PARALLEL AND DISTRIBUTED SYSTEMS: PARADIGMS AND MODELS

ACADEMIC YEAR 2018-2019

FINAL PROJECT

Autonomic farm pattern

LUCA CORBUCCI

MATRICOLA: 516450

DIPARTIMENTO DI INFORMATICA

UNIVERSITÀ DI PISA

Settembre 2019

Indice

Capitolo 1

Introduzione

Il progetto consiste nello sviluppo di una farm capace di adattare autonomamente il numero di worker attivi in base ad un certo service time scelto dall'utente.
L'autonomic farm viene avviata fornendo:

- Una collezione di task in input da calcolare;
- Una funzione per calcolare ognuno dei task;
- Il tsGoal atteso;
- Un parallelism degree iniziale.

Questo framework è stato implementato in C++ utilizzando i classici Thread e poi usando FastFlow.

Capitolo 2

Analisi del problema

L'implementazione dell'Autonomic Farm è simile a quella di una classica farm, ad esclusione del fatto che in questo caso è presente anche la possibilità di modificare il numero di worker attivi. La modifica del numero dei worker deve permettere di mantenere durante tutta l'esecuzione dell'Autonomic Farm Pattern un service time prossimo a quello indicato come parametro. Idealmente vorremmo ottenere un grafico della variazione del service time simile a quello della figura 2.1 con degli aumenti e delle diminuzioni del valore dovuti alla variazione di task che deve essere calcolato.

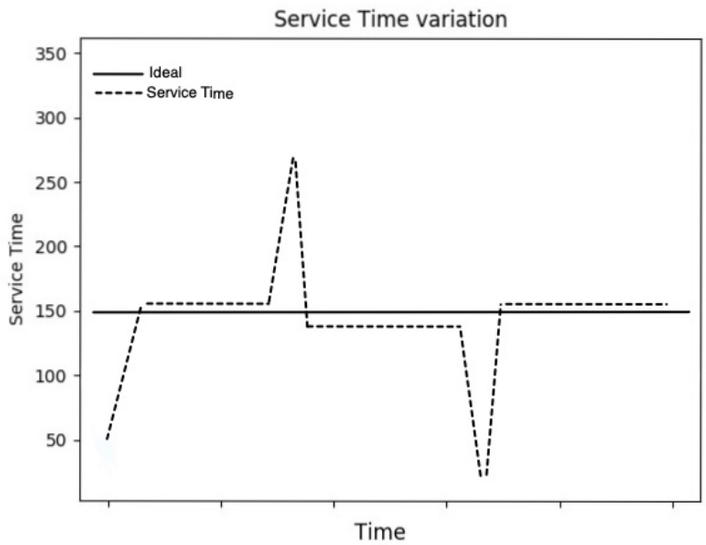


Figura 2.1: Variazione ideale del service time, in corrispondenza della variazione del task il service time aumenta o diminuisce, poi viene riportato al valore ideale modificando il numero di worker attivi

L'idea è stata quella di creare vari componenti andandoli poi ad unire in modo da formare l'Autonomic Farm. Nel caso dell'implementazione con i thread di C++ abbiamo i seguenti componenti:

- Emitter
- Worker
- Collector

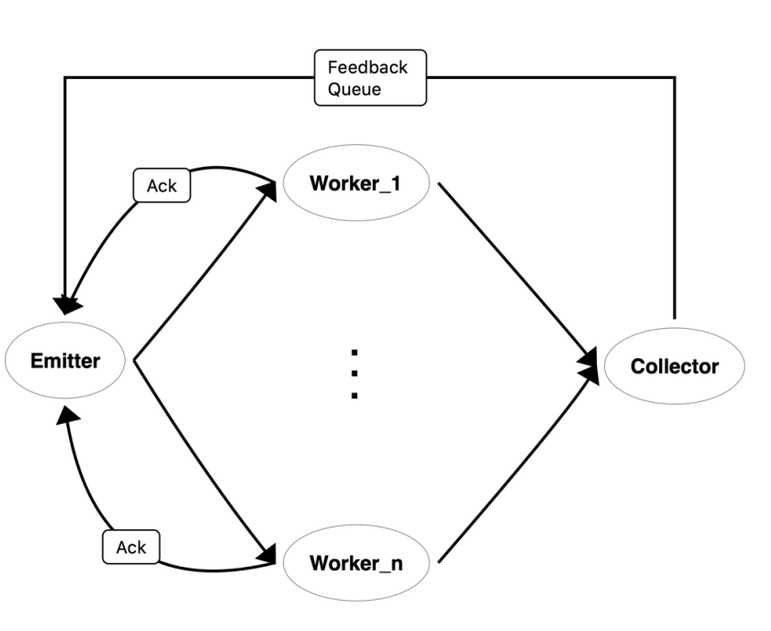


Figura 2.2: Organizzazione dei componenti nell'implementazione "Classic Threads"

Nel caso di FastFlow invece abbiamo:

- Emitter
- External Emitter
- Worker
- Collector

FastFlow.png

Figura 2.3: Organizzazione dei componenti nell'implementazione con FastFlow.

Rispetto alla soluzione con i thread classici, nella versione implementata con FastFlow è stato necessario creare un componente in più, chiamato External Emitter. Nel caso di FastFlow il progetto è stato implementato utilizzando una pipeline che contiene come primo componente l'external emitter e come secondo componente la farm. Questa scelta è stata necessaria per fare in modo che l'emitter possa ricevere tutti i vari task da eseguire inoltrandoli immediatamente ad un worker (se disponibile) oppure memorizzandoli in un array. I task memorizzati nell'array verranno inoltrati ad un worker nel momento in cui questo invierà un ack all'emitter per notificare la terminazione della computazione che gli era stata precedentemente assegnata.

Capitolo 3

Struttura del progetto e documentazione

Il pacchetto contenente il progetto è organizzato nelle seguenti cartelle:

- src: contiene tutto il codice che è stato prodotto. All'interno della cartella src sono presenti varie sotto cartelle:
  - AFP-FFQueue: contiene l'implementazione dell'Autonomic Farm che utilizza le code di FastFlow per la comunicazione;
  - AFP-SafeQueue: contiene l'implementazione dell'Autonomic Farm che utilizza le code con le lock per la comunicazione;
  - FastFlow: implementazione dell'autonomic farm utilizzando FastFlow;
  - Utils: vari file e librerie utilizzate all'interno del progetto;
  - Test: contiene il makefile e i file di test che ho usato per provare l'autonomic farm.
- Lib: contiene i file di FastFlow;
- Doc: documentazione del progetto;
- Report: contiene il report del progetto.

Tutto il codice è stato commentato in modo da rendere chiaro il funzionamento, inoltre ho utilizzato Doxygen per generare la documentazione del progetto.

Capitolo 4

Scelte implementative

4.1 Funzionamento Emitter

L'emitter deve gestire la modifica del numero di worker attualmente attivi, per farlo memorizza due array di lunghezza pari al numero di worker:

- Il primo array serve per memorizzare se un worker è attualmente impegnato nel calcolo di un task o se è libero
- Un secondo array serve per memorizzare se un worker è stato addormentato o no

Quando viene avviato per la prima volta l'emitter invia un task da completare a ciascuno dei worker che sono stati attivati, poi, una volta ricevuto il feedback deve addormentarne o risvegliarne alcuni (nella sezione dedicata ai worker verrà spiegato come vengono addormentati o risvegliati). Quando l'emitter risveglia o addormenta un worker, modifica l'array di bit nella posizione corrispondente a quello specifico worker. L'emitter invia un segnale anche se il worker da addormentare sta ancora eseguendo un precedente task. Sarà poi compito del worker terminare il lavoro che aveva iniziato e non fare altro fino a nuovo avviso.
Una volta eseguita una modifica del numero di worker l'emitter attende un certo tempo (specificato come parametro dall'utente) per effettuare la modifica successiva.
L'emitter ha i seguenti canali di comunicazione con l'esterno:

- Un numero di code pari al numero di worker per inviare i task ai vari worker;
- Una coda per ricevere gli ack dai vari worker in modo da capire quando finisce il lavoro;
- Una coda per ricevere il feedback dal collector.

Quando l'emitter ha terminato il suo lavoro, risveglia tutti i worker e poi invia un task "speciale" (contenente una variabile con valore "-1") in modo che anche questi possano terminare.

4.2 Funzionamento Worker

Ogni worker svolge i seguenti passaggi:

- Riceve un task dall'emitter;
- Calcola quel task memorizzando il tempo prima dell'inizio della computazione e il tempo finale;
- Invia al collector il risultato della computazione effettuata.

Il worker deve essere in grado di fermarsi su richiesta dell'emitter, per questo al suo interno ha una variabile di condizione che viene utilizzata per eseguire la wait() e poi per risvegliare il thread. Sono stati implementati due metodi che possono essere chiamati dall'emitter:

- stopWorker(): metodo che setta a false un booleano "status", quando il worker termina la computazione del task che ha ha ricevuto in precedenza, controlla se questo booleano è false e in tal caso esegue la wait su una variabile di condizione;
- restartWorker(): metodo che modifica il valore del booleano "status" ed esegue una $notify\_one()$ sulla variabile di condizione, in questo modo il worker può risvegliarsi e può eseguire la computazione del prossimo task.

4.3 Funzionamento Collector

Il collector si occupa di gestire il calcolo del numero di worker che dovranno essere mantenuti attivi per garantire il service time che abbiamo scelto di avere. In particolare per calcolare quanti worker dovranno svolgere i task successivi ho utilizzato questa formula:

$$Nuovo\ Numero\ Worker = \frac{Tempo\ computazione\ di\ un\ task}{TSGoal} \tag{4.1}$$

Una volta calcolato il nuovo numero di worker, il collector utilizza la feedback queue che lo collega all'emitter per inviare questa informazione. Sarà poi l'emitter a inviare i segnali ai vari worker per fare in modo che questi vengano addormentati o risvegliati.
Il collector ha anche il compito di memorizzare in una coda il risultato dell'esecuzione della funzione calcolata nel worker. Opzionalmente questa coda di risultati può anche essere stampata.

4.4 Comunicazione tra emitter, worker e collector

Quando ho iniziato il progetto una delle prime cose che ho fatto è stata un'implementazione della farm che però non comprendeva ancora la possibilità di modificare il numero di worker. In questo

primo test avevo utilizzato per la comunicazione tra l'emitter e i vari worker e tra i worker e il collector, delle code "Safe" implementate utilizzando le lock. L'utilizzo di queste code e di Task abbastanza semplici da far svolgere al Worker mi hanno portato a pensare che non fosse corretto sfruttarle nel progetto perchè il tempo per la comunicazione risultava essere maggiore rispetto al tempo necessario per svolgere il task. Quindi ho provato una soluzione differente ed ho utilizzato le code senza lock di FastFlow.
Una volta completata l'implementazione dell'Autonomic Farm pattern ho confrontato le prestazioni dei due sistemi di comunicazione e ho avuto due risultati differenti dipendenti dal tipo di task eseguito:

- Facendo svolgere all'Autonomic Farm un task abbastanza "pesante" (ad esempio il calcolo della funzione di Fibonacci con un numero maggiore di 30), tra i due sistemi di comunicazione non sono presenti grandi differenti e il tempo di completamento è molto simile. Il test in questione è stato svolto sullo Xeon Phi. In questo test ho utilizzato 128 worker, un tsGoal pari a 15 e il task da calcolare era la funzione di Fibonacci, nella coda di input ho utilizzato tre valori differenti. Il test può essere riprodotto eseguendo il file $testConfronti.sh$. Per ognuna delle opzioni sono stati eseguiti 5 test differenti in modo da calcolarne poi la media dei tempi di esecuzione, nella tabella 4.1 è presente il risultato ottenuto, il tempo è espresso in secondi.

Tipo Coda	Tempo
Safe Queue	76,83s
Code FastFlow	76,46s

Tabella 4.1: Completion time dell'Autonomic Farm con task "pesanti"

I tempi riportati nella tabella prendono in considerazione la creazione di tutti i componenti necessari all'autonomic farm, la loro esecuzione e la successiva join.

- Il discorso è decisamente differente se calcoliamo un task più "semplice", ad esempio un controllo del numero primo o la funzione di Fibonacci con numeri piccoli. In questo caso la differenza tra i due sistemi di comunicazione è piuttosto evidente, nella tabella 4.2 confrontiamo i tempi di completamento dello stesso test eseguito sullo Xeon Phi e sul mio computer (MacBook Pro con processore Intel i7 con 4 core e 8 contesti).

Task	Macchina	Tempo Safe Queue	Tempo Coda FastFlow
Fibonacci	XeonPhi	23.22s	2.64s
Fibonacci	MacBook	13.77s	0.72s
isPrime	XeonPhi	242.67	20.15
isPrime	MacBook	157.19	7.14

Tabella 4.2: Confronto tra i completion Time dell'Autonomic Farm con Safe Queue e code di FastFlow con task "leggeri"

La mia impressione è che utilizzando task più pesanti si riesca a mascherare meglio il tempo della comunicazione mentre con task "semplici" questo risulta dominante rispetto al tempo della computazione.

Per cercare di capire meglio il funzionamento delle due implementazioni ho utilizzato Gprof, all'interno della cartella Test sono presenti i risultati ottenuti con questa analisi (Analysis1.txt, Analysis2.txt, Analysis3.txt, Analysis4.txt). L'analisi effettuata con Gprof ha evidenziato che nel caso dell'utilizzo delle Safe Queue e di un task veloce da calcolare, il 77% del tempo della computazione viene speso per la comunicazione, quindi per il passaggio dei task da emitter a worker e soprattutto per l'invio dei feedback dal collector all'emitter. La stessa cosa non accade utilizzando le code di FastFlow e task "semplici", in questo caso il 94% del tempo complessivo viene speso per il calcolo della funzione dei workers. La situazione è differente quando andiamo ad utilizzare task più pesanti che quindi richiedono più tempo per essere calcolati. In questo secondo caso Gprof indica che i tempi necessari per il calcolo della funzione sono dominanti rispetto a tutto il resto, sia utilizzando le Safe Queue sia le code di FastFlow.

4.5 Opzioni

I file che ho creato per testare il progetto possono essere eseguiti indicando dei parametri e delle opzioni da cui dipende il comportamento del programma.

Le opzioni sono state implementate utilizzando la libreria cxxopts.hpp [1], quelle che ho scelto di inserire sono le seguenti:

- -s: va utilizzata specificando un valore true o false, serve per indicare l'utilizzo o meno della safe queue;
- -f: serve per indicare l'utilizzo dell'Autonomic Farm implementata con FastFlow, va usato senza specificare l'opzione precedente;
- -d: opzione di debug usata per stampare alcune statistiche e informazioni generate dall'autonomic farm pattern.

Capitolo 5

Benchmark

Gli esperimenti sono stati eseguiti sulla CPU Xeon Phi con 64 core fisici, ognuno dei quali ha 4 contesti. Nei seguenti paragrafi ogni tempo è espresso in millisecondi e rappresenta la media di 5 test differenti svolti sulla stessa macchina e con gli stessi parametri. Ognuna delle soluzioni implementata è stata testata con un numero di thread attivi $N \in 1, 2, ..., 128$. Per eseguire l'autonomic farm pattern è possibile utilizzare uno dei test presenti nella cartella "Test" specificando i seguenti parametri:

- Numero di worker;
- TSGoal;
- Dimensione della collezione in input;
- Valore in input nella prima parte della collezione;
- Valore in input nella seconda parte della collezione;
- Valore in input nella terza parte della collezione;
- Valore intero che indica ogni quanto tempo vogliamo modificare il numero di worker.

Tra i vari test è presente anche *testLeaks.sh* che controlla se alla fine dell'esecuzione del programma è stata liberata o meno tutta la memoria precedentemente allocata.

5.1 Speedup

La speedup è calcolata come $S(N) = T_{seq}/T_{par}(N)$.

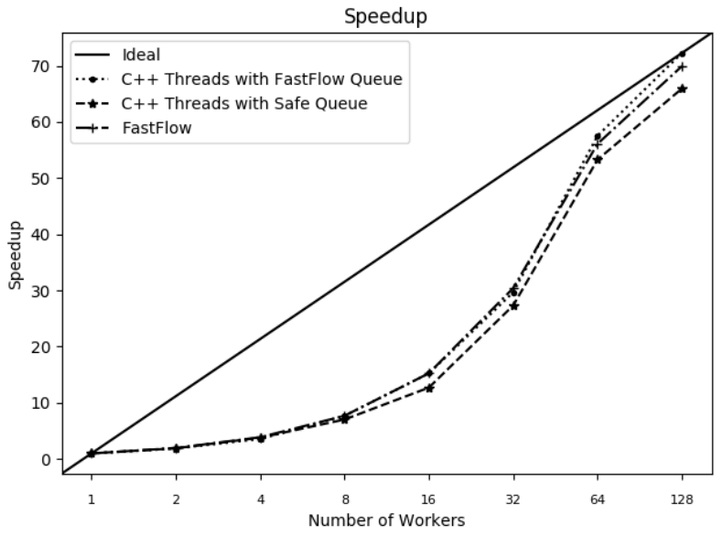


Figura 5.1: Speedup al variare del numero di worker e al tipo di implementazione

Nel grafico 5.1 è riportata la curva della speedup al variare del tipo di implementazione utilizzata (FastFlow/Classic Threads) e del tipo di comunicazione utilizzata (Safe Queue/FastFlow queue). Dal grafico possiamo notare come la curva della speedup inizia a rallentare a partire da $N = 64$.

5.2 Scalability

La scalability è calcolata come $Scalability(p) = T_{par}(1)/T_{par}(N)$. Per il calcolo della scalability ho eseguito un test sullo Xeon Phi. Il test può essere replicato eseguendo il seguente comando: *testScalability.sh*.

Nel grafico 5.2 viene mostrata la curva della scalability.

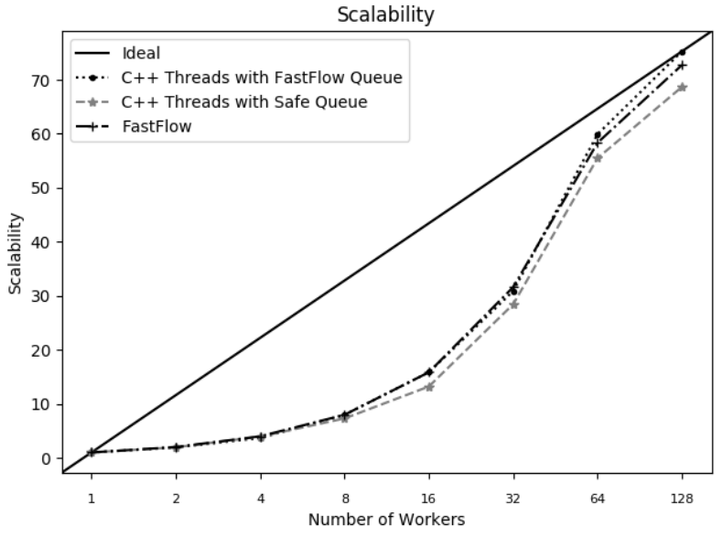


Figura 5.2: Grafico dell'andamento della Scalability

5.3 Completion Time

Per confrontare i tempi di completamento ho effettuato il seguente test andando a variare il numero di thread utilizzati, ho replicato ogni esperimento per 5 volte e i dati riportati sono una media dei risultati ottenuti. Il test può essere replicato eseguendo *testCompletionTime.sh*. Nel grafico 5.3 viene riportato il confronto del completion time tra le varie implementazioni dell'Autonomic Farm, nella tabella 5.1 sono indicati i tempi delle esecuzioni del test eseguito sullo Xeon Phi.

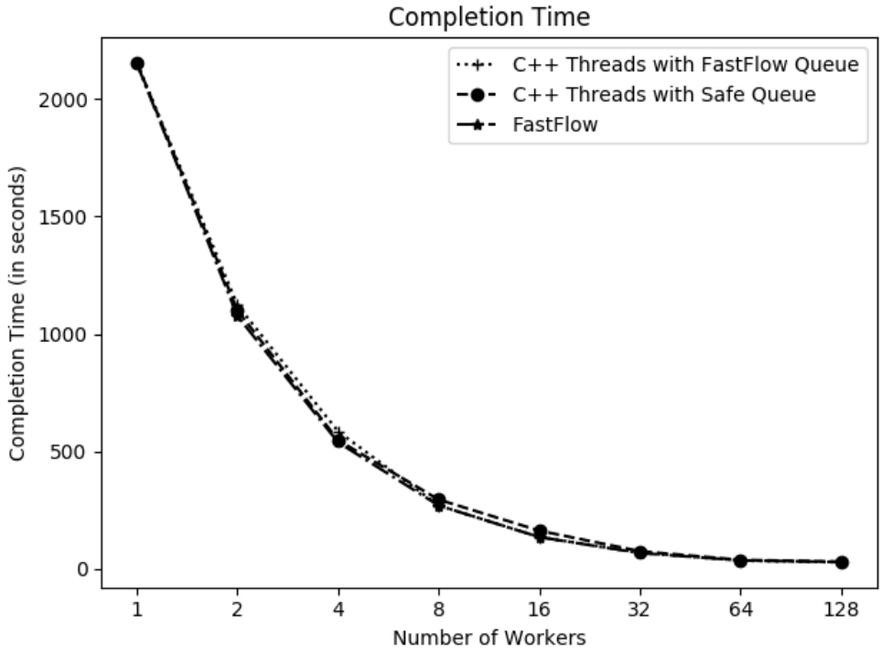


Figura 5.3: Completion time al variare del numero di workers e del tipo di implementazione utilizzata, il tempo è espresso in secondi

Worker	Tempo Safe Queue	Tempo FastFlow Queue	FastFlow
1	2154.43s	2154.31s	2151.47s
2	1099.97s	1123.07s	1076.60s
4	548.22s	585.63s	539.22s
8	295.27s	270.16s	270.09s
16	163.06s	135.84s	135.78s
32	75.72s	69.76s	68.04s
64	38.82s	35.95s	36.87s
128	31.37s	28.62s	29.58s

Tabella 5.1: Completion time al variare del numero di workers e del tipo di implementazione utilizzata, il tempo è espresso in secondi

5.4 Efficiency

L'efficiency è calcolata come $E(n) = Tseq/(n*T(n)) = Sp(n)/n$.

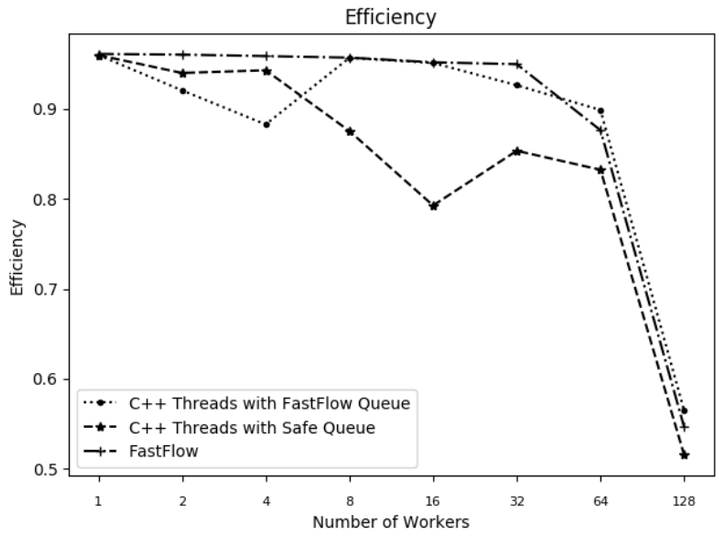
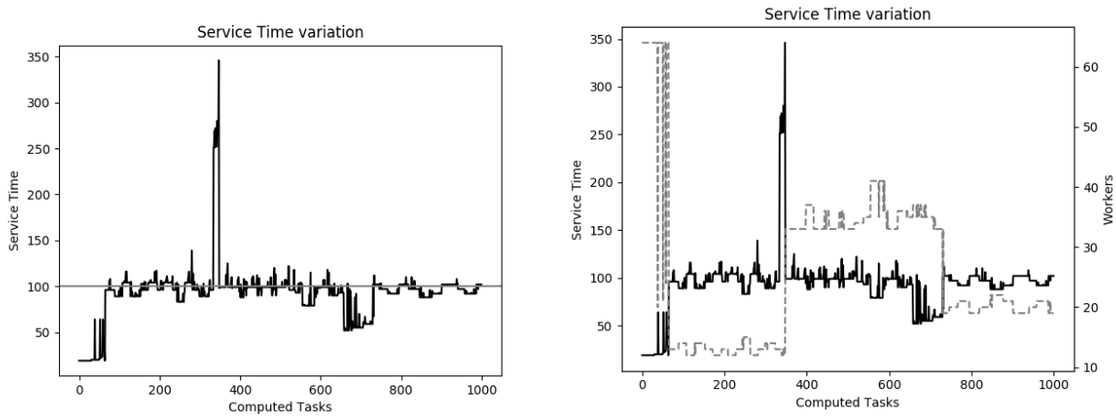


Figura 5.4: Grafico dell'andamento dell'efficiency

Come possiamo vedere nel grafico 5.4, l'efficiency mantiene un valore prossimo allo 0.9 fino a $N = 64$ per poi scendere ad un valore vicino a 0.5 nel momento in cui il numero di worker sale a 128.

5.5 Variazione del Service Time

Per capire la variazione del service time in base al numero di worker attivi e per comprendere quanto questo valore si distacca dal TsGoal passato come parametro all'Autonomic Farm ho pensato di produrre il grafico di Figura 5.5a che mostra l'andamento del valore del Service Time:



(a) Andamento del service Time con 64 thread e un tTsGoal uguale a 100

(b) Andamento del service Time e confronto con la variazione del numero di workers

Figura 5.5

L'esperimento è replicabile con il seguente comando: *mainFib.out* 64 100 1000 39 41 40 1500 $-$ *d ts*. Gli aumenti improvvisi del service time che vediamo nel grafico sono dovuti al fatto che in

quel momento si inizia a lavorare su un task differente rispetto al precedente e quindi la farm deve adattarsi e modificare il numero di worker in modo da ottenere il service time desiderato. Dalla figura 5.5b possiamo notare come in corrispondenza di un aumento del service time (riga nera) ci sia anche una modifica del numero di worker (riga tratteggiata) che permette di riportare il service time al valore passato come parametro.
In questo caso è stata eseguita l'Autonomic Farm con 64 thread, un TsGoal pari a 100ms, 1000 task in input e un aggiornamento del numero di worker ogni secondo e mezzo. Il service time in questo caso ha assunto in media un valore pari a 92.

Capitolo 6

Conclusioni

Guardando i dati relativi alla Scalability e allo Speedup possiamo vedere come l'implementazione con i thread di C++ e l'utilizzo delle code di FastFlow ci fornisca un risultato di poco migliore rispetto alle altre soluzioni. L'utilizzo delle code di FastFlow ci permette di avere buone prestazioni indipendentemente dal task che vogliamo far calcolare ai worker, a differenza della soluzione con le Safe Queue che non ci garantisce un tempo di completamento accettabile quando utilizziamo dei task "semplici". Dal punto di vista del programmatore che produce il framework la soluzione migliore è quella che sfrutta FastFlow perchè risulta più semplice implementare il tutto.

Bibliografia

[1] Cxxopts: Lightweight c++ command line option parser. https://github.com/jarro2783/cxxopts.