienti rispetto ad altri linguaggi.

Elixir sa che i dati sono immutabili, può riusarli in parte o per intero quando si creano nuove strutture. Il seguente esempio mostra come Elixir ottimizza la creazione di nuovi dati:

```
1 iex> lista1 = [ 3, 2, 1 ]
2 [3, 2, 1]
3 iex> lista2 = [ 4 | lista1 ]
4 [4, 3, 2, 1]
```

La *lista2* infatti è creata usando la *lista1* sapendo che non cambierà mai durante il flusso di esecuzione, quindi mette semplicemente in coda la *lista1*, non dovendo creare una nuova copia della *lista1* per creare la *lista2*.

Garbage Collection

Un altro aspetto che può creare dubbi sul copiare dati, è che si lasciano spesso vecchi dati non più utilizzati da dover pulire con il Garbage Collector.

Scrivendo codice Elixir però ci si rende subito conto che ci porta a sviluppare codice con migliaia di processi leggeri, ed ogni processo ha il proprio heap. I dati nell'applicazione quindi sono suddivisi nei processi, ogni heap è molto piccolo rendendo il garbage collector più veloce. Se un processo termina prima che il suo heap diventi pieno, tutti i suoi dati vengono eliminati, senza necessità di liberare la memoria rendendo il garbage collector efficiente. [21]

2.4 Concorrenza

La concorrenza è un concetto fondamentale nell'ambito dello sviluppo software, si riferisce alla capacità di eseguire più attività contemporaneamente. Questo è particolarmente importante in applicazione che devono gestire molteplici operazioni in parallelo come ad esempio un server web, applicazioni di elaborazione dati e sistemi real time.

La concorrenza consente a un programma di sfruttare appieno le risorse disponibili, e migliorando la responsività di un programma. Inoltre, permette di creare sistemi più reattivi e scalabili, in grado di gestire un numero crescente di richieste senza compromettere le prestazioni.

In generale, la gestione della concorrenza in molti linguaggi può essere complessa e soggetta a errori o comportamenti indeterminati, infatti, più processi o thread possono interagire tra loro in modo imprevedibile, essendo così costretti a usare tecniche di locking in concomitanza di memoria condivisa tra più thread.

Nel contesto di Elixir, la concorrenza è un concetto centrale e viene gestita attraverso il modello di programmazione basato su processi leggeri, tutti isolati tra loro con il proprio stack ed il proprio heap.

2.4.1 La concorrenza in Beam

La concorrenza gioca un ruolo chiave per un software che vuole essere altamente responsivo. La piattaforma Erlang fa uso dei cosiddetti processi leggeri, non sono processi del sistema operativo, ma processi della VM Beam,

chiamato processo e non thread in quanto non condividono memoria e sono completamente isolati tra di loro a differenza dei thread.

Un server tipico deve gestire migliaia di richieste, e gestirle concorrentialmente è essenziale per non far rimanere in attesa il client. Quello che si vuole è gestirli parallelamente il più possibile sfruttando più risorse della Cpu disponibile. Quello che fa la macchina virtuale per noi è permetterci la scalabilità, più richieste allora più risorse da allocare.

Inoltre, siccome il processo è isolato, un errore in una richiesta può essere localizzato senza avere impatto sul resto del sistema, così creando anche un sistema robusto agli errori.

Un processo appena creato occupa in memoria 326 words [7], quindi in una macchina a 64 bit occupa 2608 byte. Lo si può vedere in Elixir facilmente con l'esempio 2.5.

```
1 defmodule Examples.Memory do
2   def mypid do
3     receive do
4       :stop -> :exit
5       _ -> mypid()
6     end
7   end
8
9   def benchmark do
10    pid = spawn(fn -> mypid() end)
11    {_,byte_used} = :erlang.process_info(pid,:memory)
12    IO.puts("La memoria usata dal processo e': #{byte_used} byte")
13    send pid, :stop
14  end
15 end
```

Esempio 2.5: Memoria in un processo

Infatti l'output ottenuto dall'esempio 2.5 è:

```
iex> Examples.Memory.benchmark
La memoria usata dal processo e': 2640 byte
:stop
```

Si nota dall'output dell'esempio che la memoria utilizzata è leggermente superiore della memoria minima dichiarata, infatti il processo non fa nulla di particolare oltre ad aspettare il messaggio di ":stop". È da notare che è proprio questa leggerezza nei processi che permette al linguaggio di essere orientato alla concorrenza, permettendo di usare i processi senza troppe remore.

Un altro punto da notare è che si può migliorare significativamente la reattività del programma ma non l'efficienza totale del sistema, infatti non tutti i processi sono eseguiti in parallelo, quindi in una macchina con quattro processori non si possono eseguire più di 4 processi per volta ed in un normale software Elixir è normale avere migliaia di processi che lavorano, lavorando

con migliaia di processi però permette di distribuire il delay di operazioni concorrenti.

Facendo un esempio con una Cpu dual-core, i processi vengono eseguiti concorrentemente e gestiti dagli Scheduler della VM, in figura 2.1 è mostrato come la VM gestisce i processi, la VM è un unico processo nel sistema operativo e alloca uno scheduler per ogni core logico disponibile sulla macchina, gli scheduler si occupano di gestire l'esecuzione dei processi uno per volta.

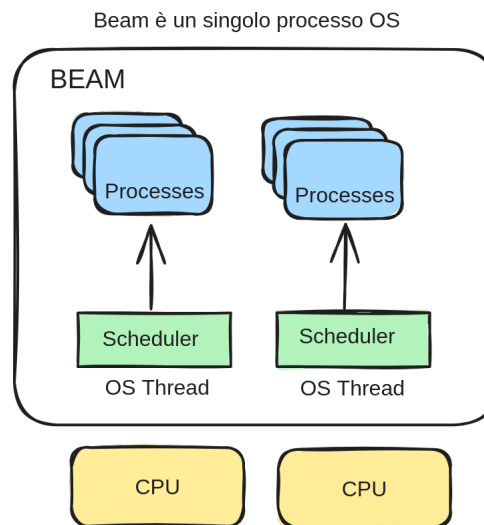


Figura 2.1: Concorrenza nella VM Beam [12]

2.4.2 Concorrenza basata su attori

Elixir usa un modello di concorrenza basato su attori, gli attori sono entità di elaborazione indipendenti che eseguono operazioni in modo asincrono, questi attori non sono altro che processi che vengono identificati attraverso un **PID** univoco. Come già detto sono isolati l'uno dall'altro e comunicano solo attraverso lo scambio di messaggi, questo scambio avviene attraverso dei canali di comunicazione detti "**mailbox**". Ogni processo ha una propria mailbox dove avviene la ricezione del messaggio da parte di altri processi.

I processi sono identificati da un PID univoco, e conoscendo il PID di un processo può avvenire la comunicazione attraverso la primitiva fornita dal linguaggio **send/2** che permette di inviare un messaggio ad un processo come avviene nell'esempio 2.5, dove il processo principale crea un altro processo che rimane in ascolto tramite il blocco **receive/1**. Quest'ultimo quando riceve il messaggio di **:stop** viene terminato concludendo il flusso definito dal processo.

Se non ci sono messaggi nella mailbox, il processo aspetta fino a quando non arriva un messaggio, in particolare, nell'esempio tutti i messaggi che non siano `:stop`, verranno ignorati continuando ad ascoltare altri messaggi chiamando ricorsivamente se stesso.

Process Linking e Process monitoring

I processi sono isolati, ma si può legare un processo al processo chiamante tramite un operazione di Process linking, questo per sapere se il processo va in errore oppure termina per un comportamento imprevisto, legando i processi si può propagare l'errore nel caso uno dei due processi non ha senso di esistere preso da solo.

Spesso i processi vengono legati ad un Supervisor, che lascerà andare in errore i processi figli, e si occuperà semplicemente di riavviarli seguendo una strategia scelta. Questa è la filosofia "Let it Crash" di Erlang, che si contrappone alla gestione delle eccezioni di altri linguaggi. Si veda l'esempio 2.6

```
1 iex(1)>spawn(fn -> raise "oops" end)
2 #PID<0.58.0>
3
4 [error] Process #PID<0.58.00> raised an exception
5 ** (RuntimeError) oops
6 (stdlib) erl_eval.erl:668: :erl_eval.do_apply/6
7
8 iex(2)spawn_link(fn -> raise "oops" end)
9
10 17:37:50.425 [error] Process #PID<0.113.0> raised an exception
11 ** (RuntimeError) oops
12 ** (EXIT from #PID<0.110.0>) shell process
13 ** exited with reason: {%RuntimeError{message: "oops"}, []}
14
15 iex(2)
```

Esempio 2.6: Process linking

Notiamo nell'esempio 2.6 che con il comando la funzione `spawn_link/1` il processo va in errore e propaga l'errore all'ambiente interattivo `iex`, riavviando anche l'ambiente `iex`.

Nel caso invece non si vuole propagare l'errore al processo chiamante, si può usare il monitoring tramite la primitiva `spawn_monitor`, in questo caso il processo chiamante non va in errore insieme al processo chiamato, ma riceve un messaggio di `:DOWN` nella mailbox da poter gestire a piacimento.

Si nota in conclusione al paragrafo che è possibile gestire migliaia di processi molto facilmente senza incrementare in modo considerevole le risorse in memoria grazie ai processi leggeri, permettendo di sviluppare un software orientato alla concorrenza, tutto è facilitato dall'isolamento di ogni processo che permette di non preoccuparsi di meccanismi di locking e sincronizzazione.

Elixir permette di non preoccuparci di come scalare le risorse hardware, potendo dare responsabilità ad altri processi creando più flussi di esecuzione.

2.5 OTP - Open Telecom Platform

L'OTP (Open Telecom Platform) è stato sviluppato da Ericsson negli anni '80 come insieme di strumenti, librerie e standard per affrontare le sfide specifiche del settore delle telecomunicazioni, come l'affidabilità e la tolleranza agli errori.

Basato sul linguaggio di programmazione Erlang, OTP è stato progettato per gestire la concorrenza e le comunicazioni asincrone, rendendolo ideale per sistemi distribuiti. Nel corso degli anni, l'OTP ha trovato applicazioni anche in altri settori, diventando una scelta popolare per sistemi altamente affidabili e scalabili.

Elixir fornisce un'interfaccia più moderna e una sintassi più chiara rispetto ad Erlang, mantenendo al tempo stesso la potenza e l'affidabilità di Erlang e OTP.

Per quanto riguarda il mantenimento di uno stato, in Elixir si può fare affidamento ai Behaviour Agent e GenServer, consentendo di dare la responsabilità al mantenimento di uno stato ad un altro processo.

ci consentono di mantenere uno stato senza dover reinventare la ruota nello scrivere un modulo soltanto per farlo. L'approccio nella programmazione è totalmente differente e piuttosto singolare rispetto ai più comuni linguaggi di programmazione, ma è proprio questa singolarità che può porta il linguaggio ad essere orientato alla concorrenza.

2.5.1 GenServer

Per mantenere uno stato si potrebbe scrivere un modulo che tramite le primitive concorrentiali gestiscono lo stato a piacimento. Elixir però mette a disposizione il Behaviour GenServer, un OTP server è un modulo con il "Behaviour" GenServer. Il "Behaviour" è un meccanismo che consente di definire uno schema comune per un tipo specifico di processo.

Ad esempio il GenServer Behaviour definisce le funzioni e le interfacce necessarie per creare un processo server in grado di gestire le richieste in modo asincrono. Utilizzando il GenServer Behaviour, è possibile definire i comportamenti di base del server e personalizzarli secondo le esigenze specifiche dell'applicazione. Questo fornisce un alto livello di astrazione per la gestione dei processi e semplifica lo sviluppo di sistemi concorrenti e distribuiti in Elixir.

Il vantaggio di utilizzare un GenServer è che ha un'insieme di interfacce standard e include funzionalità di tracciamento e segnalazione degli errori. Si può anche mettere dentro un albero di supervisione.

Questo Behaviour astrae l'interazione Client-server, come si può vedere in figura 2.2 [9], lo stato è gestito dal Server, e i processi client sono quelli che devono modificare lo stato o accedere ad esso.

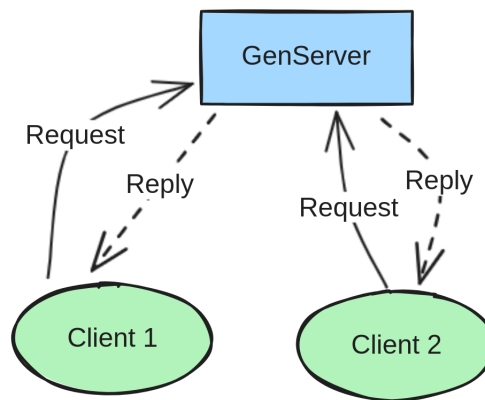


Figura 2.2: Interazione Client-Server

Per implementare il behaviour GenServer, bisogna affidarsi alla documentazione di GenServer, in particolare vanno ridefinite delle callback, ed ogni funzione può restituire un determinato insieme di strutture dati.

Nell'esempio 2.7 viene implementata una struttura dati per mantenere uno Stack di dati [9].

```

1  defmodule Stack do
2  use GenServer
3
4  # Client
5
6  def start_link(default) when is_binary(default) do
7    GenServer.start_link(__MODULE__, default)
8  end
9
10 def push(pid, element) do
11   GenServer.cast(pid, {:push, element})
12 end
13
14 def pop(pid) do
15   GenServer.call(pid, :pop)
16 end
17
18 # Server (callbacks)
19
20 @impl true
21 def init(elements) do
22   initial_state = String.split(elements, ",", trim: true)
23   {:ok, initial_state}

```

```

24 end
25
26 @impl true
27 def handle_call(:pop, _from, state) do
28   [to_caller | new_state] = state
29   {:reply, to_caller, new_state}
30 end
31
32 @impl true
33 def handle_cast({:push, element}, state) do
34   new_state = [element | state]
35   {:noreply, new_state}
36 end
37 end

```

Esempio 2.7: Implementazione Stack

Nell'esempio possiamo vedere che il modulo `stack` implementa la funzione `init/1` che inizializza lo stato con gli elementi iniziali quando il server viene avviato, la funzione `handle_call/3` chiamata per le operazioni di `:pop` dello stack, è una funzione sincrona, quindi viene usata quando ci si aspetta un valore di ritorno, infatti restituisce il valore di testa dello stack. La funzione `handle_cast/2` invece viene usata per le operazioni asincrone, quindi nel caso in esame per l'operazione di push nello Stack che non necessita di risposta.

Quindi il `GenServer` è un'astrazione che:

- Incapsula un servizio condiviso.
- Mantiene uno stato.
- Permette un'astrazione concorrente ad un servizio condiviso [1].

2.5.2 Supervisor

Un'altro concetto cardine dell'OTP è il Supervisor con cui si riesce a raggiungere un alto livello di Fault-tolerant.

Il compito principale di un supervisor è quello di monitorare, controllare e gestire il ciclo di vita dei processi all'interno di un sistema Erlang o Elixir. I supervisor sono particolarmente utili per garantire la stabilità e l'affidabilità dei sistemi distribuiti, poiché gestiscono automaticamente il riavvio dei processi in caso di fallimenti, garantendo che il sistema continui a funzionare anche in situazioni critiche.

In pratica come il `GenServer`, un Supervisor è un modulo che implementa il Behaviour Supervisor.

Ci sono diversi tipi di Supervisor in Erlang ed Elixir, tra cui:

- **Supervisor Semplice:** Questi supervisor monitorano direttamente i processi figlio e li riavviano se necessario. Sono utilizzati per gestire processi che non hanno stati interni o che devono essere riavviati senza alcuna elaborazione specifica.
- **Supervisor gerarchici:** Questi supervisor controllano una gerarchia di altri supervisor e processi. Possono essere configurati per riavviare solo parti specifiche del sistema in caso di problemi, consentendo un maggiore controllo sul comportamento di riavvio.
- **Supervisor dinamici:** Questi supervisor possono essere creati e configurati dinamicamente durante l'esecuzione del programma, consentendo una maggiore flessibilità nell'aggiunta e nella gestione dei processi.

Per implementare un Supervisor si deve decidere quali sono i processi figli da supervisionare, e si avvia il Supervisor con la funzione `start_link/2`. Bisogna quindi decidere quali sono i processi figli da supervisionare, e una volta avviato il Supervisor con la funzione `start_link/2`, questo ha bisogno di sapere come avviare, fermare o riavviare i suoi figli in caso di errore o uscita imprevista.

Una volta avviato il Supervisor, deve sapere come fare lo start/stop/restart dei suoi figli da monitorare. Per questo i moduli da supervisionare devono implementare una funzione che definisce le specifiche, la funzione in questione è `child_spec/1`, restituisce una Map per configurare il comportamento in caso di supervisione.

Alcuni moduli come il GenServer già definiscono questa funzione non avendo la necessità di ridefinirla, come il GenServer e il modulo Task che utilizzeremo successivamente.

2.6 Sistemi distribuiti

Nell'era dell'informatica moderna, la necessità di sviluppare applicazioni scalabili, affidabili e resilienti è diventata sempre più critica. In risposta a questa esigenza, i sistemi distribuiti sono emersi come una soluzione efficace per gestire carichi di lavoro complessi su più nodi di calcolo, consentendo alle applicazioni di crescere e adattarsi dinamicamente alle esigenze in continua evoluzione degli utenti.

Elixir offre un ambiente potente e flessibile per lo sviluppo di sistemi distribuiti. Grazie al suo modello di programmazione ad attori e al supporto integrato per la comunicazione distribuita, Elixir è ideale per la creazione di applicazioni distribuite altamente scalabili, affidabili e resilienti.

Utilizzando il modello di programmazione ad attori, le applicazioni possono essere decomposte in processi leggeri (attori) che comunicano tra loro attraverso lo scambio di messaggi asincroni. Questo modello semplifica la gestione della concorrenza e delle comunicazioni tra nodi, consentendo alle applicazioni di gestire carichi di lavoro elevati in modo efficiente e affidabile.

La robustezza è sicuramente un obiettivo primario nei progetti distribuiti, e Elixir fornisce strumenti e astrazioni che semplificano la gestione degli imprevisti e dei guasti. La capacità di inviare messaggi in remoto tra i nodi, utilizzando gli stessi meccanismi utilizzati per la comunicazione locale, è un esempio di come Elixir semplifica lo sviluppo di sistemi distribuiti, garantendo al contempo affidabilità e resilienza.

È importante sottolineare che molti dei concetti e delle pratiche utilizzate nello sviluppo di sistemi distribuiti in Elixir sono ereditati dall'Erlang Runtime, il che garantisce una base solida e comprovata per la costruzione di applicazioni distribuite.

Uno dei principali vantaggi dei sistemi distribuiti in Elixir è la capacità di scalare orizzontalmente aggiungendo nuovi nodi al cluster. Ciò consente alle applicazioni di crescere in modo flessibile con l'aumento del traffico e di garantire un servizio continuo agli utenti finali anche in caso di errori o guasti dei nodi.

2.6.1 Un esempio di comunicazione tra nodi

Un nodo in ogni sistema Erlang può essere identificato con un nome. Ad esempio si può avviare l'ambiente iex assegnando un nome all'istanza.

```
1 iex --sname nico@localhost
2
3 iex(nico@localhost)1>
4
```

Si può avviare un'altra istanza:

```
1 iex --sname kate@localhost
2
3 iex(nico@localhost)1>
```

Questi possono essere identificati come due nodi e comunicare tra loro tramite `Node.spawn_link/2`, prende due argomenti, il nome del nodo al quale connettersi, e la funzione da eseguire dal processo remoto.

Per la comunicazione tra i due nodi si usano le stesse primitive usate per la concorrenza, un nodo può essere visto come un processo.

Comunicazione su nodi in remoto

Per mandare messaggi e riceverli tra due nodi su macchine differenti, le istanze devono essere avviate con un cookie come segue:

```
1 iex --sname nico@localhost --cookie secret_token
```

Solo i nodi che condividono lo stesso cookie hanno il permesso di comunicare.

Un altro modo per comunicare è tramite `Node.connect/1`, i nodi si connettono tra loro, ed Elixir mantiene una connessione TCP aperta tra i due nodi. Se altri nodi si connettono, si mantengono connessioni l'un l'altro, si forma così una rete denominata 'fully meshed network'.

2.6.2 Considerazioni e sfide da considerare

La gestione delle connessioni tra i nodi distribuiti può essere complessa e non viene approfondita in questo luogo. La programmazione distribuita è un campo complesso ed Elixir prova a semplificarla usando le stesse primitive usate per la concorrenza, ma possono sorgere dei problemi.

La gestione delle connessioni tra i nodi distribuiti può essere complessa. È necessario garantire che i nodi possano comunicare tra loro in modo affidabile e che le connessioni rimangano stabili anche in presenza di interruzioni di rete o guasti hardware.

La comunicazione tra i nodi può essere vulnerabile agli attacchi di rete, quindi è importante implementare misure di sicurezza per proteggere i dati e prevenire accessi non autorizzati. Ciò include l'uso di connessioni sicure, autenticazione e autorizzazione dei nodi, e crittografia dei dati trasmessi.

La tolleranza agli errori è fondamentale nei sistemi distribuiti. È necessario gestire i guasti dei nodi in modo che l'applicazione possa continuare a funzionare in modo affidabile anche in presenza di errori o interruzioni di servizio.

La scalabilità è un altro aspetto importante nella programmazione distribuita. È necessario progettare il sistema in modo che possa gestire carichi di lavoro elevati e crescere in modo flessibile con l'aumento del traffico senza compromettere le prestazioni o l'affidabilità.

2.7 Integrazione con codice esterno

Elixir offre completa Interoperabilità con il codice Erlang, e si ha a disposizione tutta la gamma di librerie già sviluppate per Erlang, molte librerie non vengono riscritte in Elixir potendo usare quelle già ben testate sulla piattaforma Erlang.

Non tutto può essere implementato in modo efficiente usando Erlang ed Elixir, basti pensare che l'immutabilità dei dati porta molti vantaggi per lo sviluppo concorrentiale, ma porta altri svantaggi. Nella seguente lista vengono messi in evidenza alcune lacune che presenta la Erlang VM.

- **Matematica avanzata:** La VM non è stata progettata per operazioni intensive di calcolo numerico. Se l'applicazione dipende dal calcolo di statistiche si possono riscontrare delle limitazioni.
- **Soluzioni algoritmiche tramite matrici:** L'implementazione nativa di matrici in Elixir usa liste di liste, che non è una rappresentazione efficiente delle matrici multidimensionali. Inoltre la mancanza di mutabilità porta a fare una copia di matrici anche grandi per semplici operazioni.
- **Command Line Applications:** La VM impiega circa 0.3 secondi per l'avvio e lo spegnimento anche per l'hardware moderno, dopotutto Erlang è stato progettato per sistemi a lungo termine, ciò non porta ad essere la scelta ottimale per una CLI.

Ci possono essere un sacco di insiemi di problemi in cui Elixir non è la soluzione migliore, ma può risultare utile comunque usare Elixir e risolvere problemi in cui non è adatto con un codice esterno scritto in un altro linguaggio adatto alle esigenze specifiche. In particolare Elixir può comunicare con altri linguaggi in 3 modi diversi:

- **NIF (Native implemented functions):** L'applicazione può condividere lo stesso spazio degli indirizzi di memoria. Una NIF è una funzione scritta in C/C++, viene compilata come libreria condivisa e caricata all'avvio dell'applicazione.

- Ports: Elixir può invocare processi esterni alla macchina virtuale, è una forma di Interprocess communication (IPC).
- Erlang Distribution Protocol: Si può comunicare con processi esterni anche su macchine differenti attraverso il protocollo di distribuzione fornito per distribuire un'applicazione su più nodi scalando orizzontalmente un sistema, questo metodo non è stato sperimentato in questo luogo.

2.7.1 Interoperabilità tramite NIF

Le funzioni native NIF, permettono di caricare codice esterno nello stesso spazio di memoria della Erlang VM. Questa strategia di interoperabilità porta a dei benefici sulle performance elevati per alcune tipologie di calcolo come quello matematico. Erlang fornisce un API nel file header "erl_nif.h" [5] consentendo di scrivere funzioni C e poter essere chiamate direttamente dal programma Elixir/Erlang.

Queste funzioni C devono essere compilate come librerie condivise, shared object (.so in linux). Viene riportato nell'esempio 2.8 il codice NIF che stampa "Hello from C"[2].

```

1  #include "string.h"
2  #include "erl_nif.h"
3
4  static ERL_NIF_TERM hello(ErlNifEnv* env, int argc,
5                             const ERL_NIF_TERM argv[]) {
6      ErlNifBinary *output_binary;
7      enif_alloc_binary(sizeof "Hello from C", output_binary);
8      strcpy(output_binary->data, "Hello from C");
9      return enif_make_binary(env, output_binary);
10 }
11
12 static ErlNifFunc nif_funcs[] = {
13     {"hello", 0, hello},
14 };
15
16 ERL_NIF_INIT(Elixir.ElixirNif, nif_funcs, NULL, NULL, NULL, NULL)
17

```

Esempio 2.8: Funzione Nif

Si può compilare con il compilatore **gcc** con il seguente comando:

```

gcc -o <destination-directory>/hello.so -shared -fpic \
-I $ERL_ROOT/usr/include <file-directory>/hello.c

```

Una volta compilata la libreria si può caricare all'interno del modulo nel quale si vuole usare la funzione come nell'esempio 2.9.

```

1  defmodule ElixirNif do
2      @on_load :load_nif

```

```
3
4  def load_nif do
5    :ok = :erlang.load_nif(String.to_charlist("priv/elixir_nif"), 0)
6  end
7
8  def hello do
9    "Hello from Elixir"
10  end
11 end
```

Esempio 2.9: Caricamento NIF

L'annotazione `@on_load :load_nif` dice alla VM di eseguire `load_nif/0` all'avvio dell'applicazione. La funzione `hello` necessita di essere ridefinita anche in Elixir, così da permettere che la chiamata non fallisca nonostante qualche eventuale errore nel caricamento della NIF.

Nonostante è uno dei metodi più efficienti per eseguire codice C/C++, questo metodo ha degli svantaggi [5]:

- Una funzione nativa che va in crash, farà andare in crash l'intera VM, per questo è sconsigliato usare questo metodo se non strettamente necessario, questo rischio va contro la filosofia di Erlang nella costruzione di sistemi fault-tolerant.
- Una funzione nativa implementata erroneamente può causare un'inconsistenza dello stato interno della VM, che può portare al crash della VM o a comportamenti imprevisti della VM in qualsiasi momento dopo la chiamata alla funzione nativa.
- Una funzione nativa che viene eseguita a lungo degrada la responsività della VM e può portare a comportamenti strani, che possono portare a un utilizzo della memoria eccessivo.

Un esempio più complesso dell'utilizzo di una NIF è riportato nel paragrafo 3.4.5.

2.7.2 Interoperabilità con Port

Il meccanismo Port è un'alternativa più sicura per integrare codice esterno. Ogni Port comunica con un processo esterno del sistema operativo. In questo caso se il Port termina, il codice Elixir viene notificato con un messaggio e può agire di conseguenza continuando permettendo di avere un'interazione robusta in confronto alle NIF. Per esempio un Segmentation fault in un processo che comunica tramite Port non porta alla terminazione della VM. Per questi motivi prima di integrare codice esterno tramite NIF conviene sempre prendere in considerazione il meccanismo Port prima, per non perdere

i benefici dati dalla piattaforma Erlang. In Figura 2.3 si può vedere come la macchina virtuale comunica con un processo esterno attraverso questo meccanismo, il processo che crea il Port viene chiamato Processo connesso, a questo punto si comunica con il processo connesso in due modi:

1. Tramite l'API fornita da Elixir [18], si possono mandare messaggi in maniera sync.
2. Tramite le primitiva `send/2` si può inviare un messaggio al processo in maniera asincrona.

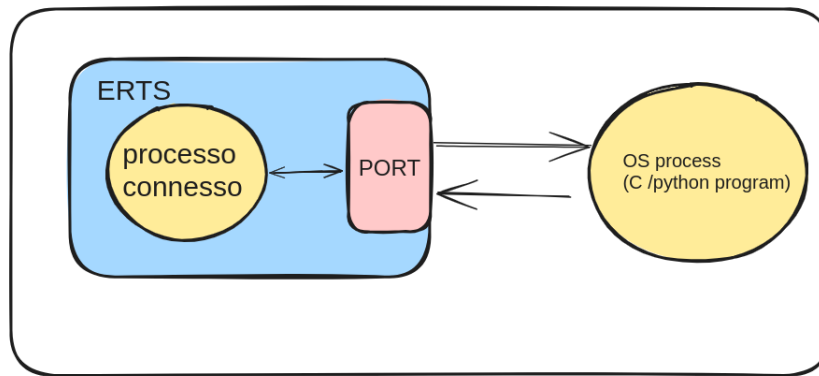


Figura 2.3: Comunicazione con Port [6]

Viene riportato nell'esempio 2.7.2, un apertura di una Port per il comando "echo" di bash.

```

1
2 iex> path = System.find_executable("echo")
3 iex> port = Port.open({:spawn_executable, path},
4 ... [:binary, args: ["hello world"]])
5 iex> flush()
6 [#Port<0.1380>, {:data, "hello world\n"}]
```

Un esempio di uso più complesso si trova nel paragrafo 3.4.4.

La libreria ErlPort

Un caso di particolare interesse può essere la comunicazione tra Elixir e python, per trarre vantaggio dalla vasta gamma di librerie offerte dalla community degli sviluppatori di Python.

Proprio in quest'ambito la libreria **ErlPort** ci consente di avviare istanze python da Elixir, una libreria open source costruita grazie al meccanismo Port, in grado di interagire con il linguaggio python o ruby. Non si entra in

dettaglio di questa libreria. ErlPort permette a Erlang e il linguaggio esterno di scambiare dati, invocare funzioni l'uno sull'altro e gestire lo stato condiviso in modo trasparente. Questo facilita l'integrazione di applicazioni scritte in Erlang con applicazioni scritte in python o ruby e consente di sfruttare le specifiche capacità di ciascun linguaggio all'interno di un unico sistema.

Ad esempio, con ErlPort, è possibile utilizzare Python per elaborare dati complessi o interfacce utente, mentre si sfruttano le potenti capacità di concorrenza di Erlang per gestire la comunicazione di rete o i processi di calcolo intensivo.

Inoltre con questa libreria non ci si deve preoccupare della scelta di codifica e decodifica dei dati, ErlPort infatti utilizza il protocollo di serializzazione di Erlang, noto come External Term Format (ETF), per la codifica e la decodifica dei dati scambiati tra Erlang e altri linguaggi di programmazione.

Non si entra in dettaglio della libreria, però questo è uno dei modi vantaggiosi per comunicare con python, e in futuri usi tale libreria è da tenere in considerazione per processare dati non convenienti per la VM beam.

Capitolo 3

Performance - Test sperimentali

3.1 Introduzione

In questo capitolo si vuole capire quali potenzialità la piattaforma di Erlang/Elixir porta, in modo da poter capire quali siano le applicazioni possibili con questa tecnologia.

Vogliamo capire i casi d'uso di questa tecnologia, attraverso una serie di test empirici e provare a fare qualche confronto con altri linguaggi.

3.1.1 Hardware utilizzato

I test vengono eseguiti sul sistema operativo linux, viene riportato in figura 3.1 l'output del tool **neofetch** che riporta le caratteristiche del computer utilizzato per l'esecuzione dei test, in particolare notiamo che la CPU di riferimento dei test ha 8 core logici e 4 fisici, e di default la VM alloca 8 scheduler, uno per ogni core logico.

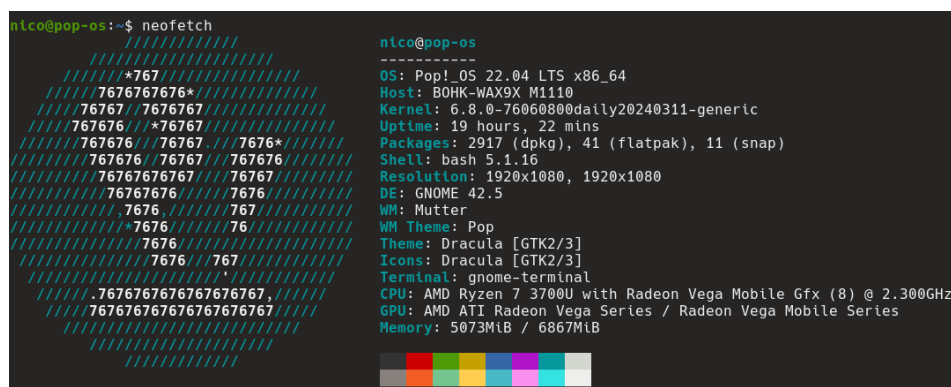


Figura 3.1: Hardware computer utilizzato

3.2 Test concorrentiale

Questo test segue dal lavoro effettuato dalla mia collega Luisa Fausta D'Epiro, è una rivisitazione dei risultati ottenuti con qualche modifica e reinterpretazione dei risultati.[14],

In questo Test empirico si vuole vedere come Elixir si comporta all'aumentare dei processi, per capire se lo scheduling rappresenta un collo di bottiglia all'aumentare dei processi utilizzati.

In particolare si sono scelti una lista di campioni di processi (*processes*) da utilizzare per ogni test, per ogni valore della lista *processes* si eseguono una serie prodotti che vanno da 500 prodotti a 100000 prodotti con uno step di 500.

Si calcola quindi il tempo impiegato per ogni campione e si stampa il campione sul un file csv da poter analizzare ed interpretare successivamente con Matlab.

I campioni di processi da creare per ogni test sono riportati nella lista:

```
processes = [1,2,3,4,5,6,7,8,16,32,64,128,256]
```

3.2.1 Implementazione del Test

Il codice Elixir che esegue il test è implementato nel modulo Elixir ConcurrentTask e riportato nel Listing 3.1.

```
1 defmodule ConcurrentTask do
2   require Logger
3   import MyFile
4
5
6   def compute_products(productsnumber) do
7     for _i <- 1..productsnumber do
8       1 * 1000
9     end
10  end
11
12
13  def run do
14    processes = [1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64, 128, 256]
15    productsnumber = 100_000
16    step = 500
17
18    for proc <- processes do
19      Logger.info("processes #{proc} and #{System.schedulers} scheduler")
20      for comp <- 500..productsnumber//step do
21        {:ok, _time} = parallel_operations(comp, proc)
22      end
23    end
24  end
25
26  def parallel_operations(productsnumber, processnumber) do
```

```

27
28     #divide the products number to assign to each process
29     temp = trunc(productsnumber / processnumber)
30     # compute the rest to compute to restTask
31     rest = rem(productsnumber , processnumber)
32
33     {time, _result} =
34     :timer.tc(
35     fn ->
36         tasks =
37         for _i <- 1..processnumber do
38             Task.async(fn -> compute_products(temp) end)
39         end
40         restTask = Task.async(fn -> compute_products(rest) end)
41
42         # Aspetta di finire ogni task per ottenere i risultati
43         for task <- tasks do
44             Task.await(task, :infinity)
45         end
46         Task.await(restTask, :infinity)
47     end,
48     [],
49     :microsecond
50     )
51     writeData2File(time, processnumber, productsnumber)
52     {:ok, time}
53 end
54
55 def writeData2File(time, processnumber, productsnumber) do
56     available_scheduler =
57     :erlang.system_info(:logical_processors_available)
58
59     scheduler = System.schedulers()
60
61     data = [
62         "#{scheduler}",
63         "#{available_scheduler}",
64         "#{time}",
65         "#{processnumber}",
66         "#{productsnumber}\n"
67     ]
68
69     # scrittura risultato su file
70     write(data)
71     {:ok, time}
72 end
73 end

```

Listing 3.1: Test concorrentiali

La funzione `write` che si occupa della scrittura dei dati su file si trova nel modulo `MyFile` riportato nel Listing 3.2.

```

1 defmodule MyFile do
2   def write(data, file_path \\ "./File/test.csv") do
3     {:ok, file} = File.open(file_path, [:write, :append])
4     IO.write(file, data)
5     File.close(file)
6   end
7 end

```

Listing 3.2: Modulo MyFile

Il modulo Task utilizzato fornisce un modo per eseguire una funzione in background e recuperarne il valore restituito in un secondo momento. Il modulo Task fornito è implementato utilizzando la primitiva `spawn_link` discussa in precedenza.

3.2.2 Esecuzione Test

Si è creato uno script per l'ambiente interattivo iex per l'avvio della funzione voluta riportato nel Listing 3.9, viene compilato il modulo `concurrentTask` ed eseguita la funzione `run()`.

```

:code.purge(ConcurrentTask)
:code.delete(ConcurrentTask)
c("lib/concurrent_task.ex")
ConcurrentTask.run
System.halt

```

Listing 3.3: Script iex per l'avvio dei test

Il test fornito è stato eseguito più volte aumentando gli scheduler allocati all'avvio dell'istanza della VM. Si è scritto un semplice script in bash per eseguire i vari test all'aumentare degli scheduler allocati, lo script è riportato nel Listing 3.10. La macchina virtuale di default viene allocata con 8 scheduler, si avvia con il numero di scheduler voluti impostando il flag `-erl "+S1"`, per l'avvio con più di 8 scheduler si deve forzare la macchina virtuale a farlo con il flag `"+sbt db"`.

```

#!/bin/bash

iex --erl "+S 1" --dot-iex "runTest.iex" -S mix
iex --erl "+S 2" --dot-iex "runTest.iex" -S mix
iex --erl "+S 3" --dot-iex "runTest.iex" -S mix
iex --erl "+S 4" --dot-iex "runTest.iex" -S mix
iex --erl "+S 5" --dot-iex "runTest.iex" -S mix
iex --erl "+S 6" --dot-iex "runTest.iex" -S mix
iex --erl "+S 7" --dot-iex "runTest.iex" -S mix
iex --erl "+S 8" --dot-iex "runTest.iex" -S mix
iex --erl "+S 9 +sbt db" --dot-iex "runTest.iex" -S mix
iex --erl "+S 10 +sbt db" --dot-iex "runTest.iex" -S mix
iex --erl "+S 11 +sbt db" --dot-iex "runTest.iex" -S mix
iex --erl "+S 12 +sbt db" --dot-iex "runTest.iex" -S mix

```

```
iex --erl "+S 13 +sbt db" --dot-iex "runTest.iex" -S mix
iex --erl "+S 14 +sbt db" --dot-iex "runTest.iex" -S mix
iex --erl "+S 15 +sbt db" --dot-iex "runTest.iex" -S mix
iex --erl "+S 16 +sbt db" --dot-iex "runTest.iex" -S mix
```

Listing 3.4: Script bash per l'avvio dei test

Per avviare lo script eseguire i comandi:

```
# solo la prima volta alla creazione del file
chmod +x <directory-path>/runtest.sh

.<directory-path>/runtest.sh
```

Una volta avviati i test verrà creato un file .csv da analizzare con Matlab, i valori presenti nel file assumono la forma:

```
N_Scheduler,N_Available_Scheduler,Time,N_Processes,N_Products
1,8,39,1,500
1,8,24,1,1000
1,8,19,1,1500
1,8,20,1,2000
1,8,22,1,2500
.....
```

Listing 3.5: File csv

3.2.3 Analisi Matlab

In Matlab viene analizzato il file .csv risultante dal test, stampando un grafico per ogni numero di scheduler utilizzato. Lo script Matlab è riportato nel Listing 3.11.

```
opts = detectImportOptions('<replace-with-filecsv-path>');
opts.DataLine = 2;
data = readtable('<replace-with-filecsv-path>', opts);

colors = [
    255 0 0 % Red
    0 255 0 % Green
    0 0 255 % Blue
    255 255 0 % Yellow
    0 255 255 % Cyan
    255 0 255 % Magenta
    128 128 128 % Gray
    255 128 0 % Orange
    128 0 128 % Purple
    0 128 128 % Teal
    128 128 0 % Olive
    255 165 0 % Orange (Web Color)
    0 255 127 % Spring Green
];
colors = colors / 255;
processes = [1,2,3,4,5,6,7,8,16,32,64,128,256];

for n = 1:16
```

```

figure;

for i = 1:length(processes)

    num_processes = processes(i);

    % Filtra i dati per N_Processes = 1 e N_Products = 1
    filteredData = ...
        data(data.N_Processes == num_processes ...
            & data.N_Scheduler==n,:);

    filteredTime = ...
        (filteredData.Time ./filteredData.N_Products);
    plot(filteredData.N_Products, filteredTime,...
        'Color', colors(i, :));

    hold on

end
xlabel('N_Products');
ylabel('Time/Products');

title('Grafico per n. Scheduler: ',n);

legend('1 process','2 processes','3 processes','4 processes', ...
    '5 processes','6 processes','7 processes','8 processes', ...
    '16 processes','32 processes','64 processes',...
    '128 processes','256 processes');

end

```

Listing 3.6: Codice Matlab per la stampa dei grafici

Lo script Matlab stampa 16 grafici, ne vengono riportati tre, per l'esecuzione con 1 scheduler in figura 3.2, con 4 scheduler in figura 3.3 per 8 scheduler in figura 3.4 e per 16 in figura 3.5.

I test risultano avere degli andamenti simili, possiamo notare che con l'effettuazione di pochi prodotti, aumentare il numero dei processi porta ad un degradamento delle performance, in quanto la VM si occupa più di schedulare i processi che a fare prodotti, uno zoom di questo andamento è riportato in figura 3.6. Per pochi prodotti come previsto lo scheduling fa perdere efficienza alle operazioni svolte. Però si nota che l'andamento con pochi processi lo scheduling non porta ad una mancanza di efficienza nonostante le operazioni da svolgere prendono pochissimo tempo.

Facendo uno zoom per un numero di prodotti più importante, come in figura 3.7, si nota che con un processo le operazioni da svolgere risultano più lente, e già con più processi la parallelizzazione inizia ad avere più senso.

Con un solo scheduler invece l'andamento con un processo risulta simile a quello con più processi, ma il tempo impiegato con 8 scheduler risulta in generale inferiore rispetto all'impiego di un solo scheduler, questo perchè Erlang riesce a parallelizzare il numero di prodotti da effettuare su più processi.

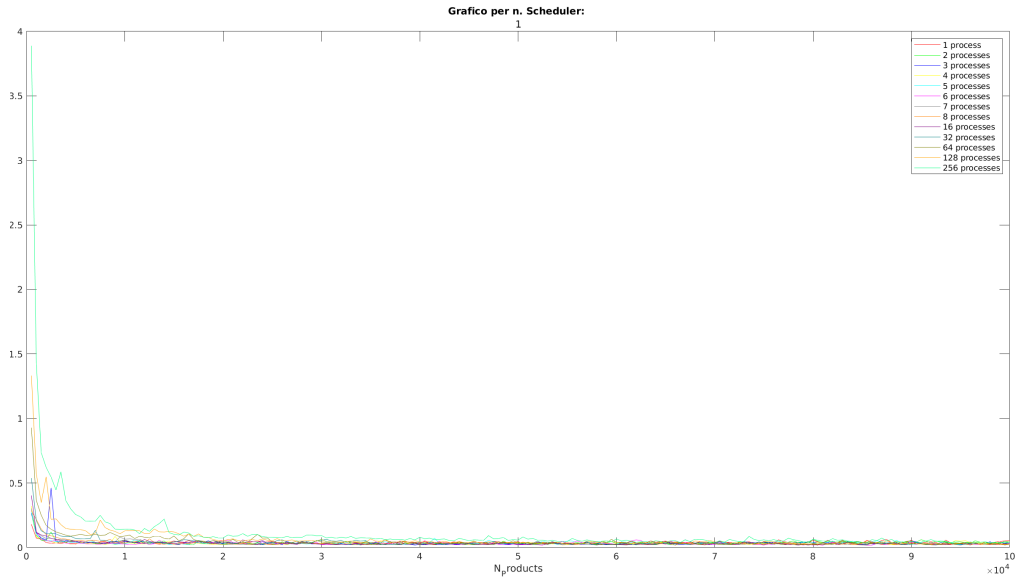


Figura 3.2: Grafico con 1 scheduler

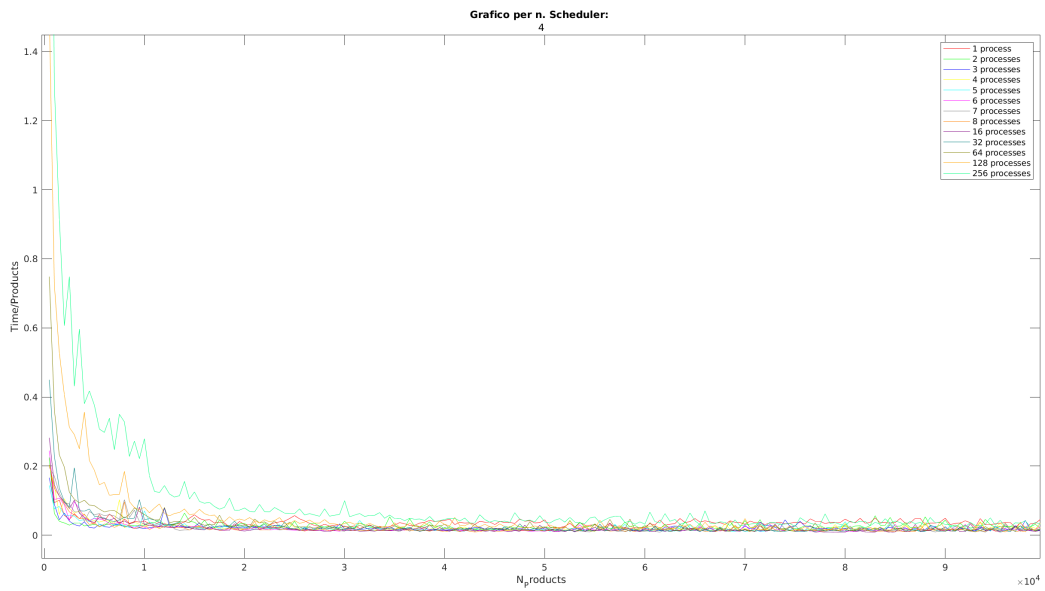


Figura 3.3: Grafico con 4 scheduler

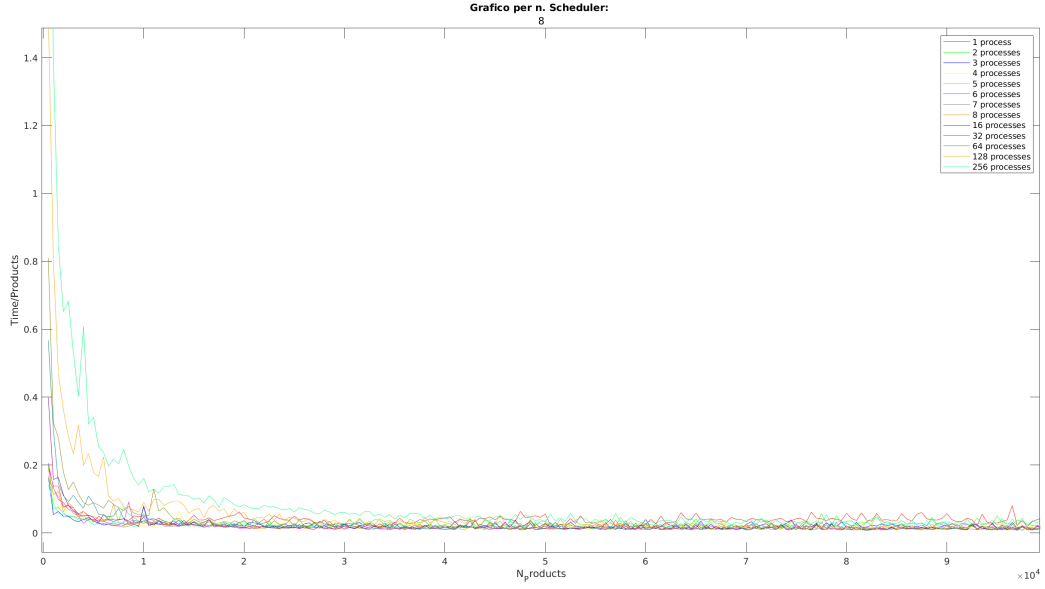


Figura 3.4: Grafico con 8 scheduler

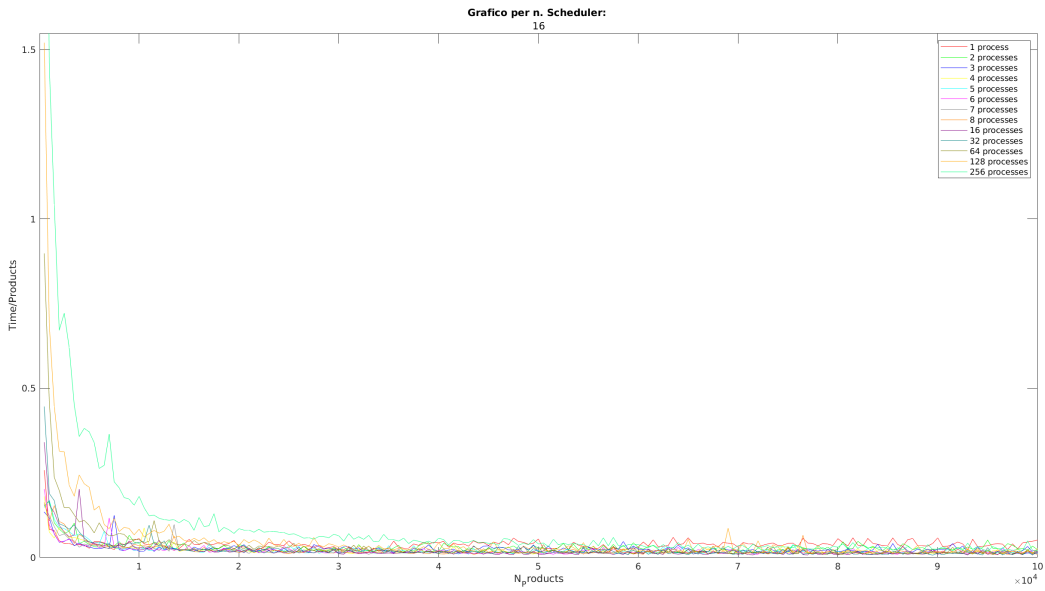


Figura 3.5: Grafico con 16 scheduler

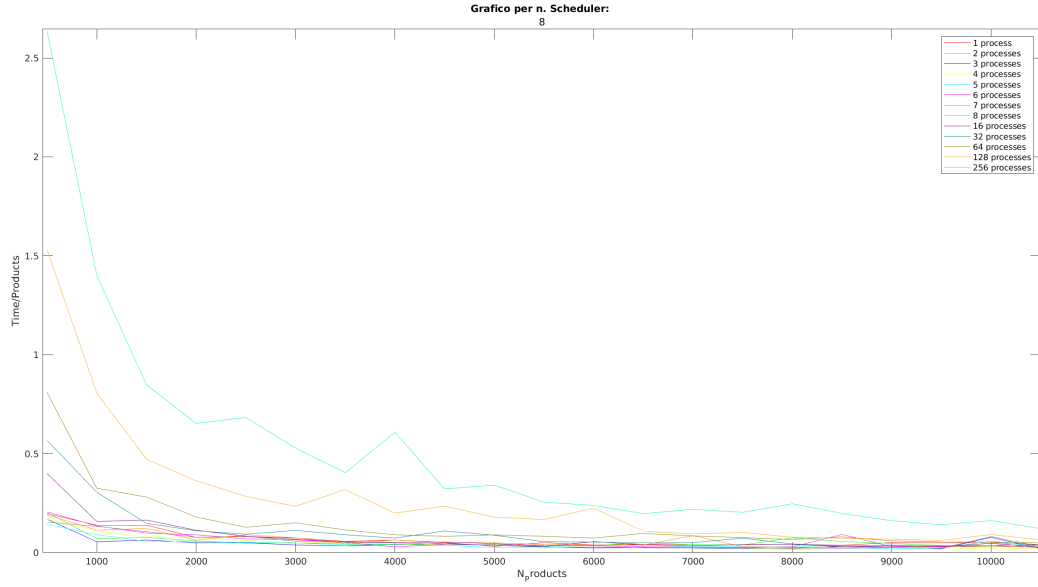


Figura 3.6: Zoom con 8 scheduler, range 500-10000 prodotti

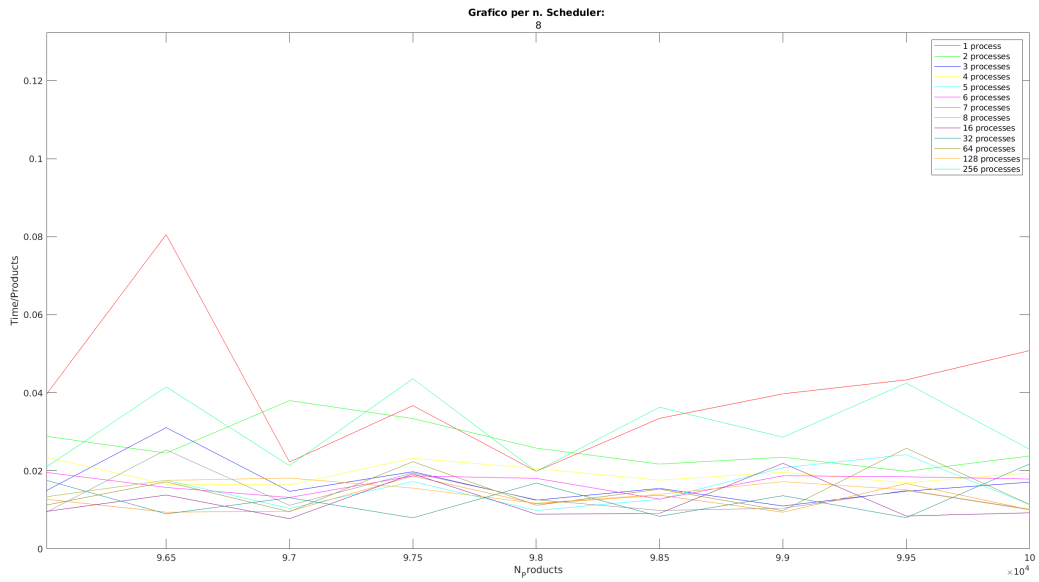


Figura 3.7: Zoom con 8 scheduler, range 95000-100000 prodotti

3.3 Test concorrentiale con IO

Nel seguente test si vuole testare la concorrenza come nel test del paragrafo 3.2 ma invece di variare il numero di prodotti per ogni processo, si varia il numero di operazioni di Input ed Output su File. Queste operazioni impiegano più tempo rispetto ai prodotti, quindi ci si aspetta di mettere maggiormente in evidenza il guadagno che si ha nell'usare più processi rispetto all'utilizzarne uno solo.

3.3.1 Implementazione del test

Il codice Elixir che esegue il test è implementato nel modulo Elixir TaskIO e riportato nel Listing 3.7.

```
1
2 defmodule TaskIO do
3   require Logger
4
5   import MyFile
6
7   # decodifica del file Json
8   def parse_json_file(file_path \\ "./File/read.json") do
9     case File.read(file_path) do
10      {:ok, json_data} ->
11        case Poison.decode(json_data) do
12          {:ok, parsed_json} ->
13            parsed_json
14
15          {:error, reason} ->
16            IO.puts("Failed to parse JSON: #{reason}")
17            nil
18          end
19
20          {:error, reason} ->
21            IO.puts("Failed to read file: #{reason}")
22            nil
23          end
24      end
25
26      # caso base ricorsione
27      def compute_products(0, _file) do
28        nil
29      end
30
31      def compute_products(operations, file) do
32        data = parse_json_file()
33
34        # calcolo risultati del json
35        somma = data["somma"] |> Enum.reduce(fn x, acc -> acc + x end)
36
37        sottrazione = data["sottrazione"]
38        |> Enum.reduce(fn x, acc -> acc - x end)
39
40        moltiplicazione = data["moltiplicazione"]
41        |> Enum.reduce(fn x, acc -> acc * x end)
```

```

42
43     divisione = data["divisione"]
44     |> Enum.reduce(fn x, acc -> acc / x end)
45
46     # scrittura risultati del nel file csv
47     result = [
48       "#{somma}",",
49       "#{sottrazione}",",
50       "#{moltiplicazione}",",
51       "#{divisione}\n"
52     ]
53
54     IO.write(file, result)
55
56     # chiamata ricorsiva
57     compute_products(operations - 1, file)
58 end
59
60 def run do
61   processes = [1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64, 128, 256, 512]
62   operationsnumber = 20_000
63   step = 250
64
65   {:ok, file} = File.open("./File/write.csv", [:write, :append])
66
67   for proc <- processes do
68     for comp <- 500..operationsnumber//step do
69       {:ok, _time} = parallel_operations(comp, proc, file)
70     end
71   end
72
73   File.close(file)
74 end
75
76 def parallel_operations(operationsnumber, processnumber, file) do
77
78   # calcolo del numero di operazioni da dare ad ogni processo
79   temp = trunc(operationsnumber / processnumber)
80
81   # calcolo del resto
82   rest = rem(operationsnumber, processnumber)
83
84   {time, _result} =
85     :timer.tc(
86       fn ->
87         tasks =
88           for _i <- 1..processnumber do
89             Task.async(fn -> compute_products(temp, file) end)
90           end
91
92       restTask = Task.async(fn -> compute_products(rest, file) end)
93
94       # Per ogni task aspetta di finire
95       for task <- tasks do
96         Task.await(task, :infinity)
97       end
98
99       Task.await(restTask, :infinity)
100     end,
101     [],
102     :microsecond
103   )

```

```

104
105     writeData2File(time, processnumber, operationsnumber)
106     {:ok, time}
107 end
108
109 def writeData2File(time, processnumber, productsnumber) do
110     available_scheduler = :erlang.system_info(
111         :logical_processors_available)
112
113     scheduler_online = System.schedulers()
114
115     data = [
116         "#{scheduler_online}",",
117         "#{available_scheduler}",",
118         "#{time}",",
119         "#{processnumber}",",
120         "#{productsnumber}",",
121     ]
122
123     # scrittura risultato su file
124     write(data, "./File/testI0.csv")
125     {:ok, time}
126 end
127 end

```

Listing 3.7: Test concorrentiali

Anche qui viene utilizzato il modulo Task che fornisce un modo per eseguire una funzione in background e recuperarne il valore restituito in un secondo momento.

Le operazioni che vengono effettuate sono la lettura del seguente file Json:

```

{
  "somma" : [40,10,20],
  "sottrazione": [10023, 1000, 200],
  "moltiplicazione": [10, 10, 10],
  "divisione": [72000,1000, 2]
}

```

Per ogni attributo viene calcolato il risultato della lista corrispondente, e stampato su un file CSV. La decodifica del file Json viene effettuata tramite la libreria **Poison**[17] che si effettua nella funzione `parse_json_file()` del Listing 3.7.

```

60,8823,1000,3.60
60,8823,1000,3.60
60,8823,1000,3.60
....

```

Listing 3.8: Output risultati Json nel csv

Come nel precedente test, per ogni n operazioni effettuate e per m processi viene riportato il risultato in un file .csv uguale al precedente test (Listing 3.2), il file è da analizzare con Matlab.

3.3.2 Esecuzione Test

Si è creato uno script per l'ambiente interattivo iex per l'avvio della funzione voluta riportato nel Listing 3.9

```
:code.purge(TaskIO)
:code.delete(TaskIO)
c("lib/taskIO.ex")
TaskIO.run
System.halt
```

Listing 3.9: Script iex per l'avvio dei test "runTestIO.iex"

Il test fornito è stato eseguito più volte aumentando gli scheduler allocati all'avvio dell'istanza della VM. Si è scritto un semplice script in bash per eseguire i vari test all'aumentare degli scheduler allocati, lo script è riportato nel Listing 3.10.

```
#!/bin/bash

iex --erl "+S 1" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 2" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 3" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 4" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 5" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 6" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 7" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 8" --dot-iex "runrunTestIOTest.iex" -S mix
iex --erl "+S 9 +sbt db" --dot-iex "runrunTestIOTest.iex" -S mix
iex --erl "+S 10 +sbt db" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 11 +sbt db" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 12 +sbt db" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 13 +sbt db" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 14 +sbt db" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 15 +sbt db" --dot-iex "runTestIO.iex" -S mix
iex --erl "+S 16 +sbt db" --dot-iex "runTestIO.iex" -S mix
```

Listing 3.10: Script bash per l'avvio dei test

Per avviare lo script eseguire i comandi:

```
# solo la prima volta alla creazione del file
chmod +x <directory-path>/runtest.sh

.<directory-path>/runtest.sh
```

3.3.3 Analisi Matlab

In Matlab viene analizzato il file .csv risultante dal test, stampando un grafico per ogni numero di scheduler utilizzato. Lo script Matlab è riportato nel Listing 3.11.

```

opts = detectImportOptions('<path-file>');
opts.DataLine = 2;
data = readtable('<path-file>', opts);

colors = [
    255 0 0 % Red
    0 255 0 % Green
    0 0 255 % Blue
    255 255 0 % Yellow
    0 255 255 % Cyan
    255 0 255 % Magenta
    128 128 128 % Gray
    255 128 0 % Orange
    128 0 128 % Purple
    0 128 128 % Teal
    128 128 0 % Olive
    255 165 0 % Orange (Web Color)
    0 255 127 % Spring Green % da togliere
    218 112 214 % Orchid
    70 130 180 % Steel Blue
];
colors = colors / 255;
processes = [1,2,3,4,5,6,7,8,16,32,64,128,256,512];

for n = 1:16

    figure;

    for i = 1:length(processes)
        num_processes = processes(i);

        % Filtra i dati per il processo da disegnare
        % e per scheduler utilizzati
        filteredData = data(data.N_Processes == num_processes...
            & data.N_Scheduler==n,:);

        filteredTime = ...
            (filteredData.Time ./filteredData.N_Products);
        plot(filteredData.N_Products, filteredTime, 'Color', colors(i, :));

        hold on
    end

    xlabel('N. Operazioni IO');
    ylabel('Time/operazione(microsec)');

    title('Grafico per n. Scheduler: ',n);

    legend('1 process','2 processes','3 processes','4 processes', ...
        '5 processes','6 processes','7 processes','8 processes', ...
        '16 processes','32 processes','64 processes','128 processes',...
        '256 processes');

end

```

Listing 3.11: Analisi dei processi in Matlab

Lo script Matlab stampa 16 grafici, ne vengono riportati quattro, rispettivamente con numero di scheduler pari a 1 riportato in figura 3.8, con 4

riportato in figura 3.9, con 8 riportato in figura 3.10, e con 16 riportato in figura 3.11.

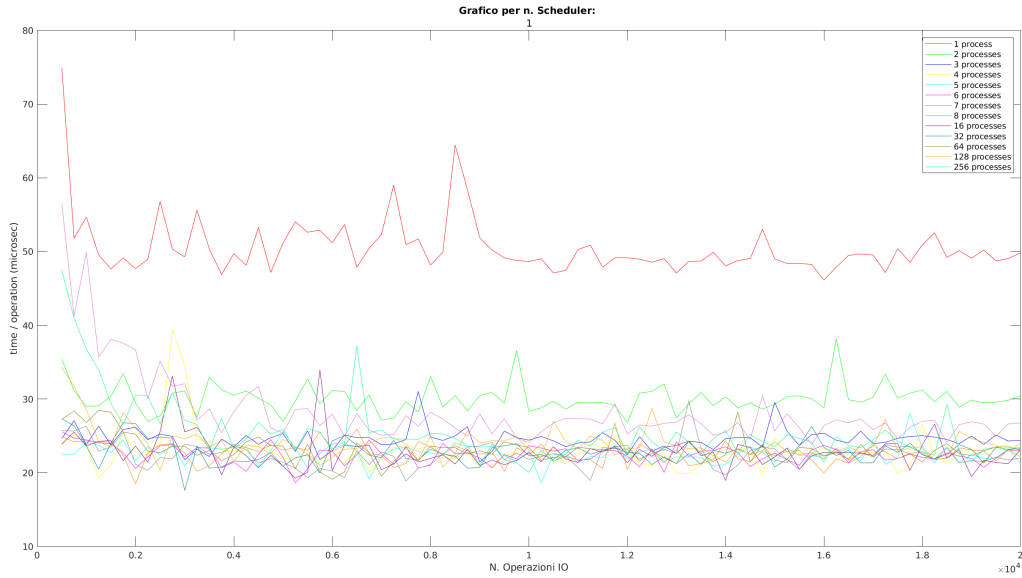


Figura 3.8: Grafico con 1 scheduler

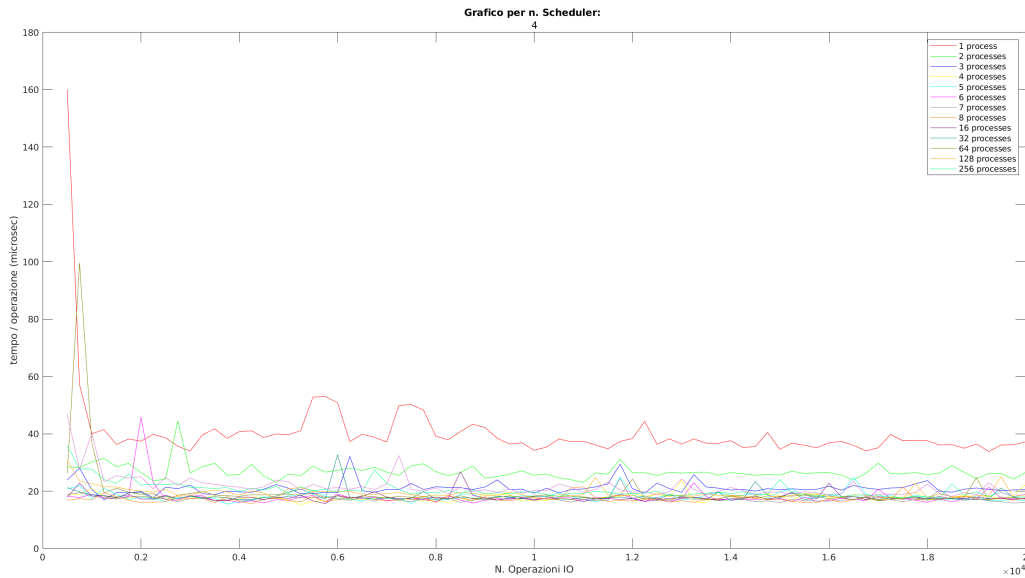


Figura 3.9: Grafico con 4 scheduler

I test risultano avere andamenti molto simili, quello che risulta avere un andamento più basso e stabile è quello con 4 scheduler.

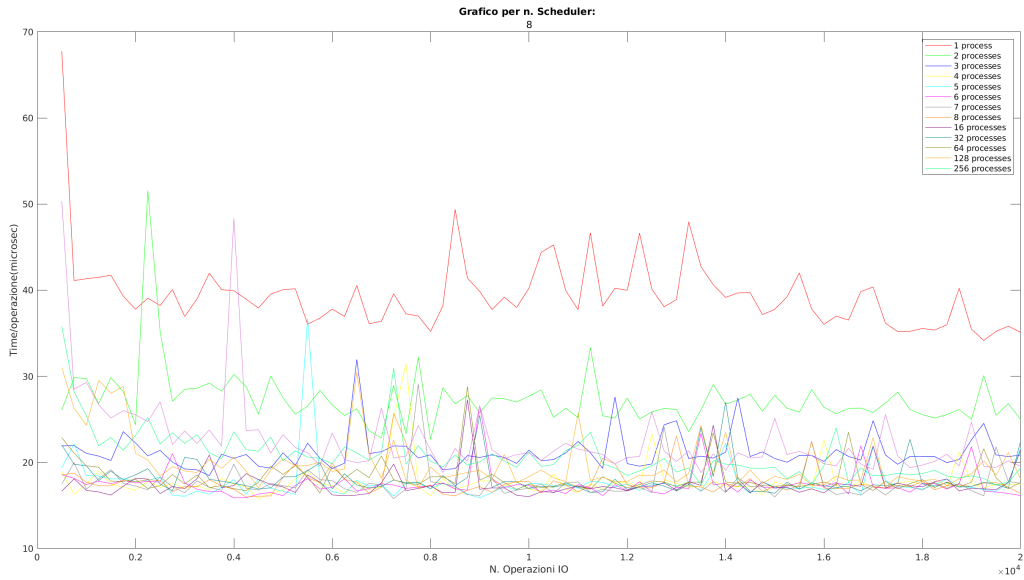


Figura 3.10: Grafico con 8 scheduler

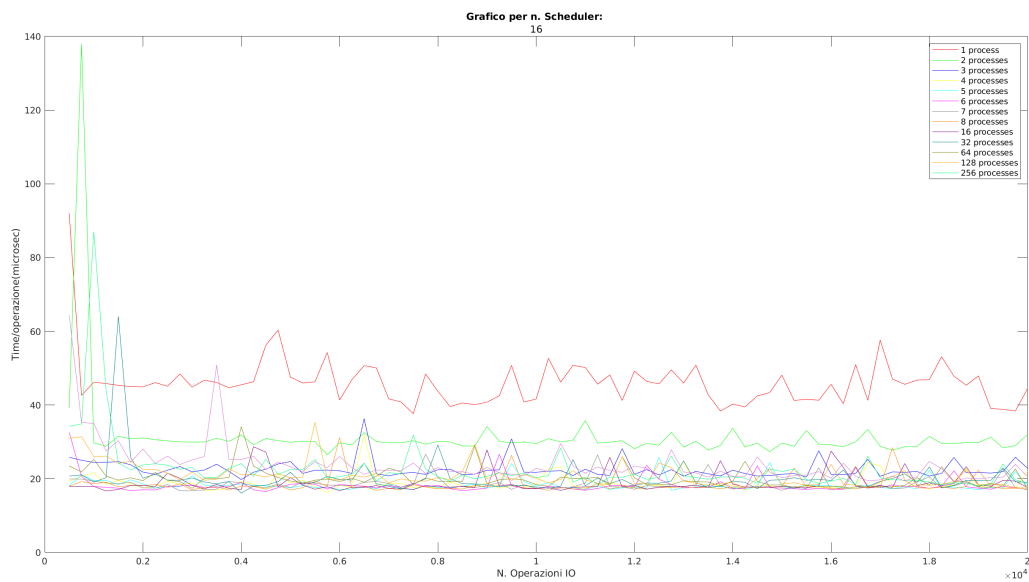


Figura 3.11: Grafico con 16 scheduler

Quello che si evidenzia è che in ogni grafico, l'andamento risultante da un solo processo risulta avere un andamento alto circa il doppio rispetto agli altri processi, questo è in linea con ciò che ci si aspettava, infatti con più processi la VM cerca di parallelizzare i processi. Con un solo scheduler ci si potrebbe aspettare che le operazioni con un solo processo risultano veloci quanto quelle con più processi, ma la CPU mentre aspetta che un operazione di Input e Output venga completata può eseguire altre istruzioni, per questo anche con un solo scheduler risulta vantaggioso utilizzare la computazione concorrentiale, il collo di bottiglia di questo test potrebbero essere le operazioni su File e non l'utilizzo, di Cpu, sarebbe da approfondire tale questione, una cosa che si può fare in futuri sviluppi è provare ad assegnare l'operazioni di Input ed Output ad un GenServer che si prende la responsabilità di scrivere su File i risultati a blocchi, mantenendo i risultati su uno stato che si svuota ogni tot operazioni, questo può portare ad un maggiore utilizzo della CPU dando la responsabilità delle operazioni di IO ad un altro processo.

Un altro punto da evidenziare è che ci sono dei picchi negli andamenti, questo può essere dovuto dalle operazioni del Garbage Collector che avviene durante l'esecuzione delle operazioni.

Guadagno dei processi

Risulta utile in questo caso fare un grafico del guadagno dei processi rispetto all'utilizzarne uno solo. Nel Listing 3.12 è riportato lo script Matlab che disegna i grafici del guadagno dei processi rispetto all'utilizzarne uno solo. Un guadagno in percentuale negativo risulta in una maggiore efficienza rispetto ad utilizzare un solo processo, un guadagno positivo risulta un degrado delle performance.

```
opts = detectImportOptions('<path-file>');
opts.DataLine = 2;
data = readtable('<path-file>', opts);

range = 500:250:20000;
processes = [2,3,4,5,6,7,8,16,32,64,128,256,512];
colors = [
    255 0 0 % Red
    0 255 0 % Green
    0 0 255 % Blue
    255 255 0 % Yellow
    0 255 255 % Cyan
    255 0 255 % Magenta
    128 128 128 % Gray
    255 128 0 % Orange
    128 0 128 % Purple
    0 128 128 % Teal
    128 128 0 % Olive
    255 165 0 % Orange (Web Color)
    0 255 127 % Spring Green % da togliere
```

```

];

colors = colors / 255;

for n = 1:16 % Scheduler
    Data1proc = data(data.N_Processes == 1 & data.N_Scheduler == n, :);
    Data1procfiltered = Data1proc.Time ./ Data1proc.N_Products;
    figure
    xlabel('N_Products');
    ylabel('Time/Products');

    for k = 1:length(processes)
        colore = colors(k,:);
        datifinali = zeros(size(range));

        %filtraggio dati
        Data2proc = data(data.N_Processes == processes(k)...
                        & data.N_Scheduler == n, :);

        Data2procfiltered = Data2proc.Time ./ Data2proc.N_Products;

        % Calcola i dati finali per ciascun valore di N_Products
        datifinali = ((Data2procfiltered - Data1procfiltered)...
                    ./Data1procfiltered)*100;

        % Traccia i dati finali per il processo corrente
        plot(range, datifinali, 'Color', colore);
        hold on;
    end

    title('Guadagno per num scheduler', n);
    legend_entries = arrayfun(@(x) sprintf('Processi %d', x),...
                             processes, 'UniformOutput', false);
    legend(legend_entries);
end

```

Listing 3.12: Guadagno rispetto ad un processo

Il codice Matlab stampa molteplici grafici, uno per ogni scheduler utilizzato, viene riportato in figura 3.12 il Grafico che fa riferimento a 4 scheduler, che mostra che accedendo ai file concorrentialmente apporta un miglioramento delle performance, consentendo di sfruttare più CPU disponibile. In particolare il numero di processi che porta più vantaggio di quelli testati è quello più alto pari a 512 che migliora le performance in modo significativo, apportando un guadagno delle performance medio di -47.8% In figura 3.13 viene riportato il guadagno in percentuale di 512 processi rispetto all'utilizzarne uno solo.

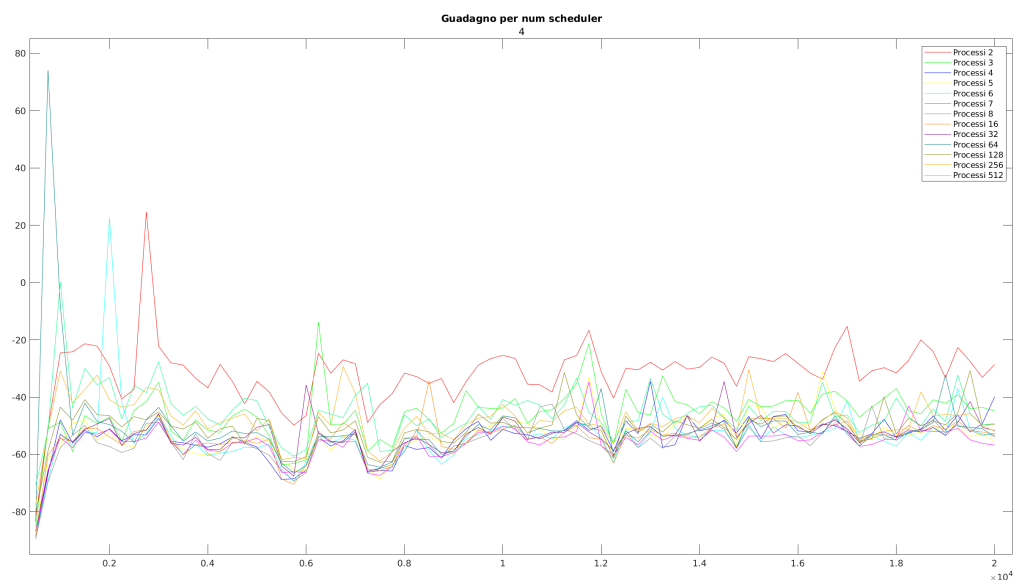


Figura 3.12: Guadagno rispetto a 1 processo con 4 scheduler

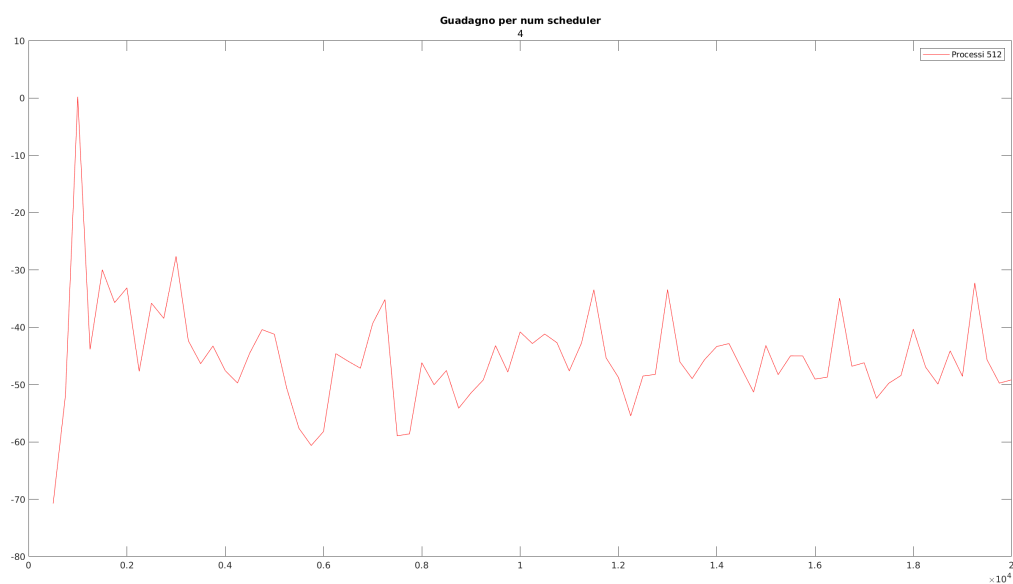


Figura 3.13: Guadagno di 512 processi rispetto a 1 processo con 4 scheduler

3.4 Test interoperabilità con C/C++

In questo test empirico viene utilizzata la libreria *Benchee* che ci permette di valutare le performance di esecuzione di funzioni, in particolare le funzioni da eseguire saranno due funzioni scritte in Elixir, ed altre due con il tramite i due meccanismi di interoperabilità discussi nel paragrafo 2.7, il meccanismo con Port e con le NIF.

3.4.1 Definizione problema da risolvere

Il problema che si vuole risolvere è il problema definito nell'equazione 3.1.

$$[n_1, n_2 \dots n_m] \rightarrow \left[\sum_{i=1}^{n_1} i, \sum_{i=1}^{n_2} i, \dots, \sum_{i=1}^{n_m} i \right] \rightarrow [sum_1, sum_2, \dots, sum_m] \quad (3.1)$$

Le funzioni implementate prendono come input una lista di interi, e si vuole restituire una lista che contiene i risultati dell'equazione 3.1.

L'input di riferimento è il seguente

$$listInput = [1000000, 2000000, 5000000] \quad (3.2)$$

Non si sono scelti numeri troppo piccoli per evidenziare al meglio le differenze tra le implementazioni.

Le funzioni si occupano di calcolare il risultato della somma dei primi n_i elementi restituendo il risultato m_i .

Le funzioni sono implementate con 4 strategie differenti:

1. **list_sum_recursive()**: Funzione implementata in elixir tramite ricorsione.
2. **list_sum_recursive_tail()**: Funzione implementata in elixir tramite ricorsione ottimizzata, dove l'ultima istruzione è la chiamata ricorsiva, in modo che elixir ottimizza la ricorsione con la tail optimization.
3. **list_sum_port()**: Funzione implementata in C++, Elixir interagisce con il codice tramite il meccanismo IPC Port.
4. **list_sum_iterative_nif()**: Funzione implementata in C++, Elixir interagisce con il codice C++ tramite il meccanismo NIF.

Tutte le funzioni elixir che seguiranno in questo paragrafo vengono messe dentro il modulo *SpeedSum* come nel Listing 3.13:

```
1 defmodule Speedsum do
2   ...
3   # Funzioni implementate
4   ...
5 end
```

Listing 3.13: Modulo di riferimento

3.4.2 Funzione ricorsiva

Nel Listing 3.14 è riportata l'implementazione della funzione ricorsiva che risolve il problema definito sopra.

```
1 def list_sum_recursive(list) when is_list(list) do
2   Enum.map(list, fn x -> sum_recursive(x) end)
3 end
4
5 #caso base
6 defp sum_recursive(0), do: 0
7
8 defp sum_recursive(n), do: n + sum_recursive(n - 1)
```

Listing 3.14: Funzione `list_sum_recursive()`

3.4.3 Funzione ricorsiva ottimizzata

Nel Listing 3.15 è riportata l'implementazione della funzione ricorsiva ottimizzata con il metodo della Tail recursion, Elixir come vedremo dai risultati del Benchmark risolve il problema definito in modo molto più veloce. Come si può vedere dal codice implementato, Elixir ottimizza le chiamate ricorsive quando l'ultima istruzione è il passo ricorsivo. Per far sì che il passo ricorsivo sia l'ultima istruzione della funzione, è necessario passare lo stato della somma come parametro della funzione.

```
1 # Funzione di somma ricorsiva ottimizzata
2 def list_sum_recursive_tail(list) when is_list(list) do
3   Enum.map(list, fn x -> sum_recursive_tail(x, 0) end)
4 end
5
6 #caso base
7 defp sum_recursive_tail(0, acc), do: acc
8
9 defp sum_recursive_tail(n, acc) when n > 0 do
10   sum_recursive_tail(n - 1, acc + n)
11 end
```

Listing 3.15: Funzione `list_sum_recursive_tail()`

3.4.4 Funzione in C++ con Port

Questa funzione è implementata con l'Inter Process Communication che Erlang ed Elixir mettono a disposizione, ovvero il metodo Port già discusso in precedenza. Con Port si può comunicare con un processo esterno tramite file descriptor, nel nostro caso è stata implementata la comunicazione usando lo standard input e lo standard output. Per la comunicazione si deve scegliere un formato di codifica e decodifica, per non complicare le cose si è scelto si è scelta una comunicazione line by line, dove la lista viene inviata da Elixir nel formato " n_1, n_2, \dots, n_m " scegliendo come carattere delimitatore dei valori della lista il carattere ",".

Il codice C++ che risolve il nostro problema è riportato nel Listings 3.16

```

1  #include <algorithm>
2  #include <iostream>
3  #include <sstream>
4  #include <string>
5  #include <vector>
6
7  //Funzione che prende in input una stringa
8  //di interi separati da un delimitatore e
9  //restituisce un vector di interi
10 std::vector<long int> tokenizeToIntVector(const std::string &str,
11                                           char delimiter) {
12     std::vector<long int> tokens;
13     std::stringstream ss(str);
14     std::string token;
15     while (std::getline(ss, token, delimiter)) {
16         tokens.push_back(
17             std::stol(token));
18     }
19     return tokens;
20 }
21
22 int main() {
23     std::string line;
24     // Ascolta lo standard input line by line
25     while (std::getline(std::cin, line)) {
26
27         char delimiter = ',';
28
29         std::vector<long> numbers = tokenizeToIntVector(line, delimiter);
30         std::vector<long> results;
31
32         for (auto &n : numbers) {
33             long sum = 0;
34             for (int i = 0; i <= n; i++) {
35                 sum += i;
36             }
37             results.push_back(sum);
38         }
39
40         // Stampa sullo standard output una stringa
41         // nello stesso formato di ricezione
42         for (int i = 0; i < results.size() - 1; i++) {
43             std::cout << results[i] << ",";
44         }

```



```
45     std::cout << results[results.size() - 1] << std::endl;
46 }
47 return 0;
48 }
```

Listing 3.16: Funzione `list_sum_port()`

Si può compilare il programma C++ tramite il compilatore gcc:

```
g++ -o list_sum_port <source-directory>/list_sum_port.cpp
```

Nel Listing 3.19 è riportata la funzione che comunica con il programma C++.

```
1
2 def list_sum_port(list) when is_list(list) do
3
4   # Apertura del Port
5   port = Port.open({:spawn_executable, "./priv/list_sum_port"},
6                     [:binary, :use_stdio])
7
8   # encoding del messaggio da inviare al port
9   message = Enum.join(list, ", ")
10
11  # invio del messaggio al port
12  Port.command(port, "#{message}\n")
13
14  receive do
15    {^port, {:data, result}} ->
16      String.trim(result)
17
18    {^port, {:exit_status, status}} ->
19      IO.puts("Processo port terminato con codice di uscita #{status}")
20  after
21    1000 ->
22      IO.puts("Timeout")
23  end
24  Port.close(port)
25 end
```

Listing 3.17: Funzione `list_sum_port()`

3.4.5 Funzione implementata in C++ tramite NIF

Nel caso del metodo NIF non c'è bisogno di decidere un metodo di codifica e decodifica, ma bisogna affidarsi all'interfaccia che Erlang ci mette a disposizione per far interfacciare il codice C con Elixir, l'interfaccia supporta solo il C, quindi per poter utilizzare codice C++ si devono fare le dovute conversioni. Nel Listing 3.18 è riportato il codice NIF per la risoluzione del problema definito.

```

1  #include "erl_nif.h"
2  #include "string.h"
3  #include <vector>
4
5
6  // funzione per riempire un vector da una lista di Elixir
7  inline bool fillVector(ErlNifEnv* env,
8                        ERL_NIF_TERM listTerm,
9                        std::vector<long int>& result)
10 {
11     unsigned int length = 0;
12     if (!enif_get_list_length(env, listTerm, &length)) {
13         return false;
14     }
15
16     long int actualHead;
17     ERL_NIF_TERM head;
18     ERL_NIF_TERM tail;
19     ERL_NIF_TERM currentList = listTerm;
20
21     // O(n), scorre tutta la lista
22     for (unsigned int i = 0; i < length; ++i) {
23         if (!enif_get_list_cell(env, currentList, &head, &tail)) {
24             return false;
25         }
26         currentList = tail;
27         if (!enif_get_long(env, head, &actualHead)) {
28             return false;
29         }
30         result.push_back(actualHead);
31     }
32     return true;
33 }
34
35 static ERL_NIF_TERM list_sum_iterative_nif(ErlNifEnv* env,
36                                             int argc,
37                                             const ERL_NIF_TERM argv[]) {
38
39     std::vector<long int> a;
40
41     if (!fillVector(env, argv[0], a)) {
42         return enif_make_badarg(env);
43     }
44
45     //creazione dell'array per i risultati da restituire
46     ERL_NIF_TERM results[a.size()];
47
48     for(int i=0; i < a.size(); ++i){
49         long int sum = 0;
50         for (int j = 1; j <= a[i]; j++) {

```

```

51     sum += j;
52 }
53
54 ERL_NIF_TERM temp = enif_make_long(env, sum);
55 results[i] = temp;
56 }
57
58 ERL_NIF_TERM list = enif_make_list_from_array(env, results , a.size());
59
60 return list;
61 }
62
63 // Definizione delle funzioni NIF
64 static ErlNifFunc nif_funcs[] = {
65     {"list_sum_iterative_nif", 1, list_sum_iterative_nif}
66 };
67
68 ERL_NIF_INIT(Elixir.SpeedSum, nif_funcs, NULL, NULL, NULL, NULL)
69

```

Listing 3.18: Funzione NIF

Come si può vedere il NIF implementato è composto da una funzione ausiliaria che si occupa di scorrere la lista passata da Elixir e riempie un `std::vector<long>`.

Dopodiché c'è l'implementazione della funzione NIF `list_sum_iterative_nif()`, e dalla macro `ERL_NIF_INIT` che mette a disposizione la funzione implementata nel nostro modulo Elixir che ne fa uso.

Si può notare che tutti i tipi che devono essere letti dalla VM devono essere degli `ERL_NIF_TERM`, che possono essere letti e scritti utilizzando l'API messa a disposizione dall'installazione di Erlang, che si trova nel file `"erl_nif.h"`.

Per utilizzare la funzione implementata, il codice deve essere compilato come una libreria condivisa, con il compilatore `gcc` si può compilare nel seguente modo:

```

g++ -fPIC -shared -o list_sum_iterative_nif.so \
    list_sum_iterative_nif.cpp -I $ERL_ROOT/usr/include/

```

La directory `$ERL_ROOT` dipende dall'installazione di Erlang, per saperne il valore si può eseguire:

```

elixir -e "IO.puts :code.root_dir()"

```

Per eseguire la funzione esterna implementata, basta definire nel modulo di riferimento il seguente codice:

```

1
2 def load_nif do
3   :ok = :erlang.load_nif(
4     String.to_charlist("priv/list_sum_iterative_nif"),
5     0)
6 end

```

```
7
8 def list_sum_iterative_nif(_n) do
9   :erlang.nif_error("Errore nel caricamento nif")
10 end
```

Listing 3.19: Funzione `list_sum_port()`

3.4.6 Esecuzione Test

Per l'esecuzione del test come precedentemente detto ci si è affidati alla libreria `Benchee` [3].

Per includere la libreria `Benchee`, basta inserire la dipendenza nel file `mix.exs`. La funzione `deps` nel file diventa la seguente:

```
1 defp deps do
2   [
3     {:benchee, "~> 1.0", only: :dev},
4     {:benchee_html, "~> 1.0", only: :dev},
5   ]
6 end
```

Listing 3.20: Dipendenze di `Benchee`

Viene inserito anche il plug-in `html` della libreria per visualizzare il report in formato `html` per una visione più chiara del risultato. Per scaricare le dipendenze basta eseguire il comando:

```
1 mix deps.get
```

Con la libreria `Benchee` sono state testate tutte e quattro le strategie di implementazione usate, ed è stata aggiunta la strategia di formattazione `HTML` per visualizzare i report nel browser web. Nel Listing 3.21 è presente la funzione che lancia il benchmark.

```
1 def speed_test() do
2   list_input = [1_000_000, 2_000_000, 5_000_000]
3   Benchee.run(
4     %{
5       "sum_recursive" => fn -> list_sum_recursive(list_input) end,
6       "sum_recursive_tail" => fn -> list_sum_recursive_tail(list_input) end,
7       "sum_iterative_nif" => fn -> list_sum_iterative_nif(list_input) end,
8       "list_sum_port" => fn -> list_sum_port(list_input) end
9     },
10    warmup: 4,
11    memory_time: 4,
12    formatters: [
13      Benchee.Formatters.HTML,
14      Benchee.Formatters.Console
15    ]
16  end
```

Listing 3.21: Funzione `speed_test()`

La funzione costruisce un report in HTML che riporta la Tabella in figura 3.14

Run Time Comparison [®]

Name	Iterations per Second	Average	Deviation	Median	Mode	Minimum	Maximum	Sample size
sum_iterative_nif	47.29	21.15 ms	±10.51%	20.39 ms	20.36 ms	19.77 ms	34.80 ms	237
list_sum_port	42.53	23.51 ms	±15.03%	22.45 ms	none	19.22 ms	42.84 ms	213
sum_recursive_tail	28.13	35.54 ms	±3.46%	35.19 ms	none	34.82 ms	42.39 ms	141
sum_recursive	8.34	119.93 ms	±78.10%	103.29 ms	none	94.50 ms	709.99 ms	42

Figura 3.14: Report di Benchec

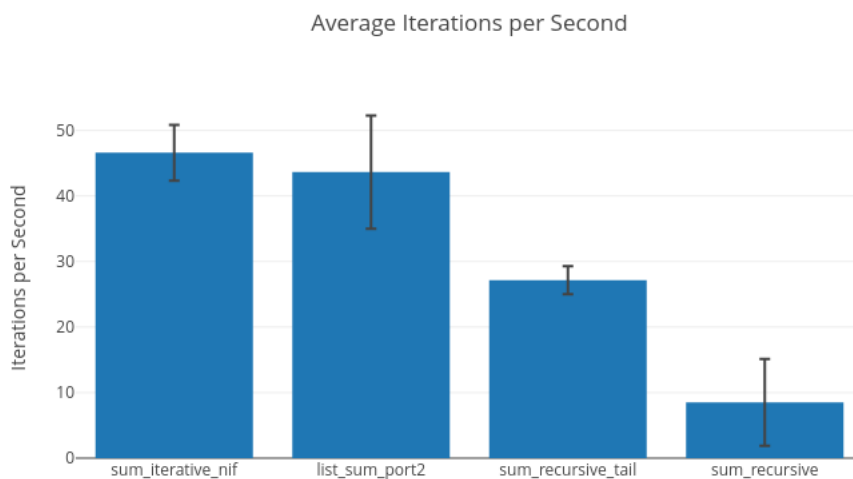


Figura 3.15: Report a barre, iterazioni a secondo

Come possiamo notare le funzioni scritte in C++ risultano più veloci, non ci sono grosse differenze tra il metodo NIF e il metodo con le Port, per molti casi la strategia con le Port dovrebbe risultare la più conveniente.

Un fatto molto curioso sono le funzioni implementate in Elixir, notiamo che la ricorsione ottimizzata è molto più veloce rispetto alla ricorsione classica, questo dà un'idea che per molte chiamate ricorsive l'ottimizzazione con la Tail Recursion sia obbligatoria.

3.5 Test Http Server

Nel seguente benchmark si vuole mettere a confronto un server http scritto in Elixir utilizzando la libreria **Plug**[16] con i server scritti in Python e Node, usando la libreria **Flask**[22] per Python e la libreria **Express**[8] per Node. I benchmark vengono eseguiti sulla stessa macchina con il tool **wrk**, wrk è uno strumento moderno di benchmarking HTTP in grado di generare un carico significativo se eseguito su una CPU multi-core [23].

3.5.1 Implementazione dei Server

Gli http server implementati sono molto semplici, per fare load testing si dovrebbero simulare situazioni di calcolo reale sul server http, in questo caso si vuole solo avere un'idea di come Elixir performa su una risposta breve ad una chiamata GET all'indirizzo localhost:5000/ping, rispondendo con un semplice html con la stringa "pong".

Http Server Elixir

Per utilizzare la libreria Plug basta inserire la dipendenza richiesta nel file mix.exs:

```
{:plug_cowboy, "~> 2.0"}, # http server library
```

Listing 3.22: Implementazione server con Plug

Basta eseguire il comando "mix deps.get" per scaricare le dipendenze.

Nel Listing 3.23 è riportato il codice che avvia un http server con la libreria Plug.

```
1 defmodule MyRouter do
2   use Plug.Router
3
4   plug :match
5   plug :dispatch
6
7   get "/ping" do
8     send_resp(conn, 200, "pong")
9   end
10
11   match _ do
12     send_resp(conn, 404, "oops")
13   end
14 end
```

Listing 3.23: Implementazione server con Plug

Per l'avvio del server ci si affida ad un Supervisor riportato nel Listing 3.24.

```

1  defmodule InteroperabilityTest.MyHttpApplication do
2    use Application
3    require Logger
4
5    def start(_type, _args) do
6
7      children = [
8        {Plug.Cowboy, scheme: :http,
9         plug: MyRouter, options: [port: cowboy_port()]}
10       ]
11
12      # opzioni per il supervisor del modulo Myhttp
13      opts = [strategy: :one_for_one, name: MyHttpServer.Supervisor]
14
15      Logger.info("Starting application on port #{cowboy_port()}...")
16      Supervisor.start_link(children, opts)
17    end
18
19    defp cowboy_port, do: Application.get_env(:example, :cowboy_port, 3000)
20
21  end

```

Listing 3.24: Supervisor dell'applicazione HTTP

HTTP Server Python

Un semplice Server python come quello di Elixir si può fare con la libreria Flask, il codice è riportato nel Listing 3.25

```

1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/ping')
6  def ping():
7      return 'pong'
8
9  if __name__ == '__main__':
10     app.run(debug=False)

```

Listing 3.25: Server Python(Flask)

Assumendo di avere l'interprete Python installato sulla macchina, per avviare il server basta installare la libreria Flask:

```

> pip install flask
...
> python server.py

```

HTTP Server Node

Con Node il Server viene implementato con la libreria Express.

```

1  const express = require('express');
2  const app = express();

```

```
3
4 app.get('/ping', (req, res) => {
5     res.send('pong');
6 });
7
8 const PORT = process.env.PORT || 5000;
9 app.listen(PORT, () => {
10     console.log('Server is running on port ${PORT}');
11 });
12
```

Listing 3.26: Server Node(Express)

Assumendo di avere Node.js installato sulla macchina per avviare il server basta installare la libreria Express:

```
> npm install express
...
> node server.js
```

3.5.2 Esecuzione Test Http

Come accennato viene usato il tool wrk per eseguire più richieste concorrenti e multithreading, vengono eseguite per semplicità sulla stessa macchina, sviluppi futuri possono prevedere un Load Testing più complesso con il tool Apache Jmeter, eseguendo i test su macchine differenti simulando applicazioni di vita reale. Come detto in questo luogo si vuole avere la percezione di quanto Elixir sia adatto a questo scopo, infatti l'architettura leggera di Elixir e il sistema di gestione della concorrenza consente di gestire migliaia di connessioni simultanee in modo efficiente senza bloccare il processo principale. Ciò consente di fornire tempi di risposta minimi, rendendo Elixir una scelta ideale per applicazioni real-time.

Per installare wrk su un sistema debian, basta eseguire

```
> sudo apt-get update
> sudo apt-get install wrk
```

Una volta installato basta eseguire un server per volta ed eseguire il comando per simulare richieste simultanee:

```
> wrk -t8 -c<n> -d10s --latency http://localhost:5000/ping
```

Il comando viene eseguito con 8 thread, e vengono lanciate n richieste concorrenti simulando n richieste simultanee di utenti. L'opzione `-latency` serve per stampare la distribuzione di latenza delle richieste.

n	richieste/s	Latenza AVG	Stdev
10	40943	334.87 μ s	2.64 ms
100	63697	1.66 ms	1.33 ms
1000	59442	42.68 ms	157.75 ms
10000	51872	81.47 ms	24.58 ms
20000	18092	215.25 ms	78.27 ms

Tabella 3.1: Elixir http server benchmark

Un esempio di output:

```
> wrk -t8 -c100 -d10s --latency http://localhost:5000/ping
Running 10s test @ http://localhost:5000/ping
 8 threads and 100 connections
Thread Stats      Avg      Stdev     Max    +/-  Stdev
Latency           1.66ms    1.33ms   34.83ms  83.83%
Req/Sec           8.01k     626.33   10.90k   70.50%
Latency Distribution
 50%      1.29ms
 75%      2.01ms
 90%      3.37ms
 99%      6.30ms
639802 requests in 10.04s, 89.70MB read
Requests/sec:  63697.09
Transfer/sec:   8.93MB
```

Sono stati effettuati dei test al variare di n richieste concorrenti e sono riportati i risultati ottenuti nelle Tabelle 3.1, 3.2, 3.3. Si può notare come Elixir performa in modo significativo rispetto agli altri due server, e da notare anche la latenza media di Elixir rispetto agli altri, d'altronde Elixir parallelizza le richieste il più possibile, durante i test con il tool **htop** si è riscontrato un uso della cpu prossima al 100% mentre con Node, il server è single Thread, perciò le richieste vengono eseguite su un'unica CPU, e non si è percepito un'aumento significativo delle CPU. E' anche da notare che anche il client eseguito sulla stessa macchina è multithreading, e fa un uso elevato della CPU anch'esso, perciò non si nega che Elixir possa performare ancor di più. Le righe con valori Null, sono test eseguiti in cui il client chiude la connessione al client per via di un timeout. Non si può affermare che Node e Python non possono fare di meglio, i server andrebbero configurati meglio, in questo luogo si è voluto vedere come Elixir con la concorrenza leggera riesce a performare e parallelizzare in modo ottimale. Ci sono benchmark in rete in cui Elixir tramite il framework **Phoenix**, riesce a gestire fino a 2 milioni di connessioni WebSocket simultanee su una singola macchina [20].

n	richieste/s	Latenza AVG	Stdev
10	6463	1.79ms μ s	6.61 ms
100	6258	15.32 ms	2.33 ms
1000	5621.09	171.61 ms	21.95 ms
10000	4988.81	519.53 ms	70.12 ms
Null	Null	Null	Null

Tabella 3.2: Node http server benchmark

n	richieste/s	Latenza AVG	Stdev
10	1679	4.69 ms	458.13 μ s
100	1695	56.36 ms	2.53 ms
1000	1609	88.96 ms	67.44 ms
Null	Null	Null	Null
Null	Null	Null	Null

Tabella 3.3: Python http server benchmark

Conclusioni e sviluppi futuri

Elixir si è dimostrato un linguaggio potente che migliora molti aspetti dello sviluppo software, soprattutto quello in ambito concorrenziale, sviluppare è stato piacevole dimostrando di essere un linguaggio moderno degno di nota.

Ha dimostrato di avere uno scheduling molto veloce che non degrada significativamente le performance, se si necessita di computazioni dove Elixir non ottiene prestazioni ottimali si può integrare con soluzioni più adatte come il C/C++, Rust o Python, tramite inter process communication o tramite funzioni native NIF. Ha dimostrato di avere una bassa latenza ottimale per sistemi real time, ed è quello il suo principale utilizzo adottato finora, ma ha dimostrato di essere una scelta promettente anche nell'ambito dell'IoT, dove affidabilità reattività e concorrenza sono dei fattori chiave.

Futuri studi possono vedere l'integrazione di Elixir in software IoT, tramite il framework Nerves o con soluzioni ad Hoc per il caso specifico.

Bibliografia e sitografia

- [1] Bruce Tate Ben Marx José Valim. «Adopting Elixir». In: *The Pragmatic Programmers*, 2018. Cap. 5, p. 96.
- [2] Bruce Tate Ben Marx José Valim. «Adopting Elixir». In: *The Pragmatic Programmers*, 2018. Cap. 7, pp. 125–128.
- [3] *bencheeorg/benchee: Easy and extensible benchmarking in Elixir providing you with lots of statistics!* <https://github.com/bencheeorg/benchee>.
- [4] *Embedded and cloud Elixir for grid-management at Sparkmeter - The Elixir programming language*. <https://elixir-lang.org/blog/2023/03/09/embedded-and-cloud-elixir-at-sparkmeter/>.
- [5] *Erlang – erl_nif*. https://www.erlang.org/doc/man/erl_nif.
- [6] *Erlang – Ports*. https://www.erlang.org/doc/tutorial/c_port.
- [7] *Erlang – Processes*. https://www.erlang.org/doc/efficiency_guide/processes.
- [8] *Express - Node.js web application framework*. <https://expressjs.com/>.
- [9] *GenServer — Elixir v1.16.2*. <https://hexdocs.pm/elixir/GenServer.html>.
- [10] *Hex*. <https://hex.pm/>.
- [11] *HexDocs*. <https://hexdocs.pm/>.
- [12] Saša Juric. «Elixir in Action». In: 2019. Cap. 5.
- [13] *Lambda calcolo - Wikipedia*. https://it.wikipedia.org/wiki/Lambda_calcolo.
- [14] Fausta D'Epiro Luisa. «Analisi prestazionali del linguaggio Elixir per operazioni massive IoT e di Cyber». Università degli studi di Cassino e del lazio meridionale, 2022/2023.
- [15] *Nerves Project*. <https://nerves-project.org/>.

- [16] *Plug* — *Plug v1.15.3*. <https://hexdocs.pm/plug/readme.html>.
- [17] *Poison* — *Poison v5.0.0*. <https://hexdocs.pm/poison/Poison.html>.
- [18] *Port* — *Elixir v1.12.3*. <https://hexdocs.pm/elixir/1.12/Port.html>.
- [19] *SparkMeter* - *For reliable, clean and efficient electricity*. <https://www.sparkmeter.io/>.
- [20] *The Road to 2 Million Websocket Connections in Phoenix* - *Phoenix Blog*. <https://phoenixframework.org/blog/the-road-to-2-million-websocket-connections>.
- [21] Dave Thomas. «Programming Elixir \geq 1.6». In: *The Pragmatic Programmers*, 2018. Cap. 3, p. 23.
- [22] *Welcome to Flask* — *Flask Documentation (3.0.x)*. <https://flask.palletsprojects.com/en/3.0.x/>.
- [23] *wg/wrk: Modern HTTP benchmarking tool*. <https://github.com/wg/wrk>.

Ringraziamenti

La scelta di un obbiettivo difficile è stato un caso, l'aiuto nel raggiungerlo non lo è stato.

Un doveroso e sentito ringraziamento al Prof. Ciro D'Elia, che mi ha fornito preziosi spunti per questo studio.

Un ringraziamento speciale va mia madre e mio padre che mi hanno permesso di intraprendere questo percorso con tutti i sacrifici che hanno fatto e continuano a fare, permettendomi di compiere i miei passi con più leggerezza.

Un altro ringraziamento speciale a Valentina che negli ultimi due anni mi ha sostenuto aggiungendo un colore alla mia vita in scala di grigi.

Grazie a tutta la mia famiglia che mi ha supportato.

Grazie a tutti i miei amici e i momenti condivisi insieme rendendo la vita più curiosa e divertente.

Grazie.