

# POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione  
Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in Ingegneria Informatica



## GENERAZIONE DI WORKLOAD PER SISTEMI PARALLELI

Relatore: Prof. Giuseppe SERAZZI

Co-relatore: Prof. Marco GRIBAUDO

Tesi di Laurea di:

Andrea ROSÀ Matr. 782866

Anno Accademico 2012-2013



*a Eleonora,  
che con amore  
mi sostiene  
in ogni istante.*



## Sommario

In questa Tesi di Laurea Magistrale l'autore propone un *benchmark* innovativo con la capacità di riprodurre un *workload* determinato, impostato dall'utente, e di calcolare alcune metriche che consentono l'analisi del traffico generato. Lo strumento sviluppato ha l'obiettivo di consentire la modellazione e l'ottimizzazione di un'applicazione arbitraria, simulandone il comportamento in un sistema e verificandone le prestazioni.

Dopo una breve introduzione ai motivi che hanno spinto allo sviluppo di un simile strumento, il presente elaborato introduce dapprima il contesto nel quale l'applicativo si colloca, descrivendo lo stato dell'arte di benchmark, sistemi paralleli e sistemi distribuiti, per poi fornire le nozioni di base necessarie alla comprensione dei metodi e degli strumenti utilizzati. In seguito, si dettaglia il benchmark in ogni sua parte, presentando la sua architettura, il suo funzionamento, la sua implementazione e le analisi prodotte. Infine si dimostra la sua capacità di riprodurre il workload di un'applicazione esistente, nota e pubblicamente accessibile. La validazione sperimentale modella un programma implementato per il framework Apache *Hadoop* mediante il paradigma *MapReduce* dopo averne analizzato le caratteristiche e ricavato un possibile taskgraph.

**Parole chiave:** performance, benchmark, workload, sistemi paralleli, *Hadoop*, *MapReduce*.



## Abstract

In this Master's Thesis the author proposes an innovative *benchmark* with the capabilities of reproducing a specific *workload*, set by the user, and computing some metrics that allow the analysis of the generated traffic. The tool aims to allow the modeling and the optimization of an arbitrary application by simulating its behavior in a given system and verifying its performance.

After a brief introduction to the reasons that led to the development of such a tool, this work first describes the cutting edge technologies of benchmarks, parallel and distributed systems, secondly it provides the basic information that is necessary for the understanding of the methods and tools used in the dissertation. Then, it details the developed benchmark and its components, presenting its architecture, its working principles, its implementation and the produced analyses. Finally, it demonstrates the ability to reproduce the workload of an existing, recognizable and publicly accessible application. The experimental validation models a program implemented for the Apache *Hadoop* framework, using the *MapReduce* programming paradigm, after having analyzed its characteristics and extracted a possible taskgraph.

**Keywords:** performance, benchmark, workload, parallel systems, *Hadoop*, *MapReduce*.





## *Ringraziamenti*

*Desidero porgere i miei più sinceri ringraziamenti alle persone che hanno contribuito, in qualche modo, alla realizzazione di questa tesi.*

*In primo luogo, ai professori Giuseppe Serazzi e Marco Gribaudo, che hanno ideato questo lavoro e il cui supporto non è mai mancato, sia in ambito accademico, sia in quello professionale. I loro preziosi suggerimenti, le loro idee e le loro correzioni hanno guidato fin dal principio lo sviluppo e l'elaborazione di questo progetto.*

*In secondo luogo, a tutti coloro che mi hanno incoraggiato e hanno creduto in questo lavoro, fin dalle sue prime fasi, fornendo circostanziate critiche e utili proposte di sviluppo.*

*Infine, un ringraziamento particolare a Eleonora, per il suo supporto morale e il suo contributo nella revisione grammaticale e sintattica dell'elaborato e ai miei genitori, per l'aiuto che mi hanno sempre dato e per avermi concesso, con i loro sacrifici, l'opportunità di scrivere queste pagine.*



# Indice

<b>Introduzione</b>	<b>1</b>
<b>1 Stato dell'arte</b>	<b>5</b>
1.1 Benchmark . . . . .	5
1.1.1 Il consorzio SPEC . . . . .	8
1.1.2 La suite di benchmark DaCapo . . . . .	14
1.2 Sistemi paralleli . . . . .	16
1.2.1 Sistemi sequenziali . . . . .	17
1.2.2 Instruction Level Parallelism . . . . .	18
1.2.3 Thread Level Parallelism . . . . .	21
1.2.4 Sistemi multicore . . . . .	23
1.2.5 Sistemi multiprocessore . . . . .	27
1.3 Sistemi distribuiti . . . . .	28
1.3.1 Cluster . . . . .	29
1.3.2 Grid . . . . .	30
1.3.3 Cloud computing . . . . .	32
1.3.4 Virtualizzazione . . . . .	38
<b>2 Nozioni di base</b>	<b>41</b>
2.1 Sistema operativo Linux . . . . .	41
2.1.1 Gestione dei processi . . . . .	42
2.1.2 Gestione dell'I/O . . . . .	48
2.1.3 Gestione del tempo . . . . .	53
2.2 Il progetto Apache Hadoop . . . . .	57
2.2.1 Hadoop Distributed File System . . . . .	58
2.2.2 Hadoop MapReduce . . . . .	61
2.3 Miscellanea . . . . .	67
2.3.1 Prodotto tra matrici . . . . .	67
2.3.2 Generazione di numeri pseudo-casuali . . . . .	68
<b>3 Il benchmark realizzato</b>	<b>71</b>
3.1 Descrizione del progetto . . . . .	71
3.1.1 Obiettivi . . . . .	71
3.1.2 Innovazione e possibili usi . . . . .	72

---

3.2	Considerazioni . . . . .	72
3.2.1	Limiti . . . . .	72
3.2.2	Usabilità del prototipo . . . . .	73
3.2.3	Durata e calibrazione . . . . .	73
3.2.4	Gestione dell'I/O . . . . .	73
3.2.5	Matrici e loro uso . . . . .	74
3.3	Architettura . . . . .	74
3.4	Il taskgraph . . . . .	75
3.4.1	Definizione . . . . .	75
3.4.2	Implementazione . . . . .	76
3.4.3	Un esempio operativo . . . . .	79
3.5	Il generatore di workload . . . . .	81
3.5.1	Riproduzione del traffico . . . . .	81
3.5.2	Gestione delle informazioni operative . . . . .	89
3.6	L'analizzatore di log . . . . .	93
3.6.1	File di statistiche testuali . . . . .	94
3.6.2	Profilo temporale dell'esecuzione . . . . .	97
3.6.3	Profilo temporale dei processi . . . . .	99
3.6.4	Profilo temporale delle fasi . . . . .	101
<b>4</b>	<b>Validazione sperimentale</b>	<b>107</b>
4.1	Il programma <i>pi</i> . . . . .	108
4.2	Ipotesi e semplificazioni . . . . .	110
4.2.1	Ipotesi generali . . . . .	110
4.2.2	Semplificazione dei tempi . . . . .	111
4.2.3	Ipotesi sulle operazioni CPU-bound e I/O-bound . . . . .	112
4.2.4	Semplificazioni su mapper e reducer . . . . .	112
4.3	Analisi del workload . . . . .	113
4.3.1	Ricostruzione del workload . . . . .	114
4.3.2	Ricostruzione del profilo temporale dell'esecuzione . . . . .	116
4.4	Riproduzione del taskgraph . . . . .	118
4.5	Commenti sui risultati . . . . .	121
4.6	Statistiche e grafici . . . . .	122
	<b>Conclusioni</b>	<b>135</b>
	Limiti e sviluppi futuri . . . . .	135
	<b>Lista degli acronimi</b>	<b>139</b>
	<b>Bibliografia</b>	<b>143</b>
	<b>Indice analitico</b>	<b>147</b>

## Elenco delle figure

1	Funzionamento di una pipeline . . . . .	19
2	Confronto tra i vari tipi di multithreading . . . . .	22
3	Diagramma semplificato degli stati di un processo . . . . .	44
4	Esempio grafico di un'operazione MapReduce . . . . .	67
5	Schema architetturale del benchmark . . . . .	75
6	Esempio di taskgraph secondo la definizione classica . . . . .	76
7	Descrizione grafica di un taskgraph di esempio . . . . .	80
8	Esempio di un <i>profilo temporale dell'esecuzione</i> . . . . .	98
9	Dettaglio di un profilo temporale di un'esecuzione . . . . .	99
10	Esempio di un <i>profilo temporale dei processi</i> . . . . .	100
11	Esempi di <i>profili temporali delle fasi</i> di tipo CPU e I/O . . . . .	104
12	Esempi di <i>profili temporali delle fasi</i> di tipo FORK e JOIN . . . . .	105
13	Esempi di <i>profili temporali delle fasi</i> di ogni tipo . . . . .	106
14	Taskgraph ricostruito del programma <i>pi</i> . . . . .	115
15	<i>Profilo temporale dell'esecuzione</i> di <i>pi</i> . . . . .	117
16	<i>Profilo temporale dell'esecuzione</i> della simulazione di <i>pi</i> . . . . .	126
17	<i>Profili temporali dell'esecuzione</i> nella fase iniziale del workload . . . . .	127
18	<i>Profili temporali dell'esecuzione</i> nella fase centrale del workload . . . . .	128
19	<i>Profili temporali dell'esecuzione</i> nella fase finale del workload . . . . .	129
20	<i>Profilo temporale dei processi</i> della simulazione di <i>pi</i> . . . . .	130
21	<i>Profili temporali delle fasi</i> CPU e I/O della simulazione di <i>pi</i> . . . . .	131
22	<i>Profili temporali delle fasi</i> FORK e JOIN della simulazione di <i>pi</i> . . . . .	132
23	<i>Profili temporali delle fasi</i> di ogni tipo della simulazione di <i>pi</i> . . . . .	133



## Elenco delle tabelle

1	Elenco delle principali suite di benchmark SPEC attive ad oggi	9
2	Elenco dei benchmark contenuti in SPEC CPU2006 . . . . .	11
3	Tempi di riferimento per tutti i benchmark di SPEC CPU2006	12
4	Elenco dei benchmark contenuti in DaCapo-9.12-bach . . . . .	15
5	Esempio di un file di statistiche testuali . . . . .	95
6	Configurazione del taskgraph validante <i>pi</i> . . . . .	118
7	File di statistiche testuali prodotto dalla simulazione di <i>pi</i> . .	123





## Elenco dei codici

1	Definizione della funzione <code>clone</code> . . . . .	46
2	Definizione della funzione <code>waitpid</code> . . . . .	47
3	Definizione della funzione <code>open</code> . . . . .	49
4	Definizione della funzione <code>pread</code> . . . . .	49
5	Definizione della funzione <code>pwrite</code> . . . . .	51
6	Definizione della funzione <code>clock_gettime</code> . . . . .	55
7	Definizione della struttura <code>timespec</code> . . . . .	55
8	Implementazione della funzione <code>computeTime</code> . . . . .	55
9	Esempio di richiesta di informazioni temporali . . . . .	56
10	Elementi della struttura <code>OrdinaryPhase</code> . . . . .	77
11	Elementi della struttura <code>LoopPhase</code> . . . . .	77
12	Elementi della struttura <code>ForkPhase</code> . . . . .	78
13	Elementi della struttura <code>JoinPhase</code> . . . . .	78
14	Codice semplificato della procedura <code>simulate</code> . . . . .	82
15	Esecuzione delle fasi I/O-bound . . . . .	85
16	Esecuzione delle fasi LOOP . . . . .	86
17	Esecuzione delle fasi FORK . . . . .	87
18	Esecuzione delle fasi JOIN . . . . .	89
19	Elementi della struttura <code>Statistics</code> . . . . .	90
20	Scrittura dei file di log su disco . . . . .	91
21	Funzioni di memorizzazione delle informazioni operative . . . . .	92
22	Elementi della struttura <code>ProcessInterval</code> . . . . .	101
23	Algoritmo di creazione degli intervalli temporali . . . . .	102



# Introduzione

## Sistemi paralleli e benchmark

*«For over a decade prophets have  
voiced the contention that the  
organization of a single computer  
has reached its limits and that  
truly significant advances can be  
made only by interconnection of  
a multiplicity of computers»*

---

Gene M. Amdahl  
1967

LE PRESTAZIONI dei sistemi di calcolo, rappresentate significativamente dal tempo necessario per compiere un'operazione, sono da molti anni oggetto di approfonditi studi di ricerca. Problemi impossibili da risolvere fino a qualche anno fa sono oggi una realtà grazie al contributo dei ricercatori e della comunità accademica.

I *sistemi paralleli* rappresentano il più moderno passo evolutivo dell'architettura del singolo calcolatore. L'impetuoso sviluppo dell'informatica e dell'elettronica negli ultimi decenni ha spinto fino al limite le strutture sequenziali degli elaboratori, in cui un singolo processore è sfruttato al colmo delle sue potenzialità e complesse tecniche di ottimizzazione permettono ad operazioni distinte di convivere all'interno della stessa unità di calcolo contemporaneamente.

Il progresso tecnologico ha incontrato barriere insormontabili nell'evoluzione di un singolo processore, dovute a numerosi vincoli, tra i quali potenza, energia, frequenza, costo e dimensioni. L'ingegno umano ha permesso di valicare questi limiti con un semplice espediente: *replicare* anziché *potenziare*.

Le complesse e costose unità sequenziali sono state affiancate e ben presto rimpiazzate da processori meno potenti, ma che contengono componenti replicati. Applicazioni che nel passato avrebbero impiegato troppo tempo per essere portate a termine da una singola CPU possono ora essere eseguite da più unità in parallelo, con costi minori e in tempi più contenuti.

Questi sistemi paralleli pervadono oggi la nostra vita, concretizzandosi in quasi tutti i calcolatori moderni. I minori costi e il ridotto tempo di calcolo rappresentano, tuttavia, una sola faccia della medaglia. Laddove il numero delle operazioni in esecuzione contemporaneamente cresce, il risultato finale

perde la sua determinatezza, e la *singola* sequenza di operazioni nota a priori diventa un *insieme* di possibili operazioni tra loro ordinate in modo non deterministico, le quali possono portare a differenti risultati.

La replicazione delle unità nei processori non può portare direttamente ad un migliore uso delle risorse: non esiste nessun metodo efficiente per trasformare automaticamente una sequenza di istruzioni codificate per un'esecuzione sequenziale in altrettante pronte per un'esecuzione parallela. Più operazioni attive contemporaneamente possono interferire tra loro, bloccando il sistema od ostacolando l'ottimizzazione desiderata. I programmatori sono chiamati a riprogettare i loro algoritmi secondo un nuovo modo di pensare.

Il paradigma della *programmazione parallela* incontra le inevitabili difficoltà di transizione da un'impostazione sequenziale ad una concorrente degli algoritmi: se è facile pensare ad una serie ordinata e determinata di passi, da eseguire uno alla volta, è sicuramente più difficile e talvolta impossibile trasformare un algoritmo da sequenziale a parallelo. Qualora ciò sia possibile ed efficacemente realizzabile, non si pensi che una semplice duplicazione delle risorse corrisponda ad un dimezzamento dei tempi di esecuzione: le varie operazioni concorrenti devono essere create, gestite, fermate in caso di conflitti, e in presenza di un numero consistente di questi è chiaro che aumentare il parallelismo di un algoritmo porta più problemi che benefici.

Se a questo si aggiunge che ogni sistema operativo gestisce il parallelismo in modo diverso e che l'*architettura hardware* condiziona notevolmente, in numerose varianti, i tempi di creazione, controllo ed esecuzione di ogni processo, appare chiaro che alle leggi teoriche (come la celeberrima *Amdahl's Law*, introdotta originariamente in [1]) è necessario affiancare un più efficace e pratico strumento di misurazione delle prestazioni, che agisca nel sistema desiderato.

Tale strumento, denominato *benchmark*, è nella sua forma più semplice un programma (o una *suite* di programmi) di uso frequente e di composizione nota, arricchito con alcune funzionalità che permettono di misurare agilmente il suo tempo di esecuzione nel sistema. Benchmark di comprovata affidabilità e di larga diffusione sono utilizzati come *riferimenti* a cui i produttori di hardware si adattano per mostrare alla comunità i punti di forza dei loro prodotti.

Nel panorama odierno, la maggior parte dei benchmark non è modificabile, ne adattabile alle necessità dei programmatori o dei progettisti. Sebbene un programma modello possa essere facilmente riconosciuto dalla comunità scientifica e industriale, è sicuramente evidente l'assenza di un benchmark *personalizzabile*, che riesca a dare un *feedback* immediato all'utente su una sequenza di operazioni decisa ed impostata dallo stesso, e non nota a priori ed immutabile.

Questa limitazione è particolarmente marcata nelle operazioni di lettura e scrittura su disco, che pochi benchmark includono tra le proprie funzionalità e sul quale spesso il progettista di algoritmi non si sofferma, ma altresì

importante, essendo gli *input* e gli *output* dei programmi spesso caricati o salvati al di fuori della memoria volatile del calcolatore.

In questo elaborato l'autore propone un benchmark *innovativo* con la capacità di colmare le lacune descritte in precedenza, abilitando la *modellazione* di un'applicazione caratterizzata da un workload conosciuto. Nello stadio finale del programma, all'utente sarà chiesto di inserire, con una sintassi formale, uno specifico comportamento da simulare (un *taskgraph* arricchito). Questo verrà riprodotto nel sistema e successivamente analizzato, calcolando utili profili temporali che consentono di individuare le prestazioni del workload modellato.

In caso di algoritmi non ancora sviluppati, ma la cui struttura è nota al progettista, è possibile simulare e analizzare il loro traffico tramite il benchmark. A seguito dei risultati mostrati, il programmatore potrà procedere ad un'ottimizzazione manuale del proprio carico di lavoro, nei limiti del possibile, osservando come vari cambiamenti nel suo taskgraph si riflettono nell'ambiente in analisi. Tramite vari tentativi, sarà possibile individuare la configurazione migliore per un determinato algoritmo, calibrando in maniera ottimale i diversi tipi di operazioni necessarie (ad esempio, *letture da disco* o *computazioni intensive*) o il più conveniente numero di processi in parallelo. Il progettista potrà dedicarsi all'implementazione dell'algoritmo desiderato con la struttura migliore emersa dalle analisi prodotte dal benchmark, evitando di modificarla in seguito.

La descrizione formale del taskgraph permette, inoltre, la generazione dello stesso workload su sistemi diversi, consentendone così il confronto a parità di traffico e individuando l'ambiente più adatto ad un'applicazione specifica.

Quanto implementato è un prototipo funzionante del lavoro ideale a cui l'autore aspira, costituente una solida base su cui costruire sviluppi futuri, commentati al termine del documento. Lo strumento è stato programmato con il linguaggio *C*, sfruttando le proprietà del *sistema operativo Linux*, che è stato possibile studiare e analizzare grazie alla sua politica aperta.

L'esposizione si articola come segue:

- IL PRIMO CAPITOLO descrive lo *stato dell'arte* dei principali concetti di questa tesi, ovvero i *benchmark*, i *sistemi paralleli* e i *sistemi distribuiti*, con approfondimenti sulle moderne frontiere raggiunte da essi.
- IL SECONDO CAPITOLO descrive le *nozioni necessarie* alla comprensione del funzionamento di un benchmark, approfondendo in particolare gli elementi rilevanti del *sistema operativo Linux*. Si introducono, inoltre, il framework Apache *Hadoop* e il modello di programmazione *MapReduce*, entrambi in via di sviluppo e di crescente importanza, per preparare il lettore alla validazione dello strumento sviluppato, descritta nel quarto capitolo.

- IL TERZO CAPITOLO descrive il *benchmark realizzato*, illustrandone l'architettura, approfondendo la descrizione dei suoi componenti e dettagliando le analisi da esso prodotte.
- IL QUARTO CAPITOLO mostra la *validazione sperimentale* dello strumento, descrivendo come sia possibile modellare con esso il carico di lavoro di un'applicazione nota e riproducibile, ottenendo dei risultati coerenti con l'esecuzione del programma che si è simulato. Prevedendo una futura espansione di *Hadoop*, questo capitolo presenta il taskgraph di un'applicazione sviluppata per tale framework, pubblicamente disponibile, descrivendone un taskgraph semplificato, mostrando come lo si è ottenuto e confrontando il suo profilo con i risultati prodotti dal benchmark.
- LE CONCLUSIONI riassumono i risultati del lavoro, evidenziano i suoi limiti ed illustrano alcuni possibili sviluppi futuri.

# Capitolo 1

## Stato dell'arte

In questo capitolo si descrive la tecnologia più progredita attualmente disponibile relativa a *benchmark*, *sistemi paralleli* e *sistemi distribuiti*, ovvero i concetti che stanno a fondamento del presente lavoro. Per ognuno di questi si farà dapprima una panoramica generale e si daranno alcune definizioni; in seguito se ne analizzerà lo stato dell'arte.

### 1.1 Benchmark

Un *benchmark* è, in informatica, un *programma software* che viene utilizzato per misurare determinati *indici di prestazione* relativi a *determinate risorse* sotto valutazione.

Ad esempio, per misurare le prestazioni di una scheda grafica può essere utilizzato un programma noto per la sua onerosa richiesta di operazioni grafiche, mentre l'efficienza di una CPU può essere testata da un'applicazione con numerosi calcoli e operazioni *floating point*. I due programmi costituirebbero, rispettivamente, un benchmark per la scheda grafica e uno per la CPU.

#### Indici di prestazione e metriche

Il termine *indici di prestazione* è utilizzato nell'informatica per designare gli indicatori di efficienza per una risorsa. Tale efficienza, tuttavia, può essere considerata sotto punti di vista totalmente diversi, e, a volte, tra loro in contrasto. Ad esempio, la *frequenza* di un processore può essere un indice della velocità massima teorica di calcolo, ma la *potenza utilizzata*, indice del consumo della risorsa, ne può risentire in proporzione inversa. Ogni utilizzatore finale è alla ricerca dei propri indici di prestazione di interesse, e spesso un compromesso tra loro è necessario.

I benchmark permettono solitamente il calcolo di almeno due indici molto importanti:

- il **tempo di risposta**, ovvero il tempo che una risorsa impiega a compiere un *determinato lavoro*<sup>1</sup>.
- il **throughput**, ovvero il carico di lavoro che una risorsa riesce a portare a termine in un *determinato tempo*.

Ovviamente, il primo indice indica alte prestazioni per bassi valori dello stesso (il tempo necessario a compiere un'operazione deve essere il minore possibile), mentre il secondo dovrebbe, auspicabilmente, essere elevato.

Come accennato in precedenza, è evidente il contrasto tra i due: un numero maggiore di processi nel sistema migliorerà il throughput, aumentando tuttavia il tempo di risposta, e viceversa. A seconda dell'utilizzatore, l'indice di prestazione di interesse cambia radicalmente. Un utente finale sarà più interessato al primo, in quanto è suo interesse terminare un'operazione il più presto possibile; al contrario, l'amministratore di un centro di calcolo sarà più interessato a servire il maggior numero di richieste contemporaneamente, preferendo il secondo indice.

Gli indici misurati dai benchmark sono spesso chiamati **metriche**, vi è la possibilità che queste siano espresse non nella loro grandezza naturale (ad esempio, il tempo), ma in un *punteggio* derivato e dipendente dal benchmark in uso.

## Composizione

Un benchmark è essenzialmente costituito da un'applicazione in grado di misurare le metriche di interesse parallelamente al suo normale carico di lavoro, avendo cura di interferire il meno possibile con esso per non falsare i risultati finali.

La caratteristica fondamentale di un benchmark è, per definizione, il *consenso* da parte della comunità: deve *essere riconosciuto* come riferimento dall'ambiente che lo utilizzerà e lo analizzerà. Affinché questo avvenga, un benchmark, solitamente, presenta questi tratti:

- è un'*applicazione nota*, ovvero il suo comportamento è chiaro alla comunità, ed è possibile analizzarlo tramite *codici sorgenti* disponibili pubblicamente.
- è un'*applicazione adeguata*, ovvero le sue operazioni sono in larghissima parte adatte al tipo di benchmark in considerazione (ad esempio, in un benchmark dedicato alla CPU vi devono essere pochissime operazioni che richiedano altre risorse, ad esempio il disco fisso).
- è un'*applicazione diffusa*, il che significa che è sensato collocarla come riferimento perché un gran numero di persone la utilizza; pertanto

---

<sup>1</sup>Con i termini *lavoro*, *carico di lavoro* e **workload** si indica l'insieme delle operazioni che un'applicazione esegue nel tempo.



alte prestazioni nell'esecuzione di questo benchmark rappresentano una utilità significativa.

- è un'*applicazione utile*, ossia risolve problemi rilevanti o è utilizzata abitualmente da una comunità.
- è un'*applicazione portabile*, vale a dire funziona in modo identico in tutte le architetture e in tutti i sistemi operativi (o in gran parte di esse o essi).

Spesso ad un *singolo* benchmark si preferisce un insieme di programmi accomunati dalla stessa natura, ma con scopi, strutture e carichi di lavoro differenti: in questi casi si parla di *suite di benchmark*.

## Motivazioni

La nascita dei benchmark è dovuta principalmente alla necessità di consumatori, produttori e ricercatori di poter *creare uno standard* nella misura *imparziale* delle prestazioni di una risorsa. Come si è accennato nell'introduzione, i semplici dati tecnici non costituiscono necessariamente la prova di un'architettura potente o efficace, e le variabili che condizionano le prestazioni di un programma, oggi come trent'anni or sono, sono troppe e non sempre predicibili.

Si immagini di avere un programma con determinate caratteristiche salienti, ad esempio la presenza massiccia di calcoli in virgola mobile. Nel momento in cui si vuole scegliere un prodotto (ad esempio una CPU) adeguato al programma, la prima scelta potrebbe ricadere su un esemplare composto da numerose unità *floating point*. Tuttavia questa risorsa potrebbe essere meno prestante di un'altra con meno unità dedicate, ma con una diversa organizzazione della *cache* o con frequenza maggiore.

Di fronte a questa scelta, la cosa migliore da fare sarebbe provare il proprio prodotto direttamente su tutte le potenziali scelte e misurare gli indici di prestazione desiderati, così da avere la sicurezza della scelta ottimale. Sebbene tale eventualità sia preclusa da tutti i venditori, si può procedere ad individuare un benchmark *noto e diffuso*, corrispondente il più possibile al proprio programma in termini di *tipo* e *flusso* di operazioni effettuati, quindi provarlo nell'architettura.

I siti web di benchmark riconosciuti ospitano i risultati che i produttori hanno ottenuto tramite l'esecuzione del benchmark, vincolata da regole rigorose imposte dallo stesso, nelle proprie architetture e che hanno deciso di pubblicare. La diffusione e la necessità di questi programmi ad oggi è tale che questa competizione è divenuta fondamentale anche per i produttori, i quali sono sempre più incentivati ad *adattarsi al benchmark* e a pubblicizzare le prestazioni delle loro creazioni.

### 1.1.1 Il consorzio SPEC

*«An ounce of honest data is  
worth a pound of marketing hype»*

---

Motto del consorzio SPEC

Gli sforzi nel raggiungimento di uno standard comune, ai quali si è accennato in precedenza, sono in gran parte dovuti al consorzio Standard Performance Evaluation Corporation (SPEC), fondato nell'ottobre del 1988 con il preciso scopo di creare un riferimento comune nella misura delle prestazioni informatiche. Il consorzio, inizialmente costituito da sole quattro aziende fondatrici, consta attualmente di più di cento membri tra corporazioni industriali e istituti di ricerca.

La prima suite di benchmark universalmente riconosciuta come tale dalla comunità scientifica, denominata **SPEC Release 1**, fu rilasciata nel 1989 dal consorzio. Prima di questa, sul mercato erano disponibili solo timidi “tentativi” di benchmark isolati, presto divenuti inadeguati (si segnalano ad esempio *Dhrystone*, *Linpack* e *Whetstone*). Grazie ai rigidi criteri di selezione, alla partecipazione sostenuta delle maggiori corporazioni industriali e alla trasparenza, SPEC riuscì nei suoi intenti, affermandosi come garante dello standard prestazionale.

Ancora oggi, lo stato dell'arte nel campo dei benchmark è rappresentato dalle ultime *release* delle suite SPEC. In questa sezione si descrive la loro composizione e il loro funzionamento.

#### Le suite di benchmark SPEC

SPEC Release 1 è solo la prima suite rilasciata dal consorzio dedicata alla misurazione delle prestazioni di CPU. Ad oggi i lavori di SPEC hanno portato alla creazione di più suite di benchmark, ognuna con differenti scopi. Nella tabella 1 è data una loro panoramica.

In pieno rispetto dei principi enunciati a pagina 6, SPEC non produce in proprio alcun benchmark, bensì si dedica a raccogliere e a selezionare applicazioni *esistenti* che potenzialmente potrebbero essere candidate per una suite. Prima di una nuova *release*, chiunque detenga i diritti di un programma può proporre il proprio applicativo per l'inclusione nella prossima versione del pacchetto. Il potenziale benchmark inizia un processo di approvazione strutturato in due fasi, al termine del quale viene eventualmente sancita la sua presenza nella suite.

SPEC garantisce la portabilità dei benchmark su più sistemi, risolvendo i relativi problemi nelle *bench-a-thon*, sessioni intensive di analisi e programmazione in cui tutti i membri del consorzio collaborano verso il raggiungimento di un prodotto finale portabile e il più possibile uguale in tutti i sistemi.

**Tabella 1:** Elenco delle principali suite di benchmark SPEC attive ad oggi.

Nome	Campo di interesse
SPEC CPU2006	CPU – Contiene benchmark dedicati sia a computazioni in aritmetica intera, sia in virgola mobile.
SPECaps	Composizione grafica – Contiene più benchmark per applicativi specifici (ad esempio <i>3D Max</i> o <i>SolidWorks</i> ).
SPEC MPI2007	CPU e floating point in sistemi distribuiti, tramite l'uso della <i>Message Passing Interface</i> .
SPEC OMP2012	CPU e consumo di energia, tramite l'uso del linguaggio <i>OpenMP</i> .
SPEC jbb2013	Prestazioni di server ospitanti applicazioni <i>Java</i> .
SPEC sfs2008	Network File System.
SPEC power_ssj2008	Potenza dissipata.
SPEC sip_infrastructure2011	Prestazioni di telefonia e <i>VoIP</i> .
SPECvirt_sc2013	Virtualizzazione.

Come si è accennato a pagina 7, la diffusione e l'efficienza di questi applicativi è tale da incentivare i produttori a migliorare in primo luogo proprio le prestazioni riportate dai benchmark. L'innovazione e la ricerca possono essere seriamente condizionate dalle suite esistenti, le quali costituiscono la prova dell'efficienza di un'architettura o di una risorsa, fintanto che rimangono valide.

Per tale motivo, una suite deve essere *mantenuta* nel tempo ed infine *rimodernata* per stare al passo con le più recenti innovazioni architetturali. Un benchmark è considerato non più significativo principalmente quando impiega troppo poco tempo per essere portato a termine. In questo caso, infatti, le interferenze nel sistema possono condizionare notevolmente le metriche calcolate dal benchmark, falsando i risultati. Si pensi, ad esempio, che nel 2006, all'uscita della suite SPEC CPU2006, le macchine dell'epoca impiegavano *alcuni giorni* a portare a termine l'intera suite di benchmark, mentre la suite che rimpiazzava (SPEC CPU2000) conteneva dei programmi che erano eseguiti dalle stesse macchine *in meno di un minuto*.

Le suite SPEC sono rilasciate con i codici sorgenti, così da garantire la massima trasparenza e la possibilità di analisi del traffico generato e delle operazioni effettuate, ma le loro licenze d'uso prevedono un costo variabile (dai 500\$ ai 2500\$).

## SPEC CPU2006

Tra le varie suite riportate in tabella 1, in questo documento si è scelto di approfondire la descrizione di SPEC CPU2006, poiché costituisce il riferimento più diffuso e accettato per misurare le prestazioni di CPU in singoli calcolatori.

Oltre al processore, la suite permette di verificare l'impatto sulle prestazioni di *architettura e compilatore*, in quanto spesso opportune *opzioni di ottimizzazione* impostate a quest'ultimo determinano cambiamenti notevoli nelle prestazioni dell'applicazione.

SPEC CPU2006 rappresenta la *quinta versione* della serie dedicata alla CPU dei benchmark SPEC, la quale segue la storica SPEC Release 1 e le successive SPEC92, SPEC CPU95, e SPEC CPU2000. Seguendo la filosofia già introdotta con la prima versione, la suite differenzia i benchmark con computazioni prevalentemente *in aritmetica intera*, da quelli con operazioni prevalentemente *floating point*.

L'insieme dei dodici programmi ad aritmetica intera è chiamato CINT2006, mentre l'analogo insieme dei diciassette programmi con operazioni prevalentemente floating point è chiamato CFP2006. Nella tabella 2 sono elencati i ventinove programmi che compongono il pacchetto completo.

Come si può notare dando un rapido sguardo alla tabella, i benchmark rappresentano una varietà di applicazioni scientifiche e non. I vari programmi svolgono funzioni differenti (anche se, talvolta, la descrizione data in tabella non permette di comprenderlo a pieno) e si rivolgono a *comunità diverse*. Ognuno di essi è, inoltre, caratterizzato da uno specifico tipo di carico di lavoro e sollecita la CPU e la memoria in modo differente. I programmi che utilizzano operazioni in virgola mobile comprendono tutti equazioni o sistemi di equazioni da risolvere e hanno applicazione nei problemi scientifici e ingegneristici. I linguaggi rappresentati dalla suite sono *C, C++ e FORTRAN*. Quasi tutti i programmi inclusi sono *multithread*.

Si può notare che alcuni dei programmi presenti hanno nomi che ricordano famose applicazioni presenti nel panorama informatico. In effetti, come già detto in precedenza, la suite è composta da *programmi diffusi*, pertanto tale coincidenza non deve meravigliare. Le versioni incluse nella suite, se differenti dalle originali, possono essere state modificate, lievemente, per due motivi: migliorarne la portabilità o eliminarne le operazioni non strettamente computazionali, come le letture o scritture da disco. Infatti, nessuno dei benchmark proposti in SPEC CPU2006 ha operazioni rilevanti che non utilizzano la CPU.

I programmi che compongono il pacchetto possono essere eseguiti singolarmente, ognuno con i propri *input*. Tuttavia, in caso si vogliano pubblicare i risultati pubblicamente sul sito della SPEC, i possibili input e le regole da rispettare sono molto rigidi e la procedura prevede, in base alle esigenze, l'esecuzione dell'*intero* insieme di benchmark con operazioni aritmetiche intere, l'*intero* insieme analogo con operazioni in virgola mobile, oppure entrambi.

**Tabella 2:** Elenco dei benchmark contenuti in SPEC CPU2006. Il filetto orizzontale separa i benchmark contenuti in CINT2006 da quelli contenuti in CFP2006. Dalla lista sono esclusi i due programmi 998.specrand e 999.specrand, i generatori di numeri pseudo-casuali rispettivamente per programmi in aritmetica intera e in virgola mobile, contenuti nella suite ma non classificabili propriamente come benchmark. Le tre cifre che precedono il nome del programma sono un identificativo univoco, che permette di diversificare programmi aggiornati compresi anche nelle suite precedenti. Ad esempio, gcc è presente fin dalla SPEC Release 1, allora con il nome di 001.gcc.

Nome	Descrizione
400.perlbench	Interprete <i>PERL</i> con alcuni moduli aggiuntivi.
401.bzip2	Compressione e decompressione.
403.gcc	Compilatore <i>C</i> .
429.mcf	Ottimizzazione dello <i>scheduling</i> nel pubblico trasporto.
445.gobmk	Simulatore del gioco <i>Go</i> .
456.hmmer	Ricerca di sequenze genetiche.
458.sjeng	Simulatore del gioco degli scacchi.
462.libquantum	Simulatore di meccanica quantistica.
464.h264ref	Compressione video <i>H.264/AVC</i> per Blu-Ray.
471.omnetpp	Simulatore di rete <i>Ethernet</i> .
473.astar	<i>Path-finder</i> per mappe e terreni.
483.xalancbmk	Convertitore <i>XML</i> in altri formati (come <i>HTML</i> ).
410.bwaves	Simulatore di fluido-dinamica computazionale.
416.gamess	Elaboratore di chimica computazionale quantistica.
433.milc	Simulatore di cromo-dinamica quantistica.
434.zeusmp	Elaboratore di magnetoidrodinamica.
435.gromacs	Simulatore di dinamica molecolare.
436.cactusADM	Elaboratore di problemi fisici di relatività.
437.lelie3d	Elaboratore di fluido-dinamica computazionale.
444.namd	Simulatore di sistemi biomolecolari.
447.dealll	Risolutore di equazioni alle derivate parziali.
450.soplex	Algoritmo del semplice in Programmazione Lineare.
453.povray	Renderer grafico ( <i>ray-tracer</i> ).
454.calculix	Elaboratore di meccanica strutturale.
459.GemsFDTD	Risolutore di equazioni di Maxwell tridimensionali.
465.tonto	Elaboratore di cristallografia quantistica.
470.lbm	Simulatore di fluido-dinamica computazionale.
481.wrf	Elaboratore di previsioni meteorologiche.
482.sphinx3	Motore di riconoscimento vocale.

**Tabella 3:** Tempi di riferimento per tutti i benchmark di SPEC CPU2006.

Nome	Tempo [s]	Nome	Tempo [s]	Nome	Tempo [s]
400.perlbench	9770	401.bzip2	9650	403.gcc	8050
429.mcf	9120	445.gobmk	10490	456.hmmmer	9330
458.sjeng	12100	462.libquantum	20720	464.h264ref	22130
471.omnetpp	6250	473.astar	7020	483.xalancbmk	6900
410.bwaves	13590	416.gamess	19580	433.milc	9180
434.zeusmp	9100	435.gromacs	7140	436.cactusADM	11950
437.leslie3d	9400	444.namd	8020	447.dealll	11440
450.soplex	8340	453.povray	5320	454.calculix	8250
459.GemsFDTD	10610	465.tonto	9840	470.lbm	13740
481.wrf	11170	482.sphinx3	19490		

### Metriche di SPEC CPU2006

Fin dalla prima release della suite SPEC apparve chiaro che un *singolo numero* caratterizzante le intere prestazioni del sistema fosse limitante. Tuttavia, la necessità di facilitare la comprensione dei risultati e la definizione di un unico parametro che, inequivocabilmente, potesse ordinare tra loro prestazioni di architetture diverse, finì per costringere la stessa SPEC a definire il *più ridotto numero possibile* di metriche.

In questa sezione si esamina come SPEC CPU2006 valuta le prestazioni di un sistema tramite l'esecuzione della suite.

Si definisce *tempo di riferimento* di un benchmark il tempo impiegato da una specifica macchina, chiamata *macchina di riferimento*, ad eseguire il benchmark stesso con una determinata configurazione e un determinato input, denominato *input di riferimento*. La macchina di riferimento è generalmente poco potente, in modo che i tempi di riferimento per tutti i benchmark siano alti, come si può vedere dalla tabella 3. La macchina di riferimento per la suite SPEC CPU2006 è una Sun Ultra Enterprise 2, dualcore con CPU da 296MHz e 2GB di memoria.

La quantità

$$\frac{\text{tempo di riferimento del benchmark}}{\text{tempo di esecuzione nel sistema sotto analisi}}$$

costituisce lo *SPECratio* del benchmark. Per definizione, dati due sistemi distinti con i relativi SPECratio, quello *più alto* determinerà il sistema più performante rispetto al benchmark, in quanto il riferimento utilizzato come paragone è lo stesso per entrambi.

Quando tutti i benchmark ad aritmetica intera o tutti quelli utilizzando floating-point sono eseguiti, è possibile elaborare, dagli SPECratio, delle metriche per il tempo di risposta tramite gli indici:

- ***SPECint2006***, ovvero la media geometrica<sup>2</sup> dei 12 SPECratio ottenuti con i benchmark in aritmetica intera.
- ***SPECfp2006***, ossia la media geometrica dei 17 SPECratio ottenuti con i benchmark in virgola mobile.

Per quanto riguarda il throughput, è possibile eseguire i benchmark con un adeguato carico di lavoro, orientato a massimizzare l'utilizzazione del sistema (ad esempio, può venire generato *un processo per ogni thread hardware*). Con questa esecuzione si potrà ottenere un indice per ogni benchmark, chiamato ***SPECrate*** e quindi, similmente a prima, potranno essere derivate delle metriche significative.

Nelle versioni precedenti, lo SPECrate era derivato con una formula precisa che teneva in considerazione il tempo di riferimento del benchmark e il più alto tempo di riferimento tra la tipologia di benchmark considerati. Tuttavia, in SPEC CPU2006 non è fornita alcuna documentazione ufficiale nella quale sia possibile reperire una formula simile. A titolo informativo si definiscono comunque le metriche relative al throughput:

- ***SPECint\_rate2006***, ovvero la media geometrica dei 12 SPECrate ottenuti con i benchmark in aritmetica intera.
- ***SPECfp\_rate2006***, ossia la media geometrica dei 17 SPECrate ottenuti con i benchmark in virgola mobile.

La suite permette, inoltre, di testare la compilazione definendo delle metriche distinte in base alle opzioni date durante essa, per mostrare come abilitare le ottimizzazioni dei compilatori possa essere determinante sulle prestazioni finali. Le quattro metriche definite fino ad ora consentono libera scelta nelle varie *flag* da fornire al momento della compilazione; pertanto si può cercare la migliore ottimizzazione possibile sperimentando con le varie funzionalità fornite dal compilatore. Tali indici sono anche detti “metriche *peak*”.

Al contrario, la variante “*base*” forza opzioni di compilazioni fisse, che non consentono ottimizzazioni al di fuori di quelle standard definite dalle regole di pubblicazione dei risultati. Le quattro metriche relative, denominate ***SPECint\_base2006***, ***SPECint\_rate\_base2006***, ***SPECfp\_base2006*** e ***SPECfp\_rate\_base2006***, sono analoghe alle quattro *peak* e permettono di escludere il compilatore dalla valutazione, concentrando l'analisi sulla sola architettura.

Si evidenzia come un confronto corretto tra architetture diverse debba necessariamente comportare *stesse* condizioni d'uso, in particolare il *medesimo input* deve essere fornito in tutte le architetture. L'input di riferimento è un

<sup>2</sup>La *media geometrica* di  $n$  numeri è definita come la radice  $n$ -esima della produttoria di essi:  $\sqrt[n]{x_1 \cdot x_2 \dots x_n}$

buon candidato a tale omogeneità, poiché è il frutto di un'accurata selezione da parte di SPEC e di una particolare cura nel rappresentare un *workload* significativo e non banale, che consideri ogni porzione del benchmark.

## SPEC CPUv6

Come si è detto in precedenza, a pagina 9, è molto importante che una suite di benchmark sia rimodernata con il passare degli anni per mantenere il suo consenso e la sua efficacia. I sette anni trascorsi dall'uscita di SPEC CPU2006 rendono evidente la necessità di una nuova release della suite e, al momento, SPEC è al lavoro per selezionare i benchmark che comporranno la nuova versione di SPEC CPU2006, chiamata SPEC CPUv6. Il fatto che i lavori siano iniziati nel 2010 prova quanto sia difficile e laborioso assicurare uno standard di alta qualità.

### 1.1.2 La suite di benchmark DaCapo

Sebbene SPEC rappresenti senza dubbio i più famosi benchmark utilizzati dalla comunità scientifica, si vuole dare atto agli sforzi compiuti anche da altri ricercatori nella composizione di suite concorrenti e competitive. In questa sezione si esamina la suite di benchmark DaCapo che, a differenza degli SPEC, è disponibile gratuitamente in rete ed è composta da programmi scritti in *Java*.

Il gruppo DaCapo, composto da professori, ricercatori e studenti di otto università americane, ha iniziato nel 2003 la ricerca di un insieme di programmi adatti a costituire una valida alternativa all'allora presente SPEC jbb2000, ad oggi rimpiazzato da SPEC jbb2013. La prima versione della suite fu rilasciata nel 2006 e successivamente aggiornata nel 2009. La composizione odierna della suite, chiamata formalmente DaCapo-9.12-bach, è riportata nella tabella 4.

Anche per DaCapo è possibile notare come le applicazioni incluse siano note e largamente utilizzate, in ottemperanza dei principi necessari al raggiungimento del consenso da parte della comunità.

Similmente a SPEC CPU2006, i benchmark che compongono la suite sono caratterizzati prevalentemente da operazioni che utilizzano la CPU in modo massiccio, per non condizionare i risultati ottenuti tramite numerose letture o scritture su disco. DaCapo è stata concepita per misurare le prestazioni di *Java Virtual Machine* su client e server e non di applicazioni *CPU-bound* generiche o di loro compilatori.

La metrica fornita dalla suite è il *tempo di esecuzione* di uno o più benchmark, senza alcuna elaborazione aggiuntiva con cui ricavare un punteggio (pertanto, valori più bassi della metrica comportano prestazioni migliori). Solo il tempo di risposta può essere misurato con DaCapo, in quanto non è definita alcuna metrica per il throughput.



**Tabella 4:** Elenco dei benchmark contenuti in DaCapo-9.12-bach. Il programma `eclipse` non esegue alcuna operazione che richieda l'intervento dell'utente, a differenza di quanto il nome potrebbe suggerire. Il programma `daytrader` è un benchmark standard fornito con il server Java Apache Geronimo, che simula un'applicazione di magazzino online.

Nome	Descrizione
<code>avrora</code>	Simulatore di programmi su microcontroller AVR.
<code>batik</code>	Simulatore di grafica vettoriale.
<code>eclipse</code>	Controllo prestazionale della omonima IDE.
<code>fop</code>	Parser e convertitore di file <i>XSL-FO</i> in <i>PDF</i> .
<code>h2</code>	Simulatore di database per applicazioni bancarie.
<code>jython</code>	Interprete di <i>python</i> .
<code>luindex</code>	Indicizzatore di documenti.
<code>lusearch</code>	Ricerca testuale di parole chiave.
<code>pmd</code>	Parser di classi <i>Java</i> alla ricerca di problemi nei sorgenti.
<code>sunflow</code>	Renderer grafico ( <i>ray-tracer</i> ).
<code>tomcat</code>	Simulatore di <i>query</i> per l'omonimo web server.
<code>tradebeans</code>	Esegue <code>daytrader</code> utilizzando <i>JavaBeans</i> .
<code>tradesoap</code>	Esegue <code>daytrader</code> utilizzando <i>SOAP</i> .
<code>xalan</code>	Convertitore di file <i>XML</i> in <i>HTML</i> .

Anche per questa suite, il gruppo DaCapo ha definito degli input standard per uniformare la valutazione delle prestazioni. Con la release vengono forniti tre tipi di input per ogni benchmark (*small*, *medium* e *large*) da utilizzare in base alla necessità dell'utente (ad esempio, il primo tipo di input è raccomandato per i test, mentre il secondo e il terzo lo sono per la pubblicazione dei risultati).

Tutti i benchmark sono *multithread* e l'utente può impostare per alcuni di essi il numero desiderato di *thread principali*, ovvero quelli che "guidano" il flusso operativo del programma. È inoltre possibile aumentare il numero di esecuzioni del benchmark, allo scopo di eliminare possibili interferenze del sistema mediando i risultati al termine dell'esecuzione.

Il sito della suite riporta la notizia, datata febbraio 2012, di una terza release che era prevista per la prima metà dello scorso anno. Al momento non si hanno ulteriori informazioni ufficiali sul progresso dei lavori, che sembrano in fase di stallo. Lo stesso sito del gruppo DaCapo pare non venga aggiornato dal 2008, il che lascia pensare ad una perdita di interesse nel progetto.

## Riferimenti

Il sito della SPEC [30] fornisce numerose informazioni sulla suite. Molte caratteristiche, generali e approfondite, di SPEC CPU2006 sono definite nella

documentazione reperibile nel sito alla pagina [8], che contiene anche numerose nozioni di base utili, come le definizioni date a pagina 5. La stessa pagina contiene anche le metriche della suite, la cui esposizione in questo documento è stata integrata con le informazioni presenti in [7].

Il ridotto impatto delle operazioni di input/output dei benchmark di SPEC CPU2006 è stato studiato da Ye et al. in [40]. I tempi di riferimento della suite e la descrizione della sua macchina di riferimento sono stati estratti da [26].

Dixit, storico presidente della SPEC, descrive in [7] il panorama dei benchmark prima di SPEC Release 1, e fornisce le definizioni di SPECrate e SPECratio, nonché i motivi e le considerazioni sulla loro creazione. I *benchmark* sono riportati in questo documento e in [16]. [7] esplica, inoltre, il processo di selezione e accettazione dei candidati benchmark in una release.

Henning, in [16] fornisce un'introduzione sulla SPEC e sui criteri di accettazione di un benchmark, focalizzandosi sul processo di valutazione di SPEC CPU2000. Espone inoltre i problemi di portabilità tra diversi sistemi.

I benchmark di SPEC CPU2006 sono riportati in modo essenziale in [17], da cui si è attinto per la redazione della tabella 2. Una esposizione dell'evoluzione storica delle suite SPEC è disponibile, sempre grazie ad Henning, in [15].

La suite di benchmark DaCapo è stata introdotta e analizzata da Blackburn et al. in [3]. Nel documento, gli autori analizzano inoltre le prestazioni della loro suite a confronto con SPECjvm98 e SPECjbb2000. Il sito della suite [5] contiene le informazioni relative ai benchmark che la compongono, alla sua natura multithreading e alle release della suite successive alla prima.

## 1.2 Sistemi paralleli

In questa sezione si vuole presentare una panoramica sull'evoluzione dei calcolatori, esaminando quali tecnologie hanno permesso ai primi esemplari di microprocessori, operanti a pochi Hz, di evolversi nelle moderne architetture con più unità computazionali al loro interno e con frequenza di qualche GHz. Le tecniche introdotte nella prima fase di tale sviluppo, al fine di aumentare le prestazioni dei più vecchi processori, sono presenti anche nei calcolatori di oggi, talvolta con tecnologia immutata; pertanto una descrizione storica del progresso tecnologico è fondamentale per la piena comprensione dell'architettura moderna.

Prima di approfondire il discorso, è bene definire con precisione i termini che verranno utilizzati nel corso della descrizione, ad oggi di uso comune ma il cui significato è spesso confuso. Non sempre è presente una definizione comune e accettata di essi; qui se ne propone una adatta a questo lavoro.

Con il termine *sistemi paralleli* si definiscono tutte le unità computazionali che possono eseguire *più istruzioni contemporaneamente*. Sebbene questi siano oggi molto diffusi, i primi calcolatori erano in grado di svolgere una singola operazione alla volta e sono classificabili come *sistemi sequenziali*.

Una **CPU**, acronimo di *Central Processing Unit*, è un'unità di calcolo indipendente, contenente al proprio interno tutti i componenti hardware necessari per portare a termine un'operazione.

Un **microprocessore** è un singolo *chip* che ospita una o più CPU. Originariamente il prefisso “micro” indicava che le varie CPU erano saldate su un *singolo chip*, a differenza dei primi calcolatori in cui l'unità di calcolo era suddivisa, per via della sua larghezza, su più chip.

Si definiva quindi con **processore** l'insieme di questi, termine che attualmente è diventato sinonimo di *microprocessore*, poiché le dimensioni di una CPU sono molto più ridotte di quelle storiche.

Se un microprocessore ospita più CPU, queste possono condividere alcuni elementi tra di loro (come, ad esempio, la memoria *cache*). Si denota con il termine **core** una CPU presente in un microprocessore assieme ad almeno un'altra CPU, indipendentemente da quante risorse condividano tra loro.

In questa trattazione non si vuole entrare nel dettaglio delle vari elementi che compongono una CPU, a cui ci si riferirà genericamente con i termini *unità* o *componenti*.

### 1.2.1 Sistemi sequenziali

Come è noto, ogni componente elettronico all'interno della CPU è caratterizzato da una certa **latenza**, ovvero il tempo che impiega a portare a termine un'operazione. Tutte le unità di un calcolatore operano ad una certa **frequenza**, scandita dagli *impulsi elettrici* inviati, ad intervalli regolari, da un dispositivo fisico chiamato **clock di sistema**. Tali impulsi sono comunemente chiamati **cicli di clock**.

Ogni componente esegue un'operazione ogni volta che il clock manda il relativo segnale. Supponendo che il clock sia unico e che ogni suo impulso raggiunga tutti i componenti del sistema, ne consegue che la massima latenza all'interno della CPU limiterà la frequenza massima del clock al suo inverso.

Un'operazione, in base al suo tipo, può attraversare diversi percorsi all'interno del processore, utilizzando componenti diversi o richiedendo un accesso alla memoria. Operazioni diverse sono caratterizzate da durate diverse, essendo differenti le latenze dei componenti necessari per esse. Anche il tempo impiegato dai segnali elettrici per viaggiare da un'unità all'altra non deve essere sottovalutato. Ne consegue che sono necessari *più cicli di clock* per portare a termine una singola operazione.

Il progresso tecnologico che ha portato all'evoluzione dei calcolatori fino ai complessi sistemi odierni è dovuto alla continua necessità di elevate prestazioni. Nei sistemi sequenziali è stato possibile progredire in questa direzione costruendo componenti di latenza più bassa, di dimensione ridotte e collocati più vicini tra loro, così da permettere una maggiore frequenza di clock.

### 1.2.2 Instruction Level Parallelism

È possibile scomporre una qualunque istruzione in diverse *fasi*, ognuna delle quali deve essere elaborata da un'unità della CPU per portare a termine l'operazione. Ad esempio, un'istruzione deve essere *letta, decodificata, eseguita* e deve *scrivere il suo risultato*. Ogni "porzione" di istruzione deve essere eseguita in ordine e le unità della CPU che si dedicano a queste sono distinte. Pertanto in un sistema sequenziale il componente che si occupa di leggere l'istruzione rimarrà inutilizzato finché quest'ultima non ha terminato la propria esecuzione, rendendo necessario il caricamento della successiva.

I limiti prestazionali di questi sistemi sono evidenti, rendendo sottoutilizzate le varie unità che compongono un processore. Il naturale miglioramento di questo approccio consiste nel permettere l'esecuzione contemporanea, all'interno della CPU, di più istruzioni. Tutte le tecniche che consentono di portare avanti, *contemporaneamente* e in *una singola CPU* più istruzioni costituiscono il paradigma del *parallelismo a livello di istruzione*, o ***Instruction Level Parallelism (ILP)***.

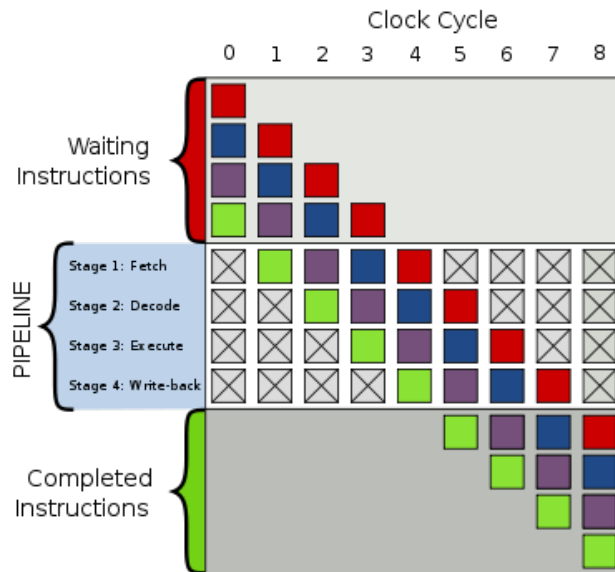
In questa sezione si descrive nei tratti essenziali il meccanismo più diffuso costituente l'ILP nei calcolatori, il ***pipelining*** (talvolta chiamata anche "*instruction pipeline*"). Questa tecnica permette di assimilare le varie unità all'interno della CPU a componenti di una catena di montaggio. Nell'esempio dato in precedenza, l'unità che permette di caricare un'operazione nel processore può, al ciclo di clock successivo, caricarne un'altra, poiché l'istruzione precedente si trova ora nell'unità di decodifica. Al terzo ciclo, la prima istruzione caricata verrà eseguita dalla corrispondente unità, la seconda verrà decodificata ed una nuova istruzione verrà caricata.

Con ***pipeline*** si indica l'insieme delle componenti di una CPU a cui viene applicato il pipelining. Ognuna di queste unità costituisce uno *stadio* della pipeline corrispondente ad una fase in cui le operazioni sono suddivise. Un'istruzione deve attraversare ordinatamente *ogni stadio* per essere elaborata fino al suo completamento.

Supponendo che ognuna di queste venga portata a termine in un ciclo di clock, è possibile in questo modo eseguire completamente un'operazione ad ogni ciclo, sebbene ognuna di esse abbia bisogno di più cicli per essere elaborata. Un'illustrazione del funzionamento della pipeline è visibile nella figura 1.

L'ILP non consente di migliorare il tempo di esecuzione di un'istruzione, bensì il throughput. Il risultato è comunque un sistema più veloce, in quanto dal momento in cui tutti gli stadi sono "pieni" la singola CPU può lavorare al massimo delle sue possibilità.

L'ILP non replica alcun componente all'interno della CPU. Il parallelismo che consente di ottenere è solo una *percezione* e non il risultato di più unità uguali presenti nel processore. Le istruzioni degli algoritmi sequenziali possono



**Figura 1:** Funzionamento di una pipeline. Nell'esempio, un'istruzione è composta da 4 fasi: la *fetch* permette il suo caricamento all'interno della CPU; la *decode* consente la comprensione del tipo di operazione da eseguire, dei suoi operandi e di dove il risultato verrà scritto; la *execute* esegue materialmente l'istruzione e la *write-back* scrive il risultato dell'istruzione nel registro o area di memoria individuati durante la seconda fase. I quadrati riempiti con colore diverso rappresentano differenti istruzioni. Si può notare come il meccanismo della pipeline consenta idealmente, dal quinto ciclo, di terminare un'istruzione ad ogni ciclo di clock evitando il sotto-utilizzo di componenti della CPU. L'immagine è ricavata da [38].

beneficiare della pipeline, dato che le diverse operazioni verranno ordinate e quindi "accodate" una dietro l'altra.

Poiché "a regime" è possibile terminare un'istruzione ad ogni ciclo di clock, il miglioramento della sua frequenza è fondamentale per ottenere delle prestazioni elevate. Come già detto, questa è determinata dal componente più lento all'interno della CPU. L'avvento della pipeline ha quindi trasformato le istruzioni in modo da essere composte da un gran numero di fasi, così che ognuna di queste possa essere gestita da un'unità più semplice, con meno latenza e quindi meno penalizzazione sulla frequenza del processore.

Quanto detto fino ad ora rappresenta il funzionamento *ideale* della pipeline. Nella realtà, l'esecuzione delle istruzioni è rallentata dai *conflitti* che sorgono in essa. Ad esempio, l'istruzione seguente ad un *salto condizionale* non è nota fino all'esecuzione del salto stesso; alcune istruzioni possono trovarsi, in stadi diversi, a dover accedere alle stesse risorse (ad esempio lo stesso registro) o ancora un'istruzione può aver bisogno di un dato che al momento non è ancora stato calcolato dalla CPU.

In aggiunta, le unità aritmetiche (come i componenti che eseguono moltiplicazioni o aritmetica in virgola mobile) e la memoria volatile sono lente, e spesso i progettisti delle architetture scelgono di ammettere per il loro funzionamento più cicli di clock per non penalizzare l'intera frequenza del sistema. Tale attesa si riflette in un peggioramento delle prestazioni ideali del processore.

Le tecniche dell'ILP hanno permesso progressi significativi nel risolvere i problemi sopracitati, la cui descrizione non è però oggetto di questo documento. Si vuole comunque segnalare che, nonostante le nuove tecnologie introdotte, l'ILP ha raggiunto il suo limite attorno al 2005, poiché non si è riusciti a trovare soluzione ad alcune problematiche fondamentali, in particolare:

- Incrementare la frequenza comporta l'aumento della *potenza dissipata* dai processori in modo non lineare. Alcuni studi<sup>3</sup> mostrano che dimezzare il tempo di esecuzione corrisponderebbe ad un aumento di otto volte della potenza. Le elevate frequenze dei processori moderni, rese possibili dalle pipeline, comporterebbero un sempre meno giustificato aumento dei costi operativi.
- Il numero delle fasi in cui suddividere un'istruzione corrisponde al numero delle unità in cui è necessario suddividere la pipeline. Al crescere di esse, la logica di controllo del processore e le risorse ausiliarie necessarie al passaggio dei dati da un'unità all'altra aumentano di numero e complessità, contribuendo anch'esse al crescere della potenza necessaria.
- L'accesso alla *memoria volatile* è necessario per molte istruzioni. Come già accennato in precedenza, tale accesso impiega diversi cicli e aumentare la frequenza del processore è infruttuoso se la velocità della memoria rimane uguale. Sebbene nei processori moderni le cache (si veda a pagina 25) consentano di accedere sempre meno alla RAM, è evidente che le dimensioni limitate di esse possano risolvere questo problema solo in parte. La velocità della memoria è rimasta praticamente costante negli ultimi anni, limitando, a sua volta, l'aumento della frequenza dei processori.
- Le unità della pipeline hanno raggiunto una maturità tale da rendere difficile una loro ulteriore semplificazione con relativo possibile aumento di velocità e frequenza.
- Le tecniche dell'ILP che affiancano la pipeline nel risolvere i conflitti hanno raggiunto il loro limite: è sempre più difficile trovare delle soluzioni che rallentino il meno possibile l'esecuzione delle istruzioni.

---

<sup>3</sup>Si veda, ad esempio, [18]

Nonostante i limiti raggiunti, le tecniche ILP esistenti sono molto efficienti e sono implementate in tutti i processori contemporanei, in aggiunta ad altre tecnologie che verranno esaminate nelle prossime sezioni. A prova della grande efficacia di questo paradigma, il processore **Pentium IV** di Intel, prodotto nel 2003 e destinato al mercato domestico, consente con le sole tecniche ILP di raggiungere la frequenza di 3,4GHz. Le istruzioni dei processori contemporanei sono composte da una ventina di fasi (ad esempio, l'architettura **Nehalem** di Intel ne ha fino a 24, e il **Pentium 4** precedentemente nominato ne aveva 31), anche se sono stati costruiti alcuni elaboratori con scopi specifici composti da *migliaia* di fasi, come ad esempio il **Xelerated Xelerator X10q**. Un tale numero di fasi è giustificato dal tipo di applicazioni (prettamente sequenziali) per cui il calcolatore è ottimizzato.

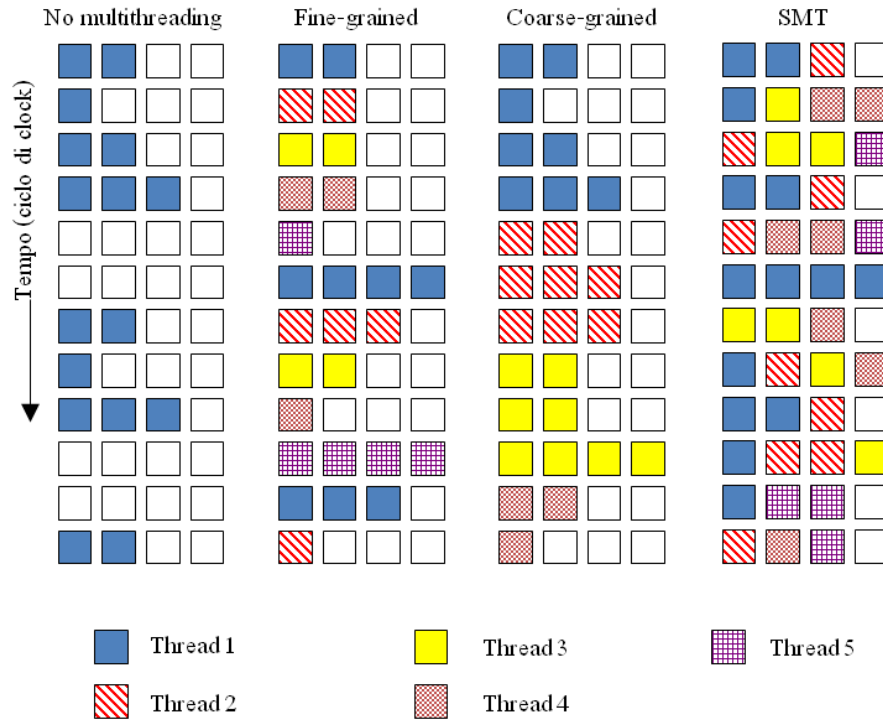
### 1.2.3 Thread Level Parallelism

I limiti dell'ILP hanno evidenziato come sia difficile oggi aumentare le prestazioni di un processore per via, tra tutti, dei conflitti tra le istruzioni. Le tecniche che compongono il paradigma del *parallelismo a livello di thread*, detto anche **Thread Level Parallelism (TLP)**, consentono al programmatore di definire più *sequenze indipendenti di istruzioni*, chiamate **thread fisici** o **thread hardware** (da non confondere con l'analogo concetto di *thread* presente nei linguaggi di programmazione; in questo capitolo si userà il termine *thread* come abbreviazione di *thread hardware*).

Differenti thread hardware, per costruzione, non presentano conflitti tra loro. L'idea principale del TLP è di eseguire *un determinato thread* per volta, *cambiandolo* qualora le istruzioni al suo interno generino conflitti, abbiano bisogno di tempo per il loro completamento (ad esempio perché si deve aspettare che un accesso alla memoria vada a buon fine) oppure, più in generale, vi siano ostacoli al mantenimento di un elevato throughput. Il cambio di thread è chiamato **thread switching**.

Il TLP consiste principalmente di tre tecniche, in alternativa tra loro:

- Il *multithreading temporale di grana grossa*, detto anche **coarse-grained multithreading**, consente la convivenza nella pipeline di *un solo thread* alla volta. Un thread switching avviene quando un'operazione all'interno della pipeline incorre in ostacoli di tempo significativi, ad esempio superiori ad un determinato numero di cicli di clock, e consiste nello svuotamento completo della pipeline con conseguente caricamento del nuovo thread.
- Il *multithreading temporale di grana fine*, anche chiamato **fine-grained multithreading**, permette la *presenza* nella pipeline di *più thread*, ma l'*esecuzione* di *un solo thread* per volta. I thread non eseguiti sono ospitati in strutture apposite installate nel processore, che servono a mantenere il loro stato durante il periodo di non esecuzione. Un



**Figura 2:** Confronto tra i vari tipi di TLP. A sinistra, il termine *No multithreading* denota architetture in cui non è presente alcun meccanismo di TLP. Il tempo scorre in verticale, dall'alto verso il basso. I vari quadrati denotano gli stadi della pipeline, che in figura sono quattro. Si può notare come nel *fine-grained* e nel *coarse-grained* multithreading tutti gli stadi della pipeline eseguano istruzioni dello stesso thread in un certo istante di tempo, mentre nel *simultaneous multithreading* più thread possono essere eseguiti in differenti stadi della pipeline contemporaneamente. L'immagine è stata rielaborata da [28].

thread switching avviene ad ogni ciclo e consiste nel cambio di thread in esecuzione in *ogni stadio* della pipeline.

- Il *multithreading simultaneo*, o ***Simultaneous MultiThreading (SMT)***, consente in ogni momento di eseguire thread differenti in diversi stadi della pipeline.

Un confronto tra i vari tipi di TLP è riportato in figura 2.

I differenti tipi di TLP modificano l'architettura del processore più o meno intensamente, introducendo *repliche* di alcuni componenti della CPU, quali i registri dati, e nel caso del SMT modificando l'unità di *fetch* in modo che sia in grado di caricare più istruzioni alla volta. Le unità devono poter consentire un cambio di thread molto veloce ed è necessario assicurare una politica di



thread switching opportuna, in modo da non favorire alcun thread rispetto ad un altro. In ogni caso, nessuna unità computazionale è replicata.

Tutte e tre le tecniche descritte consentono un miglioramento delle performance del calcolatore, andando effettivamente a *nascondere* la latenza che senza TLP sarebbe necessario aspettare. Tutto ciò ad una complessità contenuta e con aumenti di potenza dissipata considerati accettabili da molti produttori e consumatori.

Oggi giorno larga parte dei processori presenti sul mercato adotta una delle tecniche di TLP descritte in precedenza. Intel ha adottato il SMT migliorandone le funzionalità in una tecnologia proprietaria chiamata **Hyper-Threading**. Tutti i processori Intel in produzione negli ultimi anni la implementano, consentendo ad ogni CPU di lavorare con due thread diversi nella pipeline. Altri processori, come gli SPARC di Sun Microsystem, preferiscono implementare il fine-grained multithreading con un numero elevato di thread per ogni CPU (arrivando fino a 128). AMD, invece, non implementa alcuna tecnologia TLP nei propri prodotti.

Si noti che ciascuno dei differenti tipi di multithreading mostra condizioni d'uso ottimali con un diverso numero di thread. Ad esempio, **Hyper-Threading** è composta da soli due thread per CPU. Ciò perché sperimentalmente il SMT con più di due thread fisici aumenta significativamente di complessità senza mostrare un corrispondente aumento delle prestazioni, al contrario del fine-grained multithreading.

#### 1.2.4 Sistemi multicore

Le problematiche dell'ILP menzionate a pagina 20 sono state solo in parte risolte dal TLP, in quanto i vincoli di frequenza, memoria e potenza non possono essere superati da queste tecniche. ILP e TLP si rivolgono a singole CPU, e, poiché queste sembrano aver raggiunto i loro limiti, il passo successivo è rappresentato dalla loro *replicazione*.

I **sistemi multicore** sono architetture che ospitano, all'interno dello stesso processore, più CPU (chiamate, come già detto in precedenza, *core*). Questi sistemi rappresentano la realtà commerciale dei nostri giorni; i calcolatori domestici e i server presenti sul mercato sono in larga parte formati da processori con più core, ognuno di questi accompagnato da tecniche ILP e TLP.

La moderna tecnologia ha permesso di ridurre sensibilmente lo spazio occupato dai *transistor* che compongono i vari componenti di una CPU. Grazie a ciò, più core possono essere integrati sullo stesso chip. La vicinanza tra i componenti rende possibile, quindi, la condivisione di alcune risorse e la comunicazione ad alta velocità, dato che la distanza che i segnali elettrici devono percorrere nei circuiti è minore. La condivisione delle risorse può essere più o meno presente in base alle scelte architetturali del progettista.

La replica delle unità a disposizione dell'utente permette più sequenze indipendenti di istruzioni in esecuzione parallela. Mentre con il TLP il parallelismo era solo apparente (in quanto *una sola istruzione* poteva essere completata ad ogni ciclo di clock), la replicazione fisica delle unità consente il *parallelismo reale* delle istruzioni. Tale parallelismo deve, similmente alla definizione dei thread fisici, essere imposto dal programmatore.

## Architetture

La composizione dei processori, in termine di numero di core e thread, dipende dal tipo di workload per cui questi sono progettati. Applicazioni domestiche raramente hanno un alto grado di parallelismo<sup>4</sup>, pertanto il mercato dei calcolatori domestici è caratterizzato da processori con pochi core e thread, ma altamente performanti, in modo da massimizzare le prestazioni di istruzioni sequenziali. Al contrario, i server devono ospitare un gran numero di applicazioni indipendenti provenienti da più utenti contemporaneamente. La loro indipendenza rende possibile eseguirle in parallelo, motivando la diffusione di server con tanti core e thread, ma di prestazioni più contenute, che possono favorire un alto grado di parallelismo.

Un esempio evidente di come il tipo di lavoro condizioni l'architettura ottimale è rappresentato dai processori delle schede grafiche, chiamati anche Graphics Processing Unit (GPU). Il tipo di operazioni di cui la grafica è composta consente per natura un elevato numero di istruzioni svolte in parallelo e di struttura nota, al punto che le GPU sono composte da *migliaia* di core a basse prestazioni. Tali architetture sono anche chiamate *manycore* a ricalcare la loro caratteristica, ma la loro trattazione esula gli scopi di questa revisione.

Si citano, per esemplificare lo stato dell'arte, i prodotti d'avanguardia di alcune famose imprese produttrici di processori. Intel propone, per il mercato domestico, la serie Core i7 Extreme, che comprende fino a 6 core, operanti a 3,50GHz, ciascuno con 2 thread in SMT; AMD propone, invece, la serie FX, che opera fino ad una frequenza di 5GHz senza però supportare il multithreading. Per quanto riguarda i server, la linea Intel Xeon comprende prodotti da 10 core con 2 thread ciascuno, operanti a 2,26GHz, mentre gli AMD Opteron hanno fino a 16 core, con frequenza massima di 3,50GHz. Si noti come le architetture differiscano sensibilmente tra i due mercati, a testimoniare il diverso carico di lavoro per cui sono progettate.

## Eterogeneità

Nei sistemi *general purpose*, costruiti per scopi generici, solitamente i vari core sono uguali tra loro e posti allo stesso livello gerarchico (ogni CPU

---

<sup>4</sup>Con *grado di parallelismo* si intende il massimo numero possibile di istruzioni eseguibili in parallelo.

può svolgere lo stesso lavoro di un'altra CPU). Queste architetture sono chiamate *omogenee*, a differenza delle *eterogenee* in cui, di solito per operare con applicazioni specifiche, alcuni core hanno caratteristiche diverse da altri.

La differenza nella composizione dei core permette di delegare alcuni tipi di operazione alle CPU più ottimizzate per questo carico di lavoro, guadagnando in efficienza e risparmiando energia. Ad esempio, le *console di gioco* e i *sistemi di telefonia* preferiscono architetture eterogenee in quanto il carico computazionale è ben definito, a differenza di server e calcolatori domestici in cui sono utilizzate prevalentemente architetture omogenee, in mancanza di un preciso tipo di operazione da svolgere.

Esempi di architetture omogenee sono i processori citati nel precedente paragrafo, mentre una famosa architettura eterogenea è adattata dal processore Cell, sviluppato da Sony/IBM/Toshiba e famoso per essere stato incluso nella console PlayStation 3. Cell è composto da otto processori utilizzati per i calcoli aritmetici e da un nono, delegato al controllo e al coordinamento delle operazioni, e di composizione diversa dai primi.

## Gerarchia della memoria cache

In informatica con il termine *cache* si intende una memoria ad alta velocità di accesso in grado di memorizzare temporaneamente alcuni dati. Nei processori è posta tra la CPU e la memoria volatile e viene consultata prima di ogni accesso a questa. In caso il dato cercato risieda in cache, è possibile evitare l'accesso alla relativamente lenta memoria RAM, risparmiando sensibilmente tempo.

In questa sezione non si vogliono elencare tutte le problematiche relative all'uso della cache, i metodi di scrittura e i protocolli necessari per assicurarne la coerenza, sebbene siano indubbiamente importanti; si vuole invece presentare una panoramica di come tale memoria viene utilizzata nei sistemi multicore.

Il punto di forza della cache è la sua grande velocità, a scapito della sua capacità: tale compromesso è dovuto a vincoli di costi e di tecnologia interna ad essa, con prestazioni diverse anche per piccole differenze di capacità. Nei sistemi multicore è tipico collocare tra il processore e la memoria volatile una serie di memorie cache, ognuna con capacità sempre più grande (ma anche con tempo di accesso sempre maggiore). Il dato a cui si vuole accedere, teoricamente contenuto nella RAM, viene dapprima controllato nella prima memoria cache. In caso di presenza (*cache hit*) gli accessi alle memorie cache successive e alla memoria volatile vengono evitati, mentre in caso di assenza (*cache miss*) viene consultata la successiva memoria cache, di velocità inferiore, ma di dimensione più grande (e quindi, con più speranza di trovare il dato). Il meccanismo si ripete fino a che il dato non è trovato o fino all'esaurimento delle memorie cache.

L'organizzazione delle cache, come descritta in precedenza, è chiamata *gerarchia di memoria*. Nei processori moderni le memorie cache consultate prima della RAM sono tipicamente tre e prendono il nome di *L1 cache*, *L2 cache* e *L3 cache* (la prima di queste è anche la prima ad essere consultata).

Vale la pena di osservare come i sistemi multicore pongano al progettista la scelta tra condividere una cache tra tutti i core del sistema oppure replicarla in ognuno di essi, in modo che solo un singolo core abbia possibilità di accesso. Nel primo caso si parla di memoria cache *condivisa*, mentre nel secondo di cache *privata*.

La scelta è importante per almeno tre motivi diversi.

Il primo riguarda la velocità di accesso: una cache privata è collocata *prima* dei circuiti di interconnessione tra le CPU, mentre una cache condivisa è situata *dopo*. In caso di miss nella L1 cache, nell'ultimo caso la richiesta di accesso alla L2 cache deve viaggiare oltre i circuiti di interconnessione e deve dividerli con le richieste provenienti dagli altri core, situazione che non accade nel primo caso. Il tempo necessario in caso di miss ne risente in accordo.

Il secondo motivo riguarda l'efficienza delle cache nel memorizzare i dati: cache private dello stesso livello possono contenere gli stessi dati, portando non solo ad uno "spreco" di spazio, ma anche a dei più complicati meccanismi per verificare che le modifiche di un processore nella sua cache si riflettano anche sullo stesso dato memorizzato nelle altre memorie. Ciò non avviene con cache condivise.

Infine, il terzo motivo riguarda la scalabilità: cache condivise devono ospitare dati richiesti da più CPU, pertanto hanno più complessi meccanismi di accesso e i conflitti di memorizzazione dei dati saranno più numerosi.

I produttori di sistemi multicore hanno proposto soluzioni diverse per i loro prodotti. Tipicamente, la L1 cache è privata in tutte le architetture, la L3 cache è condivisa mentre la L2 cache può essere condivisa o privata. Un'eccezione è l'Intel Core 2 Quad, composto da quattro core, che presenta *due* L2 cache, ognuna condivisa con due core.

Si riporta, a titolo esemplificativo, la gerarchia di memoria cache della serie Intel Core i7. Le L1 cache e L2 cache sono private, mentre la L3 cache è condivisa. Ogni singola L2 cache ha capacità di 256KB, mentre la L3 cache ne ha 8MB. Ogni L1 cache può ospitare invece 32KB di dati e 32KB di istruzioni. Si ricorda che questo prodotto è rivolto al mercato domestico; le dimensioni raggiungono anche il doppio per i server; in alcuni è presente anche una L4 cache.

## Sottoutilizzazione

«*This plunge into parallelism is  
actually a retreat from even  
greater challenges*»

---

Nota della UC Berkeley

In chiusura di sezione, si vuole sottolineare come l'iniziale scopo di ILP e TLP, ovvero rendere il più possibile utilizzata ogni risorsa all'interno di una singola CPU, sia ora oscurato da una *sovrabbondanza* di risorse rispetto a quelle effettivamente necessarie, soprattutto nel mercato domestico.

TLP e multicore richiedono una progettazione efficiente degli *algoritmi*, che devono essere parallelizzati dal programmatore *manualmente*. Attualmente, infatti, nonostante alcuni tentativi di automatizzare il più possibile l'estrazione di parallelismo dalle istruzioni di un algoritmo, non esiste alcun metodo efficiente in tal senso. Una programmazione oculata è fondamentale per ottenere le migliori prestazioni da un'architettura, ma non è sempre possibile costruire un algoritmo parallelo o può succedere che i programmatori non siano in grado di farlo: la mancanza di un adeguato *paradigma di programmazione parallela* rappresenta il vero limite alle prestazioni dei sistemi odierni.

Questa situazione si concretizza in un'effettiva *sottoutilizzazione* delle risorse disponibili in un calcolatore moderno, l'opposto di quanto accadeva con le architetture sequenziali. Nella maggior parte dei workload domestici, utilizzare a pieno quattro core è difficile e, molte volte, avere a disposizione sei o più CPU risulta inutile per il tipo di prestazione richiesta.

Un'ulteriore evidenza di questo aspetto è la perdita di interesse nella ricerca e nel potenziamento dei singoli processori: ILP e TLP sono trascurati, mentre diventano sempre più importanti le tecniche di condivisione e di collocamento dei vari core all'interno di un singolo chip. Più che un vero progresso nella ricerca, i multicore rappresentano un ripensamento di tecnologie già esistenti (dagli anni '60), spesso riproposte in modo più semplificato e meno performanti. Non a caso in molte architetture vengono proposti core a bassa frequenza, nonostante i notevoli sforzi dell'ILP per agire in direzione contraria.

### 1.2.5 Sistemi multiprocessore

I *sistemi multiprocessore*<sup>5</sup> sono composti, come il nome suggerisce, da più chip separati che ospitano ognuno una o più CPU. L'idea di organizzare separatamente le varie unità computazionali in luoghi diversi non è recente e precede l'invenzione dei sistemi multicore. Con questi ultimi la loro diffusione è diminuita, per tornare in auge solo recentemente.

---

<sup>5</sup>Impropriamente vengono anche chiamati *sistemi multi-CPU*, confondendo gli elementi computazionali di un processore, le CPU appunto, con l'intero processore.

In origine, avere più processori a disposizione era necessario per sistemi in cui un solo processore, all'epoca composto da una singola CPU, non era in grado di gestire un appropriato carico di lavoro. Più processori comportano una gestione della memoria più complicata, in quanto entità separate devono condividere le risorse. Inoltre la distanza fisica tra essi allunga i tempi di comunicazione.

I sistemi multicore hanno oscurato la necessità di sistemi multiprocessore: i primi, infatti, permettono una riduzione dello spazio necessario, con conseguente abbassamento dei tempi di comunicazione e permettono la condivisione delle risorse e dei circuiti, favorendo il risparmio di potenza. Oggigiorno, i sistemi multiprocessore sono utilizzati nelle architetture in cui è necessario ripartire i core in più processori per via del loro alto numero. Pertanto, la loro presenza è più diffusa nei server e quasi assente nel mercato domestico.

Più core in uno stesso chip, infatti, oltre a presentare problemi nell'accesso alle risorse e ai circuiti, hanno problemi fisici riguardanti la loro area e il calore prodotto, e rendono perciò necessaria la loro separazione in differenti processori. Ad esempio, le piattaforme server citate a pagina 24, per il loro elevato numero di core, sono suddivise in più processori (solitamente due o quattro).

Altri esempi di sistemi multiprocessore sono i server Sun Enterprise SPARC T5 e IBM Power 755.

## Riferimenti

Una dettagliata analisi di tutti gli argomenti della sezione è presente in [14]. Dal documento sono state attinte, in particolare, le informazioni relative all'ILP e al TLP. [38] contiene una descrizione essenziale della pipeline, da cui si è proposta la figura 1.

La tecnica del SMT è stata originariamente illustrata in [35] e successivamente in [9]. Da [9] e da [28] è stata rielaborata la figura 2.

Una descrizione approfondita sullo stato dell'arte dei sistemi multicore è disponibile in [2], da cui si è ricavata la relativa descrizione in questa sezione, mentre [18] contiene un'analisi dei limiti di ILP e TLP, assieme a considerazioni di livello energetico e ad una descrizione riassuntiva dei vari tipi di parallelismo esistenti attualmente.

### 1.3 Sistemi distribuiti

La sezione precedente descrive l'evoluzione dei calcolatori fino ai moderni sistemi multicore e multiprocessore, mantenendo l'intero meccanismo di calcolo all'interno di una singola macchina. In questa sezione si analizza come sia possibile utilizzare *diversi calcolatori* in modo sinergico per aumentare la potenza di calcolo disponibile.

Con il termine *sistema distribuito* si indica un insieme di macchine distinte, in grado di comunicare una con l'altra, che permettono di utilizzare le loro risorse in modo collettivo. Ogni componente del sistema è una *macchina indipendente e completa*, in grado di lavorare in piena autonomia, senza alcun supporto esterno.

Nel passato la presenza di problemi o operazioni troppo onerose per una singola macchina ha reso necessario l'utilizzo di una molteplicità di queste. Anche oggi, nonostante le notevoli prestazioni dei migliori server, è necessaria la potenza congiunta di più calcolatori per operazioni quali, ad esempio, le previsioni meteorologiche, l'elaborazione cinematografica, l'analisi probabilistica, la rottura dei *cifrari* crittografici e le complesse simulazioni chimiche, fisiche, meccaniche e geologiche.

Tutte queste applicazioni sono caratterizzate da massicci calcoli aritmetici e relativamente indipendenti, che possono, quindi, trovare posto nelle diverse unità dei sistemi distribuiti. Più recentemente, grazie al *cloud* e alla *virtualizzazione* (che verranno illustrati nel seguito di questa sezione) è nata la necessità di raccogliere logicamente un insieme molto vasto di risorse, portando ad un ripensamento di questi sistemi, che comunque rimangono di fondamentale importanza.

Nel resto della sezione si descrivono le più diffuse configurazioni di sistemi distribuiti presenti oggi, e si affiancano alcuni esempi che dimostrano la maturità da essi raggiunta.

### 1.3.1 Cluster

Si definisce *cluster* un sistema distribuito i cui componenti sono *geograficamente vicini*, interconnessi da una *rete locale* e logicamente visibili come *una singola macchina*. I cluster rappresentano un sistema *centralizzato*, ovvero è possibile controllare fisicamente le macchine data la loro vicinanza e l'appartenenza ad una stessa istituzione.

L'obiettivo di questi sistemi è mettere a disposizione una grande potenza di calcolo per *scopi specifici*, la quale sarebbe impossibile o troppo costosa da avere a portata di mano utilizzando sistemi multicore o multiprocessore appositamente progettati. Per i motivi già descritti nella sezione precedente, infatti, superato un certo grado di innovazione, la disponibilità di una grande potenza di calcolo in un singolo elaboratore diventa economicamente svantaggiosa; perciò risulta conveniente replicare intere macchine piuttosto che interi processori o core.

Essendo progettati per essere visti come un singolo sistema dalle altissime prestazioni, le macchine devono essere connesse da una rete ad *elevata velocità*, che motiva anche la vicinanza tra le unità del sistema. Inoltre, per le stesse ragioni, queste sono di solito *omogenee* tra loro, in modo da semplificare il più possibile la loro gestione (anche se configurazioni *eterogenee* sono comunque

possibili). Una macchina del cluster, chiamata comunemente *master*, coordina e smista le operazioni da eseguire.

La visione logica di un unico sistema molto potente è adatta ad applicazioni caratterizzate da un *altissimo* grado di parallelismo. Ad esempio, algoritmi semplici che devono però essere applicati su un grande insieme di dati (come la rottura *brute-force* dei cifrari) sono ottimi candidati ad essere eseguiti su cluster, perché l'indipendenza delle loro computazioni può essere resa in più istruzioni parallele, utilizzando, quindi, un gran numero di CPU.

Data la velocità di connessione, anche algoritmi *estremamente* paralleli con necessità di comunicazione tra le varie CPU possono utilizzare i cluster, a differenza di grid e cloud (si vedano le sezioni successive). Al contrario, applicazioni interattive o con elevate operazioni di input/output non trovano nel cluster un'architettura ottimale, per via delle continue attese e della sincronizzazione necessaria.

Si noti come i cluster tentino il più possibile di imitare sistemi multiprocessore, al punto che la differenza tra i due è talvolta molto sottile, con reti locali sempre più somiglianti a dei *bus* e unità molto vicine. A prova di ciò, spesso alcuni cluster sono stati classificati come *supercomputer*<sup>6</sup>, venendo assimilati ad un'unica componente.

La visione logica unificata è garantita da un apposito protocollo chiamato *Single System Image*, integrato in sistemi operativi appositamente adattati. Si segnala, ad esempio, la presenza dei popolari Windows e Linux anche in versione modificata per cluster, rispettivamente nei pacchetti Microsoft Windows Compute Cluster Server e Linux Virtual Server.

Oggi come in passato, i cluster sono perlopiù proprietari, utilizzati da istituti di ricerca o commerciali per scopi specifici, come l'elaborazione grafica di alta qualità (i cluster a questo dedicati sono chiamati anche *render farm*) o le simulazioni scientifiche.

### 1.3.2 Grid

Al contrario dei cluster, basati su una localizzazione centralizzata dei loro componenti, con *grid* si indica un insieme di macchine *distribuite* anche in luoghi distanti, interconnesse da una *rete locale o globale e condividenti risorse* tra loro.

Mentre l'obiettivo di un cluster è la visione unificata della capacità di calcolo disponibile, quello del grid è di mettere a disposizione un gran numero di risorse a delle comunità, mantenendo ben *separata* ed evidente la loro appartenenza a luoghi differenti.

Le risorse in condivisione non sono necessariamente costituite da CPU, e comprendono anche dispositivi di memorizzazione o periferiche. Per natura, quindi, i vari componenti di un grid sono *eterogenei*: ciò porta, da un lato,

---

<sup>6</sup>Con *supercomputer* si è soliti indicare una *singola macchina* alla frontiera prestazionale contemporanea.



problemi di controllo, ma, dall'altro, permettono una grande scalabilità delle risorse. Non a caso infatti, sono stati installati grid di dimensioni molto vaste.

La loro nascita è dovuta alla necessità, da parte di organizzazioni diverse, di mettere a disposizione le proprie risorse in nuove comunità, in cui ogni associazione affiliata può utilizzare tutte le risorse della collettività. Al contrario dei cluster, in cui tutto è mantenuto all'interno dei propri locali, il grid ammette l'uso condiviso di tutti i macchinari e degli impianti, situati anche in continenti diversi.

Un'organizzazione distinta in un grid è chiamata *virtual organization*. Ognuna di queste ha scopi in comune e non interferenti con tutte le altre, così da motivare e dare vita alla condivisione.

La distanza tra le varie risorse e la loro eterogeneità rendono i grid adatti a tutte quelle applicazioni che non hanno un grande grado di parallelismo e che non hanno bisogno di comunicare tra loro. Ad esempio, i vari processi generati da un algoritmo pensato per cluster possono essere ricostruiti come *programmi indipendenti e sequenziali*, i quali possono più facilmente trovare posto in diversi calcolatori distribuiti nella rete, utilizzando la loro memoria di massa e comunicando solo al termine delle loro operazioni. Anche programmi interattivi o con elevate richieste di input/output, come i *web server*, possono essere efficientemente implementate in un grid.

Le reti che interconnettono le varie risorse possono coincidere con la rete Internet stessa, avendo quindi velocità molto minore rispetto alle corrispondenti reti dei cluster. Similmente a questi ultimi, anche i calcolatori facenti parte dei grid sono spesso poco costosi e facilmente reperibili sul mercato, facilitando la crescita delle dimensioni del sistema.

I grid enfatizzano il problema della *sicurezza* e della *confidenzialità* dei dati, in quanto le informazioni appartenenti ad un'organizzazione sono potenzialmente ospitate in strutture fuori dal controllo della stessa. Questa questione assume tuttora una grande rilevanza, ed è, anzi, stata accentuata dalla rapida diffusione del cloud computing.

La diffusione dei grid è iniziata nella seconda metà degli anni '90 ed è dovuta in particolare ai lavori di Foster e Kesselman, che in [10] ne descrivono l'architettura, per poi dare il via ad un software open-source per la gestione di questi sistemi chiamato Globus Toolkit<sup>7</sup>. Attualmente questo software costituisce lo standard *de facto* per i grid.

L'utilità e la diffusione di questi sistemi è efficacemente dimostrata dal progetto Folding@Home<sup>8</sup> dell'Università di Stanford, che ha istituito un grid per alcuni calcoli chimici, in particolare per il *ripiegamento proteico*. Questi calcoli sono candidati ideali in quanto possono essere scomposti in tante operazioni, indipendenti, ma estremamente numerose. Chiunque può entrare

---

<sup>7</sup><http://www.globus.org/toolkit>

<sup>8</sup><http://folding.stanford.edu>

a far parte del grid, contribuendo volontariamente alla ricerca con i propri macchinari.

Il progetto ha raggiunto una diffusione tale da collezionare una potenza di calcolo pari a 11,4 PFLOPS secondo dati risalenti ad aprile 2013. Considerato che il processore Intel Core i7 980 XE, il più veloce della serie, a 3,3GHz opera a 109 GFLOPS di picco, il grid può essere virtualmente considerato come circa 105 000 i7 operanti a piena potenza e nelle migliori condizioni. Non a caso, la piattaforma è considerata il più veloce *supercomputer virtuale* del momento, intendendo con questo termine un supercomputer rappresentato da un sistema distribuito.

### 1.3.3 Cloud computing

Da circa cinque anni si è iniziato a diffondere nel mondo informatico un termine piuttosto vago, una cui definizione largamente condivisa non è ancora esistente e dalle caratteristiche spesso confuse o, comunque, non ben delineate: *cloud computing*. Si tratta di sistema distribuito in corso di sviluppo e sempre più dominante in tutti i mercati. In questa sezione se ne delineano le caratteristiche emergenti; durante la lettura si tenga conto, tuttavia, che non esiste una rigida classificazione di queste e che spesso questo termine è confuso con i sistemi grid, per motivi che risulteranno più chiari alla fine di questa trattazione.

Alla pagina web [31] sono presenti ventuno definizioni diverse di «cloud computing». Il National Institute of Standards and Technology (NIST), in [25], ne aggiunge un'altra alla lista così come fanno numerosi articoli relativi a questo argomento. In questo testo si è scelto di tralasciare queste e altre definizioni, ritenendo che un'esposizione dettagliata chiarisca meglio i concetti basilari del cloud.

Si inizi con l'immaginare il *cloud computing* come un equivalente dei sistemi grid. Entrambi, infatti, rappresentano un modo per accedere a risorse geograficamente distanti sulla rete, sulle stesse infrastrutture e con la stessa tecnologia. Poi si modifichi la percezione di cloud computing come segue:

- Da più organizzazioni che condividono certe risorse per uno scopo comune, si ha ora una singola organizzazione che possiede più risorse fisicamente distanti, con la necessità di comunicare tra loro facilmente.
- Dai problemi scientifici, con computazioni intensive e opportunamente programmati per essere portati a termine indipendentemente, si hanno ora più applicazioni generiche, con diversi requisiti e di workload non chiaramente definito.
- Dalla possibilità di accedere ad una determinata risorsa del grid, si vuole ora celare il più possibile quale e dove tale risorsa sia localizzata,

spostando l'importanza sul *tipo* e sulle *caratteristiche* di essa e non sulla richiesta di una *determinata* risorsa.

- Dalle applicazioni operanti prevalentemente in memoria volatile e riducenti al minimo la comunicazione tra i vari componenti del grid, si hanno ora grandi moli di dati che viaggiano in rete.
- Dalla condivisione a scopo collaborativo, si ha ora un modello a pagamento, in cui chi ha bisogno di determinate risorse può richiederle e queste possono essere fornite in modo semplice.

I punti precedenti costituiscono le caratteristiche più evidenti del cloud computing: non una definizione di nuove tecnologie e innovazioni; piuttosto un ripensamento ed un uso differente dei sistemi grid, con scopi diversi.

### Utility computing

*«Computation may someday be  
organized as a public utility»*

---

John McCarthy  
1960

La caratteristica più importante del cloud computing è lontana da ogni tecnicismo informatico, ed è il *nuovo modello di business* che lo accompagna. In questo scenario, un utilizzatore che necessita di risorse, siano esse semplici depositi per file o interi server, può rivolgersi ad un'organizzazione attiva nel cloud computing, specificare le caratteristiche volute e ottenerle con facilità. L'infrastruttura esistente consente di creare un *pool* di risorse dal quale attingere nella *misura* voluta, senza preoccuparsi di dove queste siano fisicamente collocate.

Questo modello di business è talvolta chiamato *utility computing*, in quanto la direzione in cui il cloud computing sta spostando il mercato odierno è simile a quello dei servizi pubblici (*utilities* in inglese). Così come per gas, acqua o elettricità l'utenza paga in base ai consumi, anche nell'informatica è ora possibile pagare in base all'uso delle risorse. Questo rappresenta un enorme cambiamento, in quanto, fino ad oggi, un'organizzazione operante nell'informatica doveva dotarsi di *più* risorse di quelle effettivamente necessarie per assicurarsi di non dover cambiare impianti nel breve termine e per far fronte a picchi di richieste.

Ad esempio, un'azienda ospitante web server di siti italiani può aspettarsi differenti intensità di connessioni in base alla fascia oraria e alle festività, ma sarà comunque costretta a dotarsi di sistemi altamente performanti in modo da riuscire sempre a soddisfare le richieste pervenute. Al contrario, il modello dell'utility computing, e quindi del cloud computing, trasforma solo gli esatti consumi in altrettanti costi, con un notevole risparmio.

Inoltre, prima dell'avvento del cloud computing, un'impresa informatica doveva dotarsi di infrastrutture e calcolatori adatti ai programmi che necessitava, anche se destinati ad uso interno. La loro manutenzione e il personale necessario rappresentavano ulteriori costi aggiuntivi. Oggi l'impresa può rivolgersi ad aziende specializzate nel cloud computing, senza più la necessità di acquistare e mantenere macchine dispendiose.

La diffusione di sistemi multicore e multiprocessore a prezzi sempre più bassi e l'esplosione della capacità dei dispositivi di archiviazione rendono possibili economie di scala, diminuendo ancora di più le uscite.

### Astrazione

L'utility computing è ora una realtà grazie all'innovazione e alla diffusione della *virtualizzazione*, che consente di trattare le varie risorse che compongono il cloud<sup>9</sup> come un insieme aggregato di queste, da cui è possibile "estrarre" a piacimento quanto voluto. Ad esempio, se il cloud è composto fisicamente da tre dischi fissi da 2TB ciascuno, è possibile richiedere più spazi di memorizzazione pronti all'uso a piacimento, purché il totale non ecceda i 6TB. O ancora, da dei calcolatori che forniscono in totale 60GB di memoria RAM e 40 core è possibile ricavare più *ambienti* di lavoro, composti ognuno da una differente quantità di memoria volatile e da un numero di core diversi, finché ve ne è la disponibilità.

La tecnica della virtualizzazione verrà approfondita a pagina 38, a cui si rimanda anche per la comprensione di alcuni termini che verranno utilizzati in questa sezione (come *macchina virtuale* o *istanza*).

Con semplici operazioni, è possibile modificare le risorse allocate ad un certo cliente in base alle sue necessità, anche in modo automatizzato. Il *modello dinamico* di creazione e smantellamento delle risorse virtuali permette ai gestori del cloud di avere risorse sempre utilizzate, arrivando a spegnere le macchine eventualmente superflue e ottimizzando la disponibilità operativa.

Inoltre il tempo richiesto per tutto ciò è molto ridotto, essendo possibile, in *pochi minuti*, rendere operativo un ambiente e fornirlo ad un cliente. Le risorse possono in seguito essere riconfigurate, aumentando, ad esempio, lo spazio a disposizione e il numero di core disponibili. Un'azienda può acquistare una macchina virtuale adeguata ai suoi scopi iniziali e chiederne un rapido riassetto se i suoi requisiti dovessero cambiare.

### Servizi

L'astrazione delle risorse ha permesso non solo di fornire al richiedente caratteristiche personalizzate, ma anche il *tipo* di risorsa desiderato. È possibile fornire spazio di memorizzazione, ma anche interi ambienti di lavoro

---

<sup>9</sup>Con il solo termine *cloud* si indicano in modo generico le risorse e le infrastrutture che compongono il cloud computing.

o, ancora, il semplice uso di un'applicazione. Si tratta di uno scenario totalmente impensabile fino a qualche anno fa, in cui le risorse dovevano essere prese "com'erano" ed era impossibile, ad esempio, richiedere l'uso di una piattaforma ERP senza la relativa infrastruttura sottostante.

Il modello di servizi adottato dal cloud computing è spesso descritto distinguendo tre "livelli" di risorse che è possibile richiedere:

- Al livello ***Infrastructure as a Service (IaaS)*** è possibile acquistare interi ambienti equivalenti a macchinari equipaggiati con processori specifici, RAM, disco fisso e ogni altra periferica necessaria. Quanto fornito corrisponde solo virtualmente alla configurazione hardware desiderata, venendo messa a disposizione una *macchina virtuale* (si veda a pagina 39 per maggiori informazioni).
- Al livello ***Platform as a Service (PaaS)*** è possibile avere a disposizione un *ambiente*, ovvero una particolare configurazione di un sistema, non modificabile, ma utilizzabile per lo sviluppo e l'utilizzo di applicazioni. Spesso l'ambiente fornisce il supporto necessario ad un certo tipo di operazioni, ad esempio può contenere database, web server e può essere già ottimizzato per il loro uso.
- Al livello ***Software as a Service (SaaS)*** è possibile richiedere l'uso di applicazioni, generalmente poco configurabili e senza alcun dettaglio sull'infrastruttura sottostante.

Esistono anche delle estensioni del modello, che qui si è scelto di non trattare per non complicare la discussione.

## Data center

La diffusione del cloud computing per le operazioni odierne ha spostato le operazioni di calcolo dai client ai server. La rete Internet è sempre più importante, in quanto è necessario trasferire un'ingente mole di dati dalla propria postazione al luogo in cui l'elaboratore è collocato.

Per questi motivi, i dati con cui è necessario operare sono sempre di più e una loro efficiente organizzazione è divenuta fondamentale. In parallelo, anche la potenza di calcolo deve essere strutturata intelligentemente, in modo da rendere possibili o migliorare alcune caratteristiche importanti nel cloud computing:

- La *vicinanza* dei dati ai processori che opereranno con essi.
- La *alta velocità* della rete tra le macchine situate nello stesso stabilimento.
- La *capacità di migrazione* delle macchine virtuali tra gli elaboratori disponibili nel cloud (si veda a pagina 39).

- L'elevata *disponibilità* dell'infrastruttura, la presenza di meccanismi di backup e la capacità di reagire ad eventi catastrofici.
- La *scalabilità*.

Si usa il termine *data center* per indicare i complessi in cui sono situati macchine, dispositivi di memorizzazione e componenti ausiliari. Una tipica organizzazione di un sistema di larga scala comprende più data center localizzati in continenti e regioni diverse, per permettere la scelta, da parte dell'utente, dell'infrastruttura a lui più vicina e la continuità del servizio in caso di messa fuori funzione di un impianto. All'interno dei data center si usa raggruppare tra loro alcuni server e dispositivi, organizzandoli in *rack*. Ad esempio, un rack può contenere dai venti ai quaranta server, con relativa archiviazione.

### Modello di programmazione e gestione dei dati

I sistemi grid e il cloud sono composti da più macchine logicamente e fisicamente distinte, che rendono necessario l'uso di linguaggi di programmazione orientati a comunicazione esplicita, come MPI (*Message Passing Interface*). Sebbene nei sistemi grid questo possa bastare, l'avvento del cloud computing pone il problema, non presente nei primi, della vicinanza fisica dei dati ai processori che devono elaborarli, su cui l'utente non ha possibilità di controllo per via dell'astrazione introdotta dalla virtualizzazione. Il movimento dei dati da e verso i processori costituisce il collo di bottiglia più evidente dei sistemi cloud di oggi.

Inoltre, la mole dei dati è tale che questi devono essere divisi e ripartiti su più *nodi* dell'infrastruttura, oltre che replicati per ragioni di sicurezza e backup. Queste necessità hanno dato origine a nuovi modelli di programmazione e di gestione dei dati, come *MapReduce* e il *file system distribuito*. Data la loro importanza in questo documento, si esamineranno questi e alcune loro implementazioni nel secondo capitolo.

### Trend evolutivi

È indubbio che il cloud computing, grazie alla sua diffusione, alla facilità e alla rapidità nel fornire le risorse richieste, abbia sempre più spostato l'*onere computazionale* sul cloud, riducendo i terminali degli utenti sempre più all'osso.

Le workstation dell'utenza domestica sono sempre meno potenti e si pone l'attenzione più sulla disponibilità di una connessione di rete globale, che sulle loro prestazioni di calcolo. Alcuni ricercatori ipotizzano che, nel futuro, i personal computer saranno delle semplici *interfacce* di accesso alla rete, con processori e risorse appena sufficienti a tale scopo. Si dice in gergo che i "*thick client*" si stiano trasformando in "*thin client*".

Più realisticamente, è possibile immaginare che il domani vedrà sicuramente una sempre maggiore importanza del cloud computing, ma problemi di sicurezza dei dati, di confidenzialità, di disponibilità della rete e le politiche aziendali possono frenare la sua diffusione. Si pensi, ad esempio, ai dati sensibili, rigidamente regolati dalla legge o alle informazioni segrete aziendali, la cui circolazione è limitata dal management. Per questo, probabilmente, nella prossima generazione di “internet computing” i cloud saranno affiancati da grid privati o cluster, in grado di assicurare un maggior controllo sulla diffusione e gestione delle informazioni aziendali. Si ricordi, inoltre, che i prezzi dei processori domestici sono ormai relativamente contenuti.

Sebbene il cloud computing condivida la sua infrastruttura di base con i grid, beneficiando delle loro ricerche e innovazioni introdotte in quindici anni, oggi rimangono dei problemi non risolti, tra i quali una gestione ottimale della localizzazione e dell’energia consumata dai data center, la disposizione migliore di periferiche di archiviazione e processori, la sicurezza dei dati e il miglioramento dei modelli di programmazione ottimizzati per il cloud.

### **Alcuni prodotti commerciali**

In chiusura di sezione si analizzano alcuni prodotti diffusi attualmente in tutti e tre i livelli descritti a pagina 35, con lo scopo di portare degli esempi dell’importanza e dell’efficienza che i nuovi modelli introdotti dal cloud computing hanno donato all’informatica.

Il più famoso e diffuso esempio di IaaS è sicuramente Amazon Elastic Compute Cloud (EC2). Questo servizio mette a disposizione delle macchine virtuali complete, collocate nel cloud di Amazon e accessibili tramite interfacce appositamente create. Il controllo su queste da parte dell’utente è completo, simulando un server fisico di sua proprietà.

L’azienda mette a disposizione diverse *availability zones*, raggruppate in *regioni*. All’interno della stessa regione, le prime sono situate in luoghi geograficamente distanti, ma interconnesse da un’infrastruttura ad alta velocità, per consentire la rapida migrazione di istanze tra esse e a basso costo.

In EC2 i costi per l’utente si calcolano in base al tempo di effettivo utilizzo di una macchina virtuale (\$/ora) e al traffico dati da e verso essa (\$/GB), in piena aderenza al modello dell’utility computing.

Accanto a EC2, Amazon propone Simple Storage Service (S3), che mette a disposizione memoria di archiviazione. I dati sono strutturati in *objects* (fino a 5GB), a loro volta raggruppati in *buckets*. Questi ultimi possono essere memorizzati in diverse regioni, similmente a EC2. È possibile, ad esempio, posizionare una macchina virtuale di EC2 nella stessa regione dell’archivio dati fornito da S3, per ridurre al minimo le distanze e aumentare il più possibile la velocità.

Google fornisce a livello PaaS **App Engine**, una piattaforma che consente di utilizzare i data center, le strutture e i database utilizzati da Google. A differenza di EC2, l'utente ha solo una limitata possibilità di configurare l'ambiente, non potendo sceglierne le caratteristiche o il sistema operativo. L'obiettivo di **App Engine** è facilitare lo sviluppo di applicazioni web fornendo ambienti già ottimizzati, e non mettere a disposizione un proprio server virtuale da personalizzare. Oggigiorno, i linguaggi supportati sono *Java* e *Python*.

Il livello SaaS è presente e diffuso nella vita quotidiana. Ad esempio un servizio di posta elettronica, come **GMail** di Google, permette di ottenere risorse adeguate senza possibilità di personalizzazione e senza avere informazioni sulla locazione fisica dei dati. Ulteriori esempi molto diffusi attualmente sono **Dropbox** o **Google Drive**, che forniscono spazio di archiviazione accessibile da una semplice applicazione o tramite browser web.

#### 1.3.4 Virtualizzazione

Una descrizione del cloud computing deve essere accompagnata dalla trattazione delle tecniche di virtualizzazione per poter comprendere a pieno come sia stato possibile arrivare ad una così rapida diffusione dei sistemi cloud. In questa sezione si delineano le caratteristiche basilari di queste tecniche.

Per *virtualizzazione* si intende l'insieme di tecnologie e strumenti che permettono di *simulare* una risorsa, come un sistema operativo, una periferica di archiviazione o un'intera configurazione hardware.

Questa tecnica è molto utilizzata nell'informatica, fin dagli albori. Nei sistemi operativi i processi sono indotti ad agire come se fossero gli unici presenti nel sistema e "vedono" la memoria volatile come se fosse interamente a loro disposizione. In questo caso, la virtualizzazione è utilizzata per astrarre la realtà *fisica*, presentando una struttura *logica* della risorsa, non a caso chiamata *memoria virtuale*.

Linux adotta il *virtual file system*, che consente di trattare qualunque dispositivo di archiviazione dati allo stesso modo. Questa illusione consente di applicare le stesse funzioni a prescindere dall'organizzazione dei file sulle periferiche, rendendo altamente portabile il sistema.

I *dischi virtuali* consentono di forzare il sistema operativo a considerare un file, chiamato *immagine del disco*, come fosse un CD o un DVD realmente inserito in un lettore, anche in macchine sprovviste di tale periferica.

Altri esempi di virtualizzazione sono il Virtual Private Network (VPN), un protocollo che consente di creare una *rete locale* su una rete globale, quale Internet, o ancora la creazione di ambienti ristretti logicamente separati dal sistema operativo, come le *sandbox* o il precedentemente accennato *file system distribuito*, che permette di vedere file fisicamente spezzati e memorizzati in più periferiche di una rete come sequenze unite.



Il tipo di virtualizzazione più importante per i sistemi cloud è chiamata *hardware virtualization* e consiste nella simulazione di un'intera configurazione hardware e delle sue caratteristiche fisiche (come la memoria RAM disponibile, il numero e la frequenza dei core, le risorse grafiche e le periferiche collegate, come stampanti o riproduttori audio).

La configurazione voluta viene “estratta” da una risorsa hardware esistente. Come già accennato a pagina 34, è possibile comporre più configurazioni virtuali di differenti caratteristiche su una determinata macchina. Il calcolatore realmente esistente è chiamato *host*, l'ambiente simulato è chiamato *guest* e il software che realizza la virtualizzazione è detto *hypervisor*. Sinonimi di *guest* sono *Virtual Machine (VM)* o *istanza*.

La possibilità di convivenza di più macchine virtuali sullo stesso calcolatore, in parallelo con la diffusione di sistemi ad alte prestazioni a prezzi contenuti, ha consentito la nascita di server ottimizzati per la virtualizzazione e, di conseguenza, per il cloud computing, composti da numerosi core ed elevata memoria volatile.

La macchina virtuale è vista dal sistema operativo *guest* come un reale calcolatore, consentendo di installare su di essa ogni tipo di software senza limitazioni. Ad esempio, è possibile utilizzare la virtualizzazione per avere a disposizione su calcolatori domestici un sistema operativo Linux, virtualizzato in un'istanza da un hypervisor operante su Windows.

La macchina *guest* è visualizzata dal sistema operativo dell'*host* come una singola applicazione, analoga a tutte le altre usuali. Questa visione consente di attuare alcune operazioni molto utili di impossibile applicazione *fisica* su hardware reali.

Ad esempio, è possibile salvare l'*intero stato* di una macchina virtuale su un file. Tecnicamente, si indica con *snapshot* tale stato. Questa operazione, se eseguita ad intervalli periodici, consente di poter conservare copie di backup delle istanze o di poter replicare l'esatta configurazione della macchina su un altro calcolatore.

È anche possibile spostare la risorsa fisica su cui le istanze sono fisicamente virtualizzate con un processo detto *migrazione*. In pochi millisecondi è possibile *congelare* l'esecuzione della macchina virtuale, farne uno snapshot, muoverla e quindi farla ripartire, in modo del tutto trasparente all'utente (salvo un temporaneo arresto dell'istanza).

Dovrebbe apparire ora chiaro perché la virtualizzazione sia, per certi versi, la *tecnologia abilitante* del cloud computing: il suo modello dinamico, la sua astrazione e la facilità con cui permette di modificare le caratteristiche di varie istanze fanno della virtualizzazione uno strumento ideale per il business adottato nei cloud.

Mentre in passato questa tecnologia comportava dei cali di prestazione a causa del passaggio di tutte le operazioni di sistema della macchina *guest* attraverso il sistema operativo dell'*host*, recenti modifiche architetturali introdotte da Intel e AMD ai loro processori, chiamate Intel VT-x e AMD-V,

hanno permesso di evitare tale passaggio, migliorando significativamente l'efficienza dei sistemi virtualizzati.

Esempi di hypervisor disponibili oggi sul mercato sono rappresentati dai prodotti di VMware, con i suoi **Player**, **Workstation** e **vSphere**. Alternative open source sono Oracle **Virtual Box** e **Xen**, che è anche l'hypervisor adottato da EC2.

### **Riferimenti**

Foster et al. descrivono in [11] grid e cloud computing, mettendone in evidenza alcune differenze. Una panoramica sullo stato dell'arte dei sistemi cloud è invece presente in [41]. Da questi due documenti sono state estratte le sezioni relative al cloud computing e alla virtualizzazione.

I sistemi grid sono descritti in [10], mentre la discussione sui cluster è stata elaborata da [29]. [39] contiene un elenco di problemi storicamente risolti con i cluster.

## Capitolo 2

### Nozioni di base

L'obiettivo di questo capitolo è presentare al lettore tutti gli elementi necessari alla comprensione di quanto verrà illustrato nella parte successiva del documento. Il livello di dettaglio fornito per ogni argomento è proporzionale all'importanza di quest'ultimo nell'elaborato, pertanto si daranno sia semplici definizioni sia analisi approfondite, a seconda della rilevanza del tema trattato.

Il benchmark oggetto della tesi è stato implementato utilizzando il linguaggio di programmazione *C*. Nel capitolo non si tratterà di alcun aspetto di programmazione oggetto dei corsi fondamentali di informatica, assumendo che il lettore abbia familiarità con i concetti di *variabile*, *tipo*, *puntatore*, *array*, *struttura*, *floating-point*, *lista* e *lookup table*, la loro definizione in *C*, nonché la gestione delle *funzioni*, il loro uso e il passaggio dei *parametri*. Sarà data per assunta anche la conoscenza di base dei *sistemi operativi*, del loro *kernel* e dell'organizzazione della memoria del calcolatore.

#### 2.1 Sistema operativo Linux

La realizzazione di un benchmark non può prescindere da uno studio approfondito del sistema operativo sottostante. Si è scelto di utilizzare Linux in quanto la sua politica *open source* permette a chiunque di comprenderne il funzionamento di base e il modo esatto in cui comuni operazioni, quali la lettura o la scrittura su disco, vengono compiute.

Ai fini di questo lavoro è fondamentale descrivere la gestione dei processi e la loro tassonomia, le operazioni di input/output e il meccanismo che consente di operare con il tempo. La trattazione di questa sezione è in gran parte estratta dal libro "*Understanding the Linux Kernel*" di Bovet e Cesati [4], cui si rimanda per un riferimento completo al sistema operativo. A seguire si indicheranno le pagine dell'opera in cui l'argomento è approfondito.

### 2.1.1 Gestione dei processi

#### Processi, light-weight processes e thread

In informatica si definisce *processo* [4, p. 88] un'istanza di un programma in esecuzione. Nel momento in cui il sistema operativo inizia l'esecuzione di un'applicazione, vengono create in memoria volatile delle strutture dati opportune che consentono ad esso di poterla gestire. Ad esempio, il kernel deve sapere quali *istruzioni* può eseguire, quali file sono stati aperti dal processo, quali aree di memoria usa per le proprie variabili, lo stato dei *segnali* che può ricevere e così via. Ogni struttura è logicamente distinta dalle altre. Non si vuole qui descrivere con eshaustività ogni informazione necessaria alla vita di un processo, bensì la loro presenza in memoria.

Un qualunque processo può crearne altri, suoi *figli*, eseguendo un'istruzione generalmente nota con il nome di *fork*. La struttura contenente le istruzioni dei nuovi processi è condivisa con quella del *padre*. Ne consegue che essi possono eseguire le stesse istruzioni del processo che li ha creati e la loro prima azione coincide con l'operazione successiva alla *fork*.

Nonostante questa condivisione, tutte le altre strutture in memoria vengono replicate per ogni nuovo processo, in quanto questo può, per via di istruzioni condizionali, intraprendere diversi percorsi nel corso della sua vita, con conseguenti azioni diverse. Ad esempio, i file letti o le variabili allocate da due processi *fratelli* possono essere differenti. Un processo può anche essere *trasformato* affinché esegua un altro programma.

Un chiaro esempio di questo meccanismo è rappresentato dal sistema operativo stesso, costituito inizialmente da un singolo processo, chiamato *idle* [4, p. 127] e creato durante l'inizializzazione del sistema. Il processo *idle* crea sue copie che in seguito eseguiranno programmi differenti tramite trasformazione.

I programmatori possono far sì che i loro programmi creino nuovi processi per sfruttare il paradigma della programmazione parallela, di cui si è parlato a pagina 21.

La creazione di un nuovo processo è ritenuta *onerosa* in termini di tempo e di memoria utilizzati, per via delle risorse ad esso allocate e del tempo che il sistema operativo impiega per fare ciò. In aggiunta, è possibile che alcuni dei processi figli non abbiano bisogno di una struttura dati privata, in quanto operano sulle stesse risorse del padre (ad esempio, sugli stessi file) rendendo la replicazione di queste inutile, oltre che dispendiosa.

Questi motivi hanno portato alla definizione di entità simili ai processi, ma in grado di condividere con il proprio padre la maggior parte delle strutture dati. Ci si riferisce a queste con il termine *thread*. La condivisione rende possibile risparmiare tempo e memoria, molto importante in applicazioni composte da un alto numero di processi in parallelo. Il tipo di thread più famoso è il *pthread*, definito dall'Institute of Electrical and Electronics

Engineers (IEEE) nello standard Portable Operating System Interface for UniX (POSIX).

Un processo può, con una funzione simile alla `fork`, creare un *pthread*. Questi massimizzano la condivisione delle strutture dati, ma non consentono di scegliere quali condividere e quali no e il loro *stato* (si veda in seguito) è lo stesso del processo che li crea. Tutti i *pthread* di un processo fermato o terminato seguiranno quindi la stessa sorte.

Linux implementa un nuovo tipo di entità che consente di fondere l'indipendenza dal padre (tipica dei processi) con la condivisione delle strutture dati (tipica dei thread) aggiungendo, inoltre, la possibilità di scegliere quali di quest'ultime mettere in comune alla loro creazione. Tali entità sono chiamate *light-weight process* per meglio esprimere la combinazione tra le due caratteristiche. Il programmatore può comunicare esplicitamente al sistema operativo il "grado di condivisione" da assegnare ad ogni nuovo light-weight process.

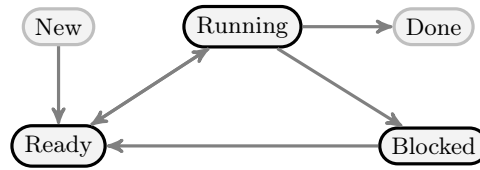
La funzione che consente di creare nuovi *light-weight process* è chiamata `clone`, e verrà descritta a pagina 46. I suoi argomenti consentono di specificare quali strutture il figlio dovrà condividere con il padre. Queste entità rendono la distinzione tra *processi* e *thread* in Linux sempre più sottile. A seguire si userà il termine *processo* per riferirsi indistintamente alle tre entità, poiché ogni proprietà e operazione di cui si tratterà è applicabile ad ognuna delle tipologie sopra descritte.

Ogni processo del sistema operativo è identificato univocamente con un numero, chiamato `pid` (*process identifier*). Linux usa una variabile di tipo `pid_t` per memorizzarlo, sebbene questo sia riconducibile ad un semplice numero intero. Numerose funzioni che operano con i processi utilizzano questo identificatore per determinare con precisione a quale entità ci si sta riferendo.

## Stati

Lo *stato* di un processo [4, p. 90] descrive la sua *condizione* in un certo istante di tempo. I sistemi operativi mantengono lo stato di tutti i processi attivi per determinare quali di questi sono in esecuzione da CPU del calcolatore, quali sono in attesa di certi eventi e quali, pur non essendo in esecuzione, potrebbero esserlo a breve.

Un processo cambia stato frequentemente durante il suo ciclo di vita, a causa di istruzioni da esso eseguite o di richieste da parte del sistema operativo. Il *diagramma degli stati* mostra in che condizione un processo può trovarsi nel tempo, assieme alle possibili transizioni da ognuno di essi. Si noti che gli stati sono in mutua esclusione tra loro. Nella figura 3, in seguito commentata, si riporta un diagramma semplificato, che coglie tutti gli stati utili alla comprensione di questo lavoro:



**Figura 3:** Diagramma semplificato degli stati di un processo.

- Un processo nello stato *running* è in esecuzione da parte di una CPU.
- Un processo nello stato *blocked* non è in esecuzione ed è in attesa di un evento esterno, ad esempio il termine di una lettura o di una scrittura su disco.
- Un processo nello stato *ready* non è in esecuzione e non attende alcun evento. Non può essere eseguito perché tutte le CPU del sistema sono occupate con altri processi.
- Quando l'evento è stato compiuto, un processo *blocked* viene posto nello stato *ready* dal sistema operativo.
- Solo i processi nello stato *ready* sono candidati ad essere eseguiti da una CPU.
- Un processo può uscire dallo stato *running* per due motivi: una lunga attesa causata da una funzione o la fine del “periodo di tempo” che è stato a lui riservato per l'esecuzione. Nel primo caso il suo stato diventa *blocked*, nel secondo *ready*. Maggiori informazioni su quest'ultimo caso sono date in seguito.
- I nodi *new* e *done* rappresentano rispettivamente la creazione e la terminazione di un processo e non sono dei veri e propri stati. La loro presenza nel diagramma è però utile per mostrare che ogni nuovo processo viene creato nello stato *ready* e terminato in *running*.

Si osservi come non sempre un processo si trovi nelle condizioni di essere eseguito, in quanto può essere in attesa di un avvenimento senza il quale non può proseguire.

I sistemi operativi devono eseguire centinaia di processi contemporaneamente anche nei più semplici ambienti di lavoro, adattando il loro funzionamento al numero di CPU presenti nel calcolatore. Nonostante la presenza di più core nei sistemi moderni, non è ovviamente possibile garantire una singola unità di calcolo per ogni processo vivente, pertanto le risorse disponibili devono essere condivise tra essi.

Anche per questo motivo è molto importante distinguere un processo che può essere eseguito da un altro che non ne ha i requisiti. Il mantenimento degli stati consente questa distinzione.

## Lo scheduler

Mentre i processi nello stato *blocked*, finché vi rimangono, non hanno possibilità di esecuzione, tutti gli altri devono contendersi le CPU disponibili. Un algoritmo che determini la *politica* di assegnamento di queste risorse ai processi non è banale, in quanto quest'ultimi sono di natura eterogenea tra loro e ognuno può avere diversi requisiti.

Si è soliti esemplificare questo concetto distinguendo i processi *interattivi* da quelli *batch*. Mentre i primi richiedono un frequente intervento da parte dell'utente, i secondi possono continuare indisturbati la loro esecuzione senza alcuna richiesta all'utilizzatore della macchina. Un esempio di programma interattivo può essere un *editor testuale*, in cui è necessario che i caratteri digitati sulla tastiera vengano prontamente elaborati e mostrati sullo schermo. Se ciò dovesse avvenire lentamente, con ritardi percepibili dall'utente, quest'ultimo sarebbe presto frustato da questo comportamento.

Una politica intelligente deve prontamente permettere l'esecuzione di un processo interattivo nel momento in cui un tasto viene premuto (o più precisamente, un *interrupt* da parte della tastiera viene rilevato) per poi consentire l'esecuzione di processi batch al termine dell'elaborazione necessaria.

L'algoritmo che gestisce la politica descritta in precedenza è chiamato *scheduler* [4, p. 255]. In Linux, nel momento in cui un nuovo processo è scelto per essere eseguito, lo scheduler determina un periodo di tempo massimo per il quale questo può utilizzare le risorse computazionali, chiamato *quanto di tempo*.

Se durante l'intervallo il processo richiede un'azione che comporta l'attesa di un evento, il processo è posto nello stato *blocked* dallo scheduler, che procede, quindi, a selezionarne un altro da quelli pronti per l'esecuzione. In assenza di azioni simili, al termine del *quanto di tempo* il processo è tolto dallo stato *running* e posto in *ready*. In questo modo il sistema operativo riesce a garantire la convivenza di numerosi processi all'interno del sistema.

Ogni processo è caratterizzato da un numero, chiamato *priorità*, che rappresenta l'"urgenza" che l'entità ha di essere eseguita a breve. Le priorità in Linux sono generalmente *dinamiche*, ovvero il sistema le aumenta o diminuisce nel corso del tempo in base a diversi fattori, come il tipo di processo e il tempo dal quale esso non è stato eseguito. Un numero *più basso* di priorità corrisponde ad un'urgenza *maggiore*.

Al momento di cambiare un processo, lo scheduler ne seleziona uno nuovo da eseguire tra quelli con *priorità più bassa* e ne calcola un *quanto di tempo* con una formula ricavata dalla priorità stessa. La sua implementazione è molto complessa e si rimanda a [4, p. 259] per ulteriori approfondimenti. Si noti, comunque, che in periodi di esecuzione diversi, lo stesso processo può ricevere *quanti di tempo* diversi.

Un programmatore *non* ha alcun modo per impostare un'*esatta* priorità per i processi del suo programma, anche se può, volendo, *aumentarla*

(rendendoli quindi *meno* urgenti rispetto ad altri). Un utente con privilegi amministrativi<sup>1</sup> può aumentare o diminuire la priorità di qualunque entità attiva, che comunque potrà essere ricalcolata in seguito dal sistema operativo e quindi cambiata. In nessun caso è possibile decidere *esattamente* il *quanto di tempo* desiderato per un processo.

Ai fini di questo documento, si vuole evidenziare che per utenti senza privilegi di amministrazione in sistemi operativi Linux non è possibile determinare *per quanto* un processo eseguirà, *quando* l'esecuzione avrà luogo, *in che punto* dell'esecuzione avverrà il cambio di stato e *quale* altro processo sarà eseguito dopo quello corrente.

Agli amministratori è concessa la possibilità di definire un tipo molto particolare di processi, chiamati *real-time* [4, p. 262], riservati per attività *estremamente* urgenti e caratterizzati da *priorità statica*. Questa è sempre *inferiore* a quella di tutti gli altri processi, e pertanto lo scheduler preferirà sempre la loro esecuzione alle altre. Non essendo soggetti a variazioni di priorità, il *quanto di tempo* ad essi assegnato è *costante* anche se non liberamente controllabile.

L'uso del precedente tipo di processi deve essere molto cauto, in quanto sono in grado di *precludere completamente* l'esecuzione di tutti gli altri. In un sistema in normali condizioni operative non è mai presente alcun processo real-time.

## Creazione e attesa della terminazione di processi

A pagina 42 si è accennato come nuovi processi possano essere creati dal sistema operativo per permettere l'esecuzione di diversi programmi, oppure dai programmatori per suddividere il carico di lavoro tra essi. In quest'ultimo caso, solitamente, questi hanno bisogno di meccanismi di *sincronizzazione* tra loro, ad esempio perché tutti i thread creati dal processo principale dell'algoritmo devono aver terminato la loro esecuzione prima di permettere a quest'ultimo di proseguire.

In questa sezione si descrivono le due funzioni che permettono la creazione e l'attesa della terminazione di un processo in Linux.

Il benchmark descritto nel terzo capitolo utilizza esclusivamente *light-weight process* per operare. La creazione di questi è possibile tramite la già citata funzione `clone`:

---

### Codice 1: Definizione della funzione `clone`.

---

```
1 pid_t clone(int (*fn)(void *), void *child_stack, int flags,  
2           void *arg);
```

---

---

<sup>1</sup>Chiamato anche *superuser* in Linux.



A differenza del meccanismo descritto a pagina 42 per la funzione `fork`, la `clone` permette di creare un nuovo *light-weight process* la cui esecuzione inizierà non all'istruzione successiva alla stessa `clone`, bensì alla prima incontrata nella funzione situata all'indirizzo contenuto in `fn`, che deve essere passato come primo argomento alla `clone`. La funzione iniziale del nuovo processo deve ammettere come argomento un puntatore di tipo generico (`void *`), che verrà passato alla `clone` nel campo `arg`.

L'argomento `child_stack` è un puntatore all'area di memoria che costituirà lo *stack* del processo figlio, mentre `flags` è una *maschera di bit* che consente di impostare alcune opzioni per il nuovo processo, tra cui la condivisione con le strutture del padre. È proprio grazie a `flags` che è possibile sfruttare le potenzialità fornite dal sistema operativo con i *light-weight process*.

Linux definisce una serie di *macro* che consentono di selezionare finemente le strutture da condividere in un light-weight process. Un elenco esaustivo di queste, assieme ad una descrizione completa della funzione, è disponibile nel manuale del sistema operativo, reperibile anche online (ad esempio in [19]).

Il valore di ritorno della funzione, in caso di successo, è il `pid` del nuovo processo, da memorizzare, come detto in precedenza, in una variabile di tipo `pid_t`.

Nel codice 17 si userà la funzione `clone` per implementare una porzione di benchmark (cui si rimanda per un esempio operativo di questa) nel quale si è anche massimizzata la condivisione delle strutture dati.

Un processo può attendere la terminazione di un altro tramite la funzione `waitpid`:

---

**Codice 2:** Definizione della funzione `waitpid`.

---

```
1 pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

---

Questa causa il cambio dello stato del chiamante da *running* a *blocked* finché il processo il cui `pid` è passato come primo argomento non *termina*, viene *posto il blocco* su di esso dal sistema operativo o tale blocco viene tolto. L'argomento `options` consente, tramite delle macro, di specificare quale di questi eventi causa la fine dell'attesa del processo.

Si noti che il termine *porre il blocco* non ha nulla a che vedere con lo stato *blocked* del processo; significa, invece, che un amministratore di sistema *esplicitamente* chiede al sistema operativo di non eseguire un dato processo per alcun motivo, fino alla rimozione di tale blocco.

Il modo più comune di utilizzare la funzione è rappresentato dalla chiamata `waitpid(pid, NULL, 0)`, che causa l'attesa del processo chiamante fino alla terminazione di `pid`.

In caso di successo il valore di ritorno della funzione è `pid` stesso. Non si analizzerà ulteriormente tale funzione, in quanto la spiegazione dell'uso

descritto precedentemente è sufficiente agli scopi di questa trattazione. Ulteriori approfondimenti sono disponibili nel manuale del sistema operativo, ad esempio in [20].

La funzione `waitpid` verrà utilizzata nel codice 18 per implementare una funzionalità del benchmark.

### 2.1.2 Gestione dell'I/O

#### Dispositivi a blocchi

Linux consente al programmatore di gestire ogni dispositivo esterno al calcolatore, come dischi rigidi, tastiere, stampanti o terminali a schermo, allo *stesso* modo. Questo è possibile grazie alla visione unificata, tipica dei sistemi UNIX, di ogni periferica: strutture dati *della stessa composizione* sono adottate per descrivere le loro caratteristiche, a prescindere dal tipo di periferica, ed è possibile utilizzare le *stesse* funzioni per operare su esse.

Linux distingue i vari dispositivi in *categorie*. In questa sezione si descrive il funzionamento dei *dispositivi a blocchi* [4, p. 543], ovvero quelli che consentono di accedere e trasferire dati a prescindere dalla loro posizione all'interno del dispositivo, impiegando all'incirca lo stesso tempo. Esempi di questa categoria sono dischi rigidi, CD-ROM, DVD-ROM o floppy disk. Assieme alla loro organizzazione si descriveranno le operazioni più importanti, nonché le ottimizzazioni che Linux compie su di essi.

Un dispositivo a blocchi memorizza al suo interno dei dati, genericamente strutturati in *file* [4, p. 29], termine che denota una sequenza ordinata di byte memorizzata in qualche periferica.

I controller hardware che consentono la lettura e la scrittura di dati sui dispositivi non sono in grado di trasferire esattamente la quantità di dati desiderata dall'utente, bensì operano su *multipli* dell'unità minima di memorizzazione, chiamata *settore*.

In *nessun* caso un sistema operativo può leggere o scrivere una quantità diversa da un suo multiplo, essendo questo un vincolo imposto per costruzione dall'hardware. Tipicamente tale dimensione è pari a 512B.

Il sistema operativo può ulteriormente ingrandire la minima unità di trasferimento, combinando più settori assieme ad ogni richiesta di lettura o scrittura. Si denota con il termine *pagina* tale unità. A differenza del precedente, questo vincolo è rimovibile su richiesta del programmatore. Di default tale dimensione è pari a 4KB.

#### Apertura

Un processo che vuole operare con un file memorizzato su un dispositivo a blocchi deve preventivamente procedere con la sua *apertura* [4, p. 33].

Le funzioni di input e output definite in Linux consentono di identificare univocamente un file tramite il suo *descrittore*, ovvero una struttura che

contiene tutte le informazioni necessarie al sistema operativo per operare sul file stesso. Ogni descrittore è memorizzato nella memoria volatile, ed è a sua volta identificato da un numero univoco.

L'operazione di apertura consente di creare un descrittore per un certo file, rendendo così possibile la sua lettura, la sua scrittura e ulteriori operazioni. Linux mette a disposizione la funzione `open` a tale scopo, definita nel codice 3.

---

**Codice 3:** Definizione della funzione `open`.

---

```
1 int open(const char *path, int oflag);
```

---

La stringa `path` consente di specificare il percorso sul dispositivo in cui il file è memorizzato, mentre l'argomento `oflag` ne specifica la *modalità* di apertura. Quest'ultima permette, tramite una serie di macro, di comunicare al sistema operativo le operazioni che si pensa di compiere sul file o il modo in cui esse devono avvenire. Ad esempio, è possibile specificare un'apertura in *sola lettura* nel caso non si voglia scrivere su di esso o in *sola scrittura* nel caso opposto oppure ancora in *lettura o scrittura*. Ulteriori opzioni saranno esaminate nel prosieguo del documento.

In caso di successo, la funzione crea in memoria il descrittore del file, e ne restituisce l'identificatore. È possibile approfondire la trattazione della funzione `open` in [21] o in un manuale del sistema operativo.

## Letture

Tra le funzioni che consentono la *lettura* [4, p. 612] di un file da una periferica si riporta qui la stessa utilizzata nella realizzazione del benchmark, la `pread`:

---

**Codice 4:** Definizione della funzione `pread`.

---

```
1 size_t pread(int fd, void *buf, size_t count, off_t offset);
```

---

Il comportamento di questa funzione percepito dal programmatore è il seguente: il sistema operativo tenta di leggere *una porzione* di file, aperto in precedenza e definito dal descrittore `fd`. La dimensione della porzione letta è pari a `count` *byte* e la lettura avviene a partire dal byte numero `offset` della sequenza (si ricordi infatti che un file è una sequenza di byte).

I tipi `size_t` e `off_t` sono delle semplici ridefinizioni di un intero, come `int` o `long int`. I dati letti vengono messi in una variabile il cui indirizzo in memoria è memorizzato dall'argomento `buf`. Ovviamente la variabile deve essere grande a sufficienza per contenere almeno `count` byte di dati.

Il valore di ritorno della funzione è il numero di byte effettivamente letti dal disco. In caso di successo tale valore coincide con `count`, altrimenti è diverso. Quest'evenienza può indicare la fine del file o il sopraggiungere di un evento che impedisce una lettura completa. Ulteriori informazioni sulla funzione si trovano in [22] o in un qualunque altro manuale di Linux.

Si noti che qualsiasi *lettura* da disco pone il processo nello stato *blocked* finché questa non è stata terminata.

Le operazioni di lettura costituiscono parte fondamentale del benchmark realizzato in questa tesi e il loro funzionamento è visibile nel codice 15.

### Page cache e read-ahead

Sebbene la funzione `pread` consenta di leggere la quantità di dati desiderata dal programmatore, se esistente, a pagina 48 si è detto che vi sono dei limiti strutturali alle dimensioni trasferibili.

Il sistema operativo nasconde all'utente ogni possibile limitazione grazie alla *page cache* [4, p. 580], una memoria dati *non accessibile* in alcun modo e non visibile al programmatore. Una richiesta di lettura di `count` byte causa in realtà le seguenti operazioni:

1. Il sistema operativo richiede la lettura della *pagina* che contiene i dati richiesti.
2. Il controller hardware legge la pagina dal disco, che viene quindi posizionata nella *page cache* dal kernel.
3. Dalla *page cache* i soli `count` byte richiesti vengono trasferiti nell'area di memoria voluta dall'utente.

Il primo punto elencato in precedenza può prevedere, inoltre, la lettura di *più* pagine contigue dal disco, e non solo di quella contenente il dato. Questo meccanismo, chiamato *read-ahead* [4, p. 619], è motivato dalla nota lentezza dei dispositivi di archiviazione: la velocità con cui il disco accede al dato dipende in larga parte dalla posizione della *testina* al momento della richiesta. La *read-ahead* tenta di evitare, in caso di letture di due pagine adiacenti in istruzioni diverse, di perdere il tempo che sarebbe necessario al posizionamento della testina, che può risultare molto penalizzante.

È inoltre empiricamente dimostrato che l'insieme delle porzioni di file richieste in lettura è quasi sempre situato su più pagine: raramente un file viene aperto con lo scopo di leggerne solo pochi byte. Page cache e read-ahead rappresentano validi meccanismi di ottimizzazione dei tempi, la cui adozione è adeguata in quasi tutte le situazioni.

Nella realizzazione di un benchmark in cui si vuole simulare un'*esatta* operazione di lettura, tuttavia, deve essere fatto notare che questi meccanismi condizionano *considerevolmente* le operazioni eseguite, in quanto:

- Una richiesta di lettura da disco *non sempre* corrisponde ad una tale operazione su esso. Se infatti la sequenza di byte richiesta è già presente nella page cache, il sistema operativo si limiterà a copiare nell'area di memoria definita dal programmatore tali dati.
- Non ci si deve aspettare che ad una richiesta di `count` byte corrisponda un'effettiva lettura di tale quantità dal disco: una o più pagine verranno lette e caricate in memoria, a prescindere dalla quantità voluta dall'utente.

Read-ahead e page cache possono essere *disabilitate* per singoli file, aprendoli in *modalità diretta* [4, p. 640]. In questo caso le *richieste* di lettura corrispondono ad effettive operazioni su disco. Il sistema operativo non limita più la dimensione minima di lettura ad una *pagina*, tuttavia il limite fisico del *settore* rimane presente. Sarà cura del programmatore chiedere alla funzione `pread` un numero di byte multiplo della dimensione di questo.

## Scrittura

Oltre alla richiesta di dati da un dispositivo, anche la *scrittura* [4, p. 624] di questi è possibile tramite diverse funzioni messe a disposizione da Linux. Tra esse si descrive la `pwrite`, di struttura simile alla `pread` e utilizzata nella realizzazione del benchmark:

---

### Codice 5: Definizione della funzione `pwrite`.

---

```
1 size_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

---

Come si può notare, la sua definizione è praticamente uguale a quella della `pread` riportata nel codice 4. Intuitivamente, ciò che cambia è la *direzione* del trasferimento dei dati: il contenuto dell'area di memoria indirizzata da `buf` viene memorizzato nel file il cui descrittore è identificato da `fd` a partire dalla posizione `offset` e per un totale di `count` byte. Eventuali dati già presenti in quella porzione del file vengono *sovrascritti*.

Analogamente alla `pread`, il valore di ritorno della funzione è il numero di byte effettivamente scritti da questa. In caso di successo tale valore coincide con `count`, altrimenti è diverso. Quest'evenienza può indicare l'esaurimento dello spazio sul disco, l'impossibilità di scrivere su di esso o il sopraggiungere di un evento che ha causato l'interruzione dell'operazione. La pagina [22] contiene approfondimenti sulla `pwrite`, oltre che sulla `pread`.

Anche le scritture su disco sono utilizzate come parte fondamentale del benchmark e possono essere viste in funzione nel codice 15, assieme alle letture dal dispositivo.

## Osservazioni sull'operazione di scrittura

A differenza di una lettura, ogni scrittura è, di default, *asincrona*. Questo termine significa che il processo *non* è per forza posto nello stato *blocked* fino alla fine dell'operazione, bensì la sua esecuzione può continuare come se la richiesta non fosse avvenuta.

La distinzione nel comportamento delle due operazioni è motivata dalle diverse dipendenze che il processo ha nei confronti dei dati letti o scritti. Nel primo caso, generalmente, un algoritmo richiede i dati perché ne ha bisogno in operazioni successive. Nel secondo caso invece, il processo continua ad avere in memoria quelli che scrive su disco. Tale operazione è svolta non per necessità del processo, ma per salvare permanentemente informazioni che saranno lette in seguito da altri programmi o dagli utenti.

Appurato la correttezza di non eseguire subito una scrittura fisica su disco, l'asincronismo consente di evitare inutili attese *posticipando* l'operazione a discrezione del sistema operativo. Nel caso di più richieste di scrittura sulla stessa porzione di un file, questo meccanismo consente, inoltre, di riflettere sul dispositivo solo l'*ultima* di esse, evitando di eseguire costosi movimenti della sua testina.

Similmente alla page cache e alla read-ahead, l'asincronismo ottimizza le operazioni sulle periferiche di archiviazione, mantenendo la loro correttezza e risparmiando tempo. Tuttavia il fatto che ad una richiesta di scrittura *non sempre* corrisponda un'effettiva operazione in tal senso deve essere considerato con attenzione nella realizzazione del benchmark.

Linux permette di disabilitare l'asincronismo delle operazioni di scrittura tramite l'apertura di un file in *modalità sincrona*. Questo comporta per la `pwrite` e funzioni simili lo stesso comportamento previsto per le operazioni di lettura: il processo verrà posto nello stato *blocked* fino al termine di questa.

Un'altra osservazione riguarda la page cache, che modifica sia la scrittura, sia la lettura. In sua presenza, una `pwrite` di `count` byte da un'area di memoria indirizzata da `buf` si comporterà come segue [4, p. 602]:

1. I `count` byte di dati contenuti nell'area di memoria puntata da `buf` vengono trasferiti nella *page cache*.
  - Se la pagina corrispondente alla porzione del file sull'hard disk su cui devono essere scritti i dati non è presente in memoria, questa viene *letta* dal disco e quindi modificata come richiesto, tenendo conto anche del meccanismo di read-ahead.
2. Il sistema operativo scriverà l'intera *pagina* in memoria solo in seguito, in un istante *non predicibile*.

Ancora una volta, la page cache si rivela un meccanismo molto utile per ottimizzare il più possibile i movimenti della testina del disco rigido. Tuttavia appare chiaro che, in sua presenza:

- Una richiesta di scrittura su disco *non corrisponde mai* ad una tale e *immediata* operazione su esso, in quanto i dati verranno prima memorizzati nella page cache.
- Non è possibile sapere *quando* la scrittura avverrà realmente.
- La richiesta di count byte non corrisponderà un *effettiva* scrittura di tale quantità sul disco, poiché sarà un'intera pagina ad essere trasferita.
- Paradossalmente, una scrittura può comportare anche una *lettura* da disco di almeno una pagina.

È possibile assicurare un comportamento coerente con quanto ci si attende tramite la modalità diretta, avendo cura di richiedere scritture di dati di dimensioni multiple di un settore, similmente a quanto accadeva per le letture.

Le osservazioni riportate in questa sezione sulla page cache, read-ahead e scrittura asincrona devono essere considerate con attenzione quando, come nel caso di questo lavoro, si vuole riprodurre *con esattezza* un certo comportamento desiderato da un utente. Si discuteranno nuovamente tutte queste caratteristiche nel corso del terzo capitolo.

### 2.1.3 Gestione del tempo

Ogni operazione di un sistema operativo è guidata dallo scorrere del tempo [4, p. 225]. Lo scheduler, introdotto a pagina 45, rappresenta solo un esempio di come sia importante mantenere un efficiente meccanismo di misurazione temporale.

I programmatori possono trarre vantaggio da ciò chiedendo a Linux la creazione e la gestione di “conti alla rovescia” personalizzati, impostati per “avvisare” un processo quando questi esauriscono il tempo a loro disposizione.

Un'altra funzionalità del sistema è il mantenimento della data e dell'ora corrente, la cui presenza è evidente a tutti gli utenti di ogni personal computer.

È inoltre possibile conoscere il tempo trascorso dall'accensione del sistema per avere dei riferimenti temporali nella propria applicazione, ad esempio per misurare l'istante di inizio e di fine di un'operazione.

### Misura del tempo

Il metodo più semplice per misurare lo scorrere del tempo all'interno di un calcolatore è memorizzare in un *contatore* il numero di impulsi elettrici prodotti da un *oscillatore* (chiamato anche *orologio* o *clock*) operante a frequenza nota. Poiché questa è, per definizione, il numero di impulsi prodotti in un secondo, è possibile ottenere una misura temporale dividendo per essa il contenuto del contatore e ricavando, in questo modo, i *secondi* trascorsi dal momento in cui si è iniziato a tenere traccia dei segnali generati dall'oscillatore.

Come già descritto a pagina 17, le azioni di ogni componente di un calcolatore sono scandite da un dispositivo fisico che ad intervalli regolari emette degli impulsi elettrici, permettendo il loro funzionamento. Ci si riferisce a questo con il termine *clock di sistema*. Il contatore del numero di cicli da esso generati è chiamato ***Time Stamp Counter (TSC)*** ed è presente in ogni macchina.

La sua frequenza è, tuttavia, soggetta ad aumenti e diminuzioni esplicite da parte dei più moderni processori, che la abbassano quando le operazioni in corso sono poche o non richiedono grande potenza di calcolo, al fine di risparmiare energia. Utilizzare il clock di sistema come oscillatore per gestire il tempo non è quindi sicuro, perché l'intervallo trascorso tra un impulso e un altro potrebbe cambiare, falsando il tempo ricavato. Questo fenomeno è noto con il nome di *time warp*.

Per superare questo problema, nel corso degli anni le macchine hanno visto l'inserimento al loro interno di nuovi orologi [4, p. 227]. Il più potente di questi è lo ***High Precision Event Timer (HPET)***, che non memorizza il numero di impulsi generati dal clock di sistema, ma da un *oscillatore proprio* la cui frequenza è liberamente programmabile dal sistema operativo in qualsiasi momento e non soggetta a modifiche. Grazie a questo è possibile avere una misura coerente del tempo, precisa al *nanosecondo* data la sua architettura avanzata.

### Richiesta di informazioni temporali

Linux mette a disposizione di ogni processo diversi tipi di informazione temporale. Ad esempio, il programmatore può richiedere da quanto tempo un processo o l'intero sistema sono attivi o può visualizzare la data e l'ora correnti. Il contenuto dello HPET è convertito in una grandezza temporale e, quindi, fornito all'utente.

Il benchmark oggetto della tesi ha la chiara necessità di conoscere la durata di una certa sequenza di istruzioni. Un modo per ottenere questa informazione consiste nel richiedere l'*uptime* di sistema, ovvero l'intervallo di tempo da cui questo è in funzione, *subito prima* dell'inizio della sequenza e *subito dopo*. La differenza tra i due uptime costituisce approssimativamente la misura richiesta.

Un'altra funzionalità che si desidera implementare è mostrare *quando* le varie operazioni sono avvenute. Poiché l'*uptime* scorre indipendentemente dalla vita dei processi, procedendo come descritto sopra è possibile avere un riferimento *assoluto* dell'inizio e della fine delle istruzioni, permettendo quindi il confronto di questi valori.

La funzione che consente di ottenere diverse misure temporali è chiamata `clock_gettime`, la cui definizione è mostrata nel codice 6.

Tramite essa è possibile richiedere al kernel diversi tipi di informazione riguardanti il processo o l'intero sistema. Il primo argomento, `clk_id` consente



**Codice 6:** Definizione della funzione `clock_gettime`.

---

```
1 int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

---

di specificare quale misura temporale si desidera, ad esempio da quanto tempo il sistema è attivo o la data e l'ora attuali. L'informazione richiesta viene restituita in una struttura `timespec`, composta come segue:

**Codice 7:** Definizione della struttura `timespec`.

---

```
1 struct timespec {
2     time_t    tv_sec;        /* secondi */
3     long     tv_nsec;       /* nanosecondi */
4 };
```

---

Il benchmark utilizza `clock_gettime` per ottenere l'uptime del sistema. Questo è possibile passando la macro `CLOCK_BOOTTIME` come primo argomento. Data la composizione della struttura `timespec`, è possibile trasformare le informazioni ricevute in una quantità più pratica come mostrato dalla semplice funzione seguente, in grado di ricavare un tempo espresso in *millisecondi*:

**Codice 8:** Implementazione della funzione `computeTime`.

---

```
1 double computeTime(struct timespec time) {
2     return (double) time.tv_sec * (double) 1000 +
3     (double) ( (double) time.tv_nsec / (double) 1000000 );
4 }
```

---

Maggiori informazioni sulla funzione `clock_gettime` e sulla struttura `timespec` si possono reperire, ad esempio, in [23].

**Osservazioni sulle informazioni richieste**

Si consideri il codice 9. È intento del programmatore misurare sia il tempo di esecuzione della funzione `operazioneDaMisurare`, sia gli istanti di inizio e fine di questa. Si ipotizzi che `ottieniUpTime` inglobi tutte le operazioni necessarie per ottenere un uptime corretto, procedendo come descritto in precedenza.

In questa sezione si vuole discutere di come, nonostante l'apparenza, in `inizio`, `fine` e `durata` non siano contenuti gli effettivi valori desiderati dal programmatore.

Si osservi, innanzitutto, come dalla chiamata della `clock_gettime` (ad opera di `ottieniUpTime` nell'esempio) e l'inizio di `operazioneDaMisurare` passi un certo intervallo temporale, che sarà incluso in `durata`. Allo stesso

**Codice 9:** Esempio di richiesta di informazioni temporali.

---

```
1 double inizio = ottieniUpTime();
2 operazioneDaMisurare();
3 double fine = ottieniUpTime()
4 double durata = fine-inizio;
```

---

modo, tra la fine della funzione e la seconda chiamata di `clock_gettime` alcuni millisecondi andranno ad aumentare il valore di `durata` impropriamente. Le variabili `inizio` e `fine` non memorizzeranno *esattamente* i due istanti desiderati, bensì dei valori, rispettivamente, leggermente in anticipo e in ritardo. Essendo tutto ciò inevitabile, è necessario cercare di *minimizzare* il più possibile la distanza, in termini di istruzioni, tra le richieste dell'*uptime* e la funzione da monitorare.

Nonostante una programmazione accurata, lo scheduler potrebbe inoltre togliere dall'esecuzione il processo tra le righe 1 e 2, o tra le righe 2 e 3. Questo causerebbe l'inclusione in `durata` di tempi nel corso dei quali, in realtà, l'applicazione era stata temporaneamente fermata. Allo stesso modo, `inizio` o `fine` sarebbero ulteriormente falsati.

Sarebbe possibile prevenire questa inclusione richiedendo a `clock_gettime` un'informazione diversa, come il tempo di *esecuzione effettiva* di un processo, anziché l'*uptime* del sistema. In questo modo sarebbe possibile determinare in maniera più corretta le durate delle operazioni, in quanto il tempo in cui il processo non si trova nello stato *running* non verrebbe conteggiato. Tuttavia, in questo modo non sarebbe possibile creare un sistema di riferimento assoluto, in cui mettere a confronto gli istanti di creazione e terminazione di *tutti* i processi di un'applicazione: essendo anche questo uno scopo del benchmark, questa soluzione non è applicabile.

Infine, lo scheduler potrebbe anche decidere di interrompere l'esecuzione del processo nel corso di `operazioneDaMisurare`. Anche in questo caso `durata` sarebbe falsata da un tempo aggiuntivo fittizio. Segnalando un fatto ancora più importante, si immagina che il codice precedentemente riportato sia eseguito anche da un secondo processo, entrambi in esecuzione su un sistema con una sola CPU. Se lo scheduler decidesse di togliere dall'esecuzione uno dei due processi per poi eseguire l'altro, non solo `durata` verrebbe falsata, ma anche gli intervalli tra `inizio` e `fine` dei due processi si sovrapporrebbero, creando l'illusione di un'esecuzione parallela quando, in realtà, è ben noto che questa è solo apparente (si veda quanto descritto a pagina 18).

Come già detto a pagina 45, non è possibile sapere quando un processo verrà tolto dall'esecuzione. Inoltre il cambio di stato è *trasparente* dal punto di vista del programmatore: *non esistono* funzioni in grado di notificarlo, né è possibile tenere traccia in alcun modo dei *solli istanti* in cui un processo è nello stato *running*.

È importante avere ben presenti queste considerazioni durante l'analisi dei grafici prodotti dal benchmark, che verranno discussi nel terzo capitolo.

### Controllo dei tempi

Il programma che si vuole realizzare ha un'altra importante caratteristica: deve essere in grado di eseguire un certo tipo di operazione *per un periodo di tempo* stabilito dall'utente. Se la sola misura della durata delle operazioni è problematica, ancora più difficile risulta assicurarsi che una certa funzione operi esattamente per il tempo voluto.

Linux mette a disposizione un meccanismo che consente di creare un "conto alla rovescia" di durata stabilita dal programmatore, al cui termine attiva una funzione specificata alla sua creazione. Questi strumenti sono noti come *timer software* [4, p. 243] e sono ampiamente utilizzati nel sistema operativo stesso per svolgere le sue regolari funzioni.

Un possibile approccio per realizzare quanto desiderato potrebbe consistere nel creare un timer con una certa durata, quindi proseguire con l'esecuzione dell'operazione. Allo scoccare del timer, una funzione potrebbe segnalare al processo la scadenza del tempo configurato dall'utente, in modo che l'operazione possa essere fermata.

Il problema principale dei timer è la loro *gestione tardiva* da parte del sistema operativo. Sebbene si abbia la sicurezza che la funzione associata allo scadere di essi *non* venga chiamata prima dell'esaurimento del tempo, non si ha tuttavia la certezza che venga eseguita *nell'esatto istante* in cui il timer raggiunge lo zero.

Il motivo di questo comportamento è la considerazione dei timer software come strumenti *non urgenti*, alla cui gestione sono preferite altre operazioni più importanti. La funzione associata viene invocata quando il sistema operativo è "poco impegnato", con un ritardo che può essere anche nell'ordine di *centinaia di millisecondi*, tempo che risulta troppo oneroso ai fini del benchmark.

Per questo motivo si è scelto di *non* implementare timer software nel prototipo realizzato, sebbene si sia ritenuto opportuno descriverli brevemente dato che sono molto conosciuti ed ampiamente utilizzati.

Nel terzo capitolo si proporrà un meccanismo per gestire il comportamento desiderato senza l'uso di timer software.

## 2.2 Il progetto Apache Hadoop

A pagina 36 si è parlato della necessità di un nuovo modello di programmazione in grado di utilizzare al meglio le potenzialità offerte dal *cloud computing*. Assieme alla continua evoluzione dell'infrastruttura e al crescere delle sue dimensioni, gli sviluppatori di applicazioni che intendono sfruttare

il cloud hanno bisogno di strumenti per interfacciarsi ad esso in modo semplice e pratico, alla luce dei concetti di *astrazione* e *trasparenza* che rendono impossibile utilizzare un'esatta risorsa nella rete esistente.

In questa direzione Google ha introdotto nel 2004 *MapReduce* [6], un nuovo paradigma di programmazione specifico per il calcolo distribuito su cloud e grid che tiene in considerazione l'alto grado di parallelismo delle applicazioni e il numero sempre crescente delle unità di calcolo a disposizione.

Le enormi dimensioni dei file con cui si opera nel cloud computing necessitano, inoltre, di un nuovo modello di gestione dei dati. Il concetto di *file system distribuito* consente la visione unificata di ogni dispositivo di memorizzazione presente sulla rete, trattando ognuno di questi come parte di un unico grande archivio a disposizione delle applicazioni.

I modelli teorici definiti da MapReduce e dal file system distribuito sono stati implementati da parte di Apache nel progetto *Hadoop* [32], un *framework* che permette l'elaborazione distribuita di grandi insiemi di dati tramite delle semplici API a disposizione dei programmatori.

Hadoop è ad oggi la piattaforma più diffusa per il calcolo distribuito, per via della facilità d'uso e della sua politica *open source*. L'intero progetto è stato costruito in sintonia con i principi del cloud computing, potendo così essere installato con facilità su comuni elaboratori di natura eterogenea. È scritto nel linguaggio di programmazione *Java*, pertanto può essere eseguito su ogni macchina in grado di ospitare una *Java Virtual Machine*.

Famose aziende nel campo informatico come Yahoo!, Microsoft e Amazon utilizzano Hadoop per le loro operazioni già a partire dal 2007.

Nel quarto capitolo si mostrerà come sia possibile ricavare e riprodurre tramite il benchmark realizzato un workload generato da un'applicazione utilizzante Hadoop.

Il modello di programmazione e di gestione dei dati sono i due elementi che caratterizzano maggiormente il progetto. In seguito si descrive come Hadoop li implementa, fornendo i concetti necessari alla comprensione del loro funzionamento in applicazioni basilari, senza trattarne funzionalità avanzate.

### 2.2.1 Hadoop Distributed File System

Il file system distribuito implementato da Apache è chiamato *Hadoop Distributed File System (HDFS)* [33]. Questo è stato progettato tenendo conto delle condizioni di funzionamento tipiche dei sistemi cloud, che non solo includono calcolatori di prestazioni variabili dislocati in regioni potenzialmente molto distanti, ma anche enormi quantità di dati operativi. Il funzionamento dell'HDFS è ispirato ad un'altra architettura sviluppata da Google che tende allo stesso fine, chiamata *Google File System* [12].

A differenza dei dispositivi di archiviazione locali, come gli hard disk, l'HDFS assume che il *fallimento* dei nodi in cui sono contenuti i file sia un'evenienza non trascurabile, mettendo in atto un meccanismo di *replicazione*

per consentire l'accesso ai dati in ogni evenienza. Le grandi dimensioni dei file, che possono raggiungere anche alcuni *terabyte* di grandezza singolarmente, rendono necessarie opportune politiche di *divisione e distribuzione* di questi, il tutto in modo trasparente agli utenti del cloud.

Diversamente dai normali file system, le informazioni contenute nell'HDFS sono inoltre ottimizzate per applicazioni specifiche, progettate per scrivere i file *una sola volta*. Non sono dunque ammesse modifiche ai dati memorizzati in rete, se non la loro cancellazione o il cambio del loro nome.

L'organizzazione dei file è tale da *massimizzarne il throughput* piuttosto che la *velocità* di trasferimento. Ci si deve dunque aspettare grandi quantità di dati letti o scritti con una sola istruzione piuttosto che pochi byte, in quanto il tempo impiegato per tali operazioni è per costruzione molto elevato.

## Architettura

HDFS è composto da un'architettura di tipo *master/slave*, in cui ogni componente del sistema (*slave*) agisce esclusivamente sotto ordine di un *master*, a cui rendiconta le operazioni svolte.

Si definisce **NameNode** l'*unico* master presente nella rete. Tutti gli altri componenti sono noti con il nome di **DataNode**. Questi ultimi contengono al loro interno i dati memorizzati dalle applicazioni, ed eseguono le operazioni di creazione, scrittura, lettura, rimozione e *replicazione*. Affinché possano operare necessitano di un esplicito ordine da parte del *NameNode*, che stabilisce *quali DataNode* debbano contenere determinati dati.

Gli slave della rete *non* conoscono la presenza o la locazione degli altri *DataNode*, se non nel caso di un'esplicita richiesta di trasferimento dati da o per essi coordinata dal *NameNode*. Un'applicazione interagisce esclusivamente con quest'ultimo, che smista le operazioni ai nodi più opportuni. Il master non riceve o scrive direttamente alcun file, né essi transitano da esso.

Come già accennato, un file *non* può essere modificato una volta che lo si è memorizzato nell'infrastruttura. Una richiesta di creazione comporta la *divisione* del file in *blocchi*, ognuno dei quali verrà memorizzato in un diverso *DataNode*, se ne esistono a sufficienza. Tipicamente un blocco ha dimensione pari a 64MB, anche se un'applicazione può modificarla per singoli file. Essa è uguale per ogni blocco del file, ad eccezione, eventualmente, dell'ultimo che sarà più piccolo degli altri. Il *NameNode* decide in quali *DataNode* i blocchi devono essere memorizzati.

L'intera struttura del file system, tra cui la posizione di ogni blocco nella rete, è mantenuta esclusivamente dal *NameNode* nella sua memoria volatile e tramite il suo *log* nel disco fisso. Il master, unico per costruzione, costituisce il *Single Point Of Failure (SPOF)* della rete e la sua caduta metterebbe fuori operatività l'intero HDFS.

## Affidabilità e replicazione

La piattaforma è progettata per operare con un numero di *DataNode* anche nell'ordine delle centinaia. Con tale estensione, e ricordando che nel cloud è solitamente impiegato hardware comune, senza attenzioni speciali sulla sua durabilità, il fallimento di un nodo della rete è da prendere in seria considerazione. HDFS mette in atto un sistema di **replicazione** per prevenire il più possibile i danni causati da un'eventuale caduta di un nodo, di un intero rack o del data center (si ricordi la sua struttura, commentata a pagina 35).

Ogni file è caratterizzato da un **fattore di replicazione**, che rappresenta il *numero di repliche* che ogni suo blocco deve avere all'interno del file system. Alla creazione di un file, il *NameNode* non si limita a dividerlo in blocchi e a decidere la posizione di ognuno di essi, ma ordina anche la loro replicazione su più *DataNode* del sistema.

Il *NameNode* controlla costantemente che nessun blocco scenda al di sotto del fattore di replicazione del proprio file, neanche in seguito alla loro memorizzazione, ad esempio per guasti ai *DataNode*. In tal caso dispone nuove repliche in nuovi nodi.

Ogni *DataNode* invia al *NameNode* un segnale periodico, denominato *Heartbeat*, per notificare la sua operatività. Questo è accompagnato dall'elenco dei file in lui memorizzati, in modo che il *NameNode* possa tenere traccia della presenza e dello stato di ogni blocco. La mancata ricezione dell'*Heartbeat* concretizza la caduta del nodo.

Similmente alla grandezza di un blocco, un'applicazione può scegliere anche un diverso fattore di replicazione per ogni file. Di default questo è pari a *tre*, pertanto ogni blocco sarà memorizzato in almeno *quattro DataNode* diversi.

La politica con cui questi sono scelti prevede di distribuire le repliche sia tra *DataNode* dello *stesso rack* sia tra nodi di *rack diversi*. Con il fattore di replicazione di default, *una* replica sarà memorizzata nello stesso rack del nodo contenente il blocco originale, mentre *due* repliche saranno memorizzate in un rack diverso, ognuna in un differente *DataNode*.

In questo modo, Hadoop tenta di limitare il traffico *inter-rack*, utilizzando solo due di essi, salvaguardando allo stesso tempo il sistema dalla caduta di un intero rack, ad esempio per guasti di una certa entità all'interno di un data center.

Le repliche sono totalmente identiche al blocco originale, pertanto una richiesta di lettura di un blocco può essere portata a termine indifferentemente su esso o su una delle sue repliche. In questo modo il *NameNode* può ottimizzare la vicinanza di dati e applicazioni, minimizzando il tempo necessario a compiere le operazioni. L'impossibilità di modificare un file in seguito alla sua memorizzazione semplifica inoltre la gestione delle repliche, che sono sempre *coerenti* con l'originale.

## 2.2.2 Hadoop MapReduce

Le potenzialità offerte dall'HDFS sono sfruttate al meglio dividendo un programma in più applicazioni indipendenti, ognuna in esecuzione in prossimità dei dati che necessita e comunicanti tra loro solo nelle fasi finali dell'esecuzione, per unificare i risultati prodotti. Il modello di programmazione proposto con MapReduce consente facilmente di strutturare un programma in tal modo.

Per comprendere la necessità di un simile paradigma, si ricordi che, come detto più volte, i dati su cui le applicazioni operano sono molto grandi e distribuiti in più nodi nell'intero cloud. Se questi non sono composti esclusivamente da periferiche di archiviazione, ma anche da risorse computazionali, è possibile localizzare tutte le unità in cui i dati sono dislocati e far svolgere ai processori *delle stesse unità* le operazioni necessarie *sui loro dati*. Al termine di queste, i risultati ottenuti possono essere trasferiti ad altri nodi che si limiteranno a rielaborarli e quindi unificarli, fornendo all'utente il risultato richiesto.

Quanto descritto sopra costituisce l'essenza di MapReduce: le operazioni svolte sui dati locali sono chiamate *map*, mentre la rielaborazione finale dei risultati è chiamata *reduce*. L'implementazione del modello in Hadoop è chiamata *Hadoop MapReduce* [13, 27, 34, 36, 37].

### Architettura

Hadoop MapReduce e HDFS lavorano in stretta sinergia per consentire le migliori potenzialità al calcolo distribuito. Anche MapReduce è composto da un'architettura master/slave: ogni nodo in grado di compiere un'attività computazionale è chiamato *TaskTracker* ed opera sotto la direzione dell'unico master del sistema, il *JobTracker*.

È quest'ultimo a dialogare con ogni applicazione richiedente delle operazioni distribuite, stabilendo in base a queste in quante fasi di *map* dividerle, in quante *reduce* e in quali *TaskTracker* compierle. Ogni slave rende conto lo stato delle operazioni al *JobTracker*.

Come nell'HDFS l'eventualità di fallimento delle operazioni ordinate dal master viene gestita dallo stesso, comandando operazioni analoghe in nuovi *TaskTracker* della rete. Il *JobTracker* è ovviamente il SPOF del sistema.

### MapReduce e HDFS

Non dovrebbe sorprendere che una tipica installazione di Hadoop consti di un singolo nodo che funge sia da *JobTracker* sia da *NameNode*, mentre tutti gli altri operano sia come *TaskTracker* sia come *DataNode*.

Infatti, un *JobTracker*, nel determinare i *TaskTracker* migliori per eseguire un'operazione MapReduce, deve necessariamente sapere *in quali nodi* i blocchi dei file che l'applicazione richiede sono situati, consultando quindi il *NameNode*. La coincidenza dei due evita lo scambio di informazioni nella rete.

Inoltre, il posto migliore in cui eseguire un'operazione su un dato contenuto in un *DataNode* è nel *TaskTracker* a lui più vicino, condizione che verrebbe ottimizzata da una convivenza dei due nello stesso nodo.

La singola macchina master del sistema è dunque l'unica che dovrebbe essere equipaggiata per evitare il più possibile ogni guasto e con la migliore qualità di risorse, in quanto non solo essa è l'unica ad orchestrare tutte le operazioni di Hadoop, ma è anche il SPOF della rete.

### Funzioni di map e di reduce

Un'applicazione che desidera utilizzare Hadoop MapReduce deve implementare alcuni componenti per assicurare la corretta operatività del sistema. In questa sezione si approfondisce il discorso sulla loro modalità di funzionamento e si distinguono formalmente le varie fasi previste dal modello.

Le funzioni di *map* e *reduce* operano esclusivamente su dati formati da coppie del tipo (*chiave*, *valore*), nel seguito abbreviate intuitivamente con la notazione  $(K, V)$ . Nella loro forma generale è possibile descriverle come segue:

- Una **funzione di map** trasforma una coppia  $(K_1, V_1)$  in un *insieme* di coppie  $(K_2, V_2)^*$  che possono essere anche di tipo diverso dalla prima coppia.
- Una **funzione di reduce** trasforma una coppia  $(K_2, V_2^+)$  in un *insieme* di coppie  $(K_3, V_3)^*$ , il cui tipo può, anch'esso, differire da quello della coppia iniziale.

L'operatore *croce*  $^+$  indica una sequenza non vuota di *valori*, mentre la *stella di Kleene*  $^*$  ne indica una eventualmente vuota. Questo significa che gli *output* delle funzioni possono anche essere nulli per un certo input.

Come si può notare, le *chiavi* in input alla funzione di reduce sono simili all'output della funzione di map: l'unica differenza è che, mentre quest'ultima può fornire in output *più coppie con la stessa chiave*, nella reduce sono necessari input formati da *chiavi diverse*.

I diversi valori associati alla stessa chiave nell'output della map possono essere logicamente riuniti in un "*macrovalore*" composto da essi. Ad esempio, le tuple  $\{(k_1, v_1), (k_1, v_2)\}$  risultanti dalla funzione di map dovrebbero essere trasformate in un'unica tupla  $(k_1, [v_1, v_2])$  per poter essere gestite da una reduce.

### Mapper e reducer

Le due funzioni da sole non possono essere utilizzate per svolgere alcuna operazione completa in MapReduce, in quanto manca il passaggio che trasforma l'output della map nel formato adeguato per l'input della reduce, come si può notare dalla loro precedente definizione.



Un altro fattore da considerare è la *formattazione* dei dati come coppie (*chiave, valore*), che non può essere automatico e deve essere pensato attentamente. È chiaro quindi che map e reduce da sole costituiscono solo *una parte* del necessario.

Si definisce **mapper** la porzione di un programma MapReduce che si occupa, come minimo, di:

- Leggere i file con cui operare.
- Elaborarli in modo da formare opportune coppie  $(K_1, V_1)$ .
- Chiamare, per ogni coppia generata, la *funzione di map*.
- Memorizzare l'insieme  $(K_2, V_2)^+$  generato sul file system distribuito.

Si definisce invece **reducer** la porzione di un programma MapReduce che si occupa, come minimo, di:

- Leggere l'insieme  $(K_2, V_2)^+$  prodotto dalle map.
- Unificare i valori di tutte le coppie con la stessa chiave, formando quindi le tuple  $(K_2, V_2^+)^+$ .
- Chiamare, per ognuna di essa, la *funzione di reduce*.
- Memorizzare l'insieme  $(K_3, V_3)^+$  generato sul file system distribuito, che costituisce anche il risultato desiderato dall'applicazione.

### Un esempio operativo

Un tipico esempio utilizzato in molti *tutorial* per chiarire il funzionamento di un'applicazione MapReduce è l'implementazione di un programma che restituisca come risultato quali *parole* compongono un file di testo, assieme al *numero* delle loro occorrenze in esso. Si vogliono consolidare i concetti fino ad ora introdotti discutendo quali siano i passaggi necessari in una simile applicazione.

Si assuma che il file di testo su cui opera l'applicazione abbia il seguente contenuto:

```
ape bue gatto ape cane ape bue
```

Il mapper, responsabile di formare un possibile input per la funzione di map, potrebbe leggere il file e per ogni parola incontrata formare la coppia (*posizione, parola*), dove la *chiave* sarebbe costituita dalla posizione del primo carattere di *parola*. Nell'esempio sopra, le coppie prodotte sarebbero:

```
(1,ape), (5,bue), (9,gatto), (15,ape), (19,cane), (24,ape), (28,bue)
```

La map potrebbe così operare su queste, scritte nel giusto formato, e restituire come output, per ogni coppia (*posizione, parola*), la tupla (*parola, 1*) in questo modo:

(ape, 1), (bue, 1), (gatto, 1), (ape, 1), (cane, 1), (ape, 1), (bue, 1)

L'insieme verrebbe quindi scritto dal mapper nel file system, da cui verrà prelevato dal reducer. Si noti come l'output della map sia di tipo completamente diverso dal suo input, con chiavi totalmente differenti.

Il reducer potrebbe unificare l'insieme precedente nelle seguenti tuple:

(ape, [1, 1, 1]), (bue, [1, 1]), (cane, 1), (gatto, 1)

Per poi passarle, ora che sono nel giusto formato, alla funzione di reduce che produrrebbe quindi il risultato desiderato:

(ape, 3), (bue, 2), (cane, 1), (gatto, 1)

### Implementazione in Hadoop MapReduce

I concetti fino ad ora trattati costituiscono una base iniziale con cui poter comprendere il comportamento generale di un'applicazione MapReduce. In questa sezione si illustra come questo è implementato in Hadoop.

Nell'esempio precedente si è supposto di avere un file molto piccolo e che l'applicazione fosse eseguita in un unico nodo. Nella realtà quotidiana i dati sono molto più grandi e occupano più *DataNode*. È quindi lecito aspettarsi che più *TaskTracker* eseguano operazioni sui dati che hanno localmente.

Si utilizza il termine *job* per identificare una richiesta al *JobTracker* di un'operazione MapReduce in Hadoop. Con *task* ci si riferisce, invece, ad un'operazione ordinata dal *JobTracker* ed eseguita su un *TaskTracker*, come un mapper o un reducer. Un'applicazione che sottomette una richiesta di *job* al sistema è chiamata *client*. Il programmatore di questa ha preventivamente implementato mapper e reducer desiderati e i file operativi sono situati o sono stati collocati nell'HDFS.

Un'applicazione inizia il *job* contattando il *JobTracker* del sistema, specificando anche i dati su cui opera ed eventualmente il *numero di reducer* desiderato. Per smistare i vari *task* in maniera ottimale questo richiede la posizione di tutti i blocchi componenti i file, tramite il *NameNode*. Con questa informazione è in grado di selezionare i *TaskTracker* più opportuni in cui far eseguire mapper e reducer.

Il *numero di blocchi* su cui l'applicazione opera rappresenta anche il *minimo numero* di mapper che verranno programmati. L'applicazione può impostarne un numero ancora più alto, ma non può scendere al di sotto di questo. Il motivo è che, intuitivamente, il *JobTracker* tenta di programmare un'operazione di map per ogni blocco nel punto migliore in cui questa può

eseguire, ovvero il più possibile vicino al blocco stesso. Non sarebbe ottimale avere un numero di map inferiore al numero di blocchi su cui si opera, in quanto questo comporterebbe necessariamente il trasferimento di porzioni di file nella rete, per essere gestite da un *TaskTracker*.

Ad esempio, se il file testuale dell'esempio precedente fosse stato grande 8GB anziché pochi byte, con una grandezza di blocco pari a 64MB ci si dovrebbe aspettare un numero di mapper minimo pari a  $\frac{8192\text{MB}}{64\text{MB}} = 128$ , ognuno di essi operante su una porzione di file.

Su ogni *TaskTracker* viene mantenuta una *coda* di operazioni pendenti, ordinate dal *JobTracker* ma non ancora eseguite, ad esempio perché le risorse computazionali sono in uso da parte di un altro *task*. Il *JobTracker* tiene conto anche del riempimento della coda di un nodo nel programmare un'operazione su uno di essi. Anche dopo essere stata allocata, il master potrebbe annullarla o programmarla su un nuovo *TaskTracker* in caso di problemi di rete o di fallimento del *task*.

Ogni *TaskTracker* esegue il mapper appena ne ha l'opportunità, rendicontando periodicamente lo stato di avanzamento al *JobTracker*. Per accedere agli *input* della funzione di map, il nodo deve chiedere al *NameNode* l'accesso ai dati, che procederà a contattare i *DataNode* appropriati. Le funzioni di read sull'HDFS saranno portati a termine da questi, così come le operazioni di write che risulteranno come output della funzione di map.

Il numero di reducer impostato dall'applicazione determina il *numero di file di output* che ogni mapper produce. Ad esempio, in caso si desiderino otto reducer, i 128 mapper dell'esempio precedente produrranno ognuno otto file di output, uno per ogni reducer.

A loro volta, ognuno di essi produrrà come risultato *un singolo file*. È dunque opportuno bilanciare il numero di questi in base alla quantità di file in cui si vuole il risultato finale. Nell'esempio precedente, gli otto reducer produrrebbero un totale di otto file di risultato, dividendo dunque il "conteggio" delle parole in più parti.

Nonostante sia comune volere il risultato su un unico file, impostando dunque un *singolo* reducer, questo potrebbe rallentare l'esecuzione del lavoro, in quanto esso non sfrutterebbe più nodi in parallelo. Se ad esempio i 128 output dei mapper dovessero essere elaborati da un solo reducer, questo impiegherebbe un tempo considerevole a gestirli tutti. Un approccio più oculato consisterebbe nell'ammettere l'esecuzione di più funzioni di reduce con lo svantaggio di avere più file di risultato al termine del *job*.

L'output di ogni mapper è scritto sull'HDFS. Nel caso di *più* reducer, ogni file è *partizionato* per chiave in modo che, su accordo del *JobTracker*, ogni reducer possa lavorare su un insieme ben bilanciato di tuple. Ad esempio, in caso di chiavi ordinabili alfabeticamente, uniformemente distribuite su tutto l'alfabeto e con otto reducer, il primo file di output *di ogni* mapper potrebbe essere composto dalle chiavi a partire dalla A fino alla C, il secondo file conterrebbe chiavi dalla D alla F e così via.

Il funzionamento dei reducer è simile a quello dei mapper: il *JobTracker* ordina l'esecuzione di essi sui *TaskTracker* appropriati (ad esempio in nodi vicini a dove gli output dei mapper sono stati memorizzati) e questi, tramite *NameNode* e *DataNode*, accedono ai file di input per la funzione di reduce. Ogni reducer accede a *tutti e soli* i file di output prodotti *per lui* dai mapper, senza procedere alla lettura di quelli riservati ad altri task.

Durante la lettura, le tuple vengono trasformate nel formato opportuno per le reduce, aggregando ogni valore con chiave uguale. Il framework prevede anche il loro *ordinamento* durante questa trasformazione.

Infine, le funzioni di reduce vengono eseguite, e i loro output memorizzati sull'HDFS come menzionato in precedenza. La posizione viene restituita al *JobTracker*, che quindi notifica il *client* del termine dell'operazione e della locazione del risultato.

Si noti che se la presenza di più file come risultato non è gradita, ma è necessaria per velocizzare le operazioni, nulla vieta di scrivere una seconda applicazione MapReduce, operante sui *risultati dei reducer*, per elaborarli ulteriormente, questa volta con un numero inferiore di task richiesti. Ad esempio, gli otto risultati dei precedenti reducer possono essere passati in input a otto mapper, impostando questa volta un singolo reducer che produrrà il risultato finale in un unico file, come desiderato.

Si propone un diagramma riassuntivo delle varie fasi di un'operazione MapReduce nella figura 4.

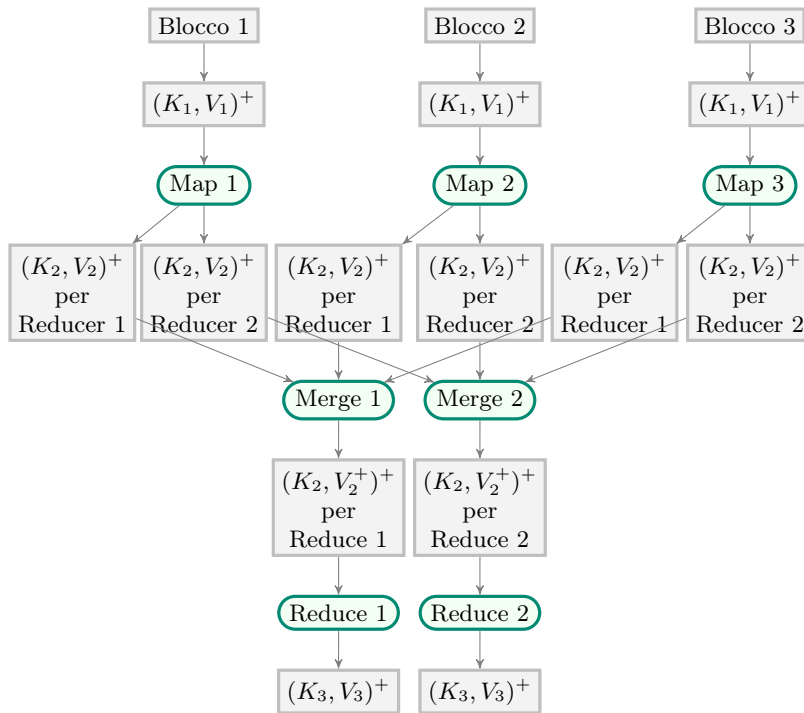
### Altri task e file di log

Hadoop MapReduce utilizza un ulteriore tipo di *task* per effettuare operazioni ausiliarie in precedenza o in seguito all'effettivo *job* da compiere. Si identifica con ***JobSetup*** il task responsabile di gestire le operazioni che precedono le decisioni da parte del *JobTracker* riguardanti lo smistamento dei vari mapper e reducer nel sistema. Ad esempio, si occupa di allocare spazio *temporaneo* per memorizzare informazioni necessarie all'inizializzazione di un *job*.

La ***JobCleanup*** esegue, invece, l'operazione opposta, occupandosi di rimuovere dal sistema tutto lo spazio utilizzato temporaneamente dal *JobTracker* o, in caso di fallimenti di task, elimina le risorse che erano state allocate ad un *TaskTracker*. La sua esecuzione ha luogo al termine del job, poco prima che il *JobTracker* restituisca il risultato al client.

Sebbene si tratti di operazioni secondarie, la loro presenza è stata monitorata durante la fase di validazione del benchmark.

Inoltre, è stato possibile esaminare le *operazioni* compiute dai vari componenti di Hadoop grazie ai loro *file di log*, memorizzati nell'hard disk della macchina in cui operano. Questi sono fondamentali per l'analisi del workload di un'applicazione MapReduce, come verrà commentato nel quarto capitolo.



**Figura 4:** Esempio grafico di un'operazione MapReduce su un file composto da tre blocchi e configurata con due reducer. È stata identificata con *merge* la fase di aggregazione dei valori e loro ordinamento, compiuta dai reducer. Le coppie (*chiave, valore*) sono state identificate secondo le definizioni di pagina 62. L'immagine è stata corposamente rielaborata da uno schema presente in [27].

## 2.3 Miscellanea

In chiusura di capitolo si descrivono brevemente due argomenti il cui uso è stato necessario per realizzare il prototipo del benchmark. Si definirà dapprima il concetto di *prodotto tra matrici* e in seguito si darà una panoramica sulla *generazione di numeri pseudo-casuali*.

### 2.3.1 Prodotto tra matrici

Secondo le definizioni dell'algebra lineare, una *matrice* è una *tabella ordinata* di elementi. Si utilizzano generalmente le lettere maiuscole in *nero matematico*, come  $\mathbf{A}$ , per riferirsi ad esse. La struttura degli elementi si rappresenta in questo modo:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Con la notazione  $a_{ij}$  si indica l'elemento della matrice  $\mathbf{A}$  situato alla  $i$ -esima riga e alla  $j$ -esima colonna. Ad esempio,  $a_{12} = 2$  mentre  $a_{23} = 6$ .

Una matrice composta da *numeri reali* con  $m$  righe e  $n$  colonne si indica con la notazione  $\mathbf{A} \in \mathbb{R}^{m \times n}$ . Intuitivamente, la matrice di esempio definita in precedenza può essere indicata con  $\mathbf{A} \in \mathbb{N}^{2 \times 3}$ .

Siano  $\mathbf{A} \in \mathbb{R}^{m \times p}$  e  $\mathbf{B} \in \mathbb{R}^{q \times n}$ . Il **prodotto tra matrici**, indicato con  $\mathbf{A} \cdot \mathbf{B}$ , è *non definito* se  $p \neq q$ , altrimenti è la matrice  $\mathbf{C} \in \mathbb{R}^{m \times n}$  i cui elementi sono definiti dalla formula:

$$c_{ij} = \sum_{k=1}^p a_{ik} \cdot b_{kj} \quad 1 \leq i \leq m \quad 1 \leq j \leq n \quad (2.1)$$

Se il prodotto è possibile, ovvero se  $p = q$ ,  $\mathbf{A}$  e  $\mathbf{B}$  sono dette *conformabili*.

Il benchmark utilizza il prodotto tra matrici per tutto il suo funzionamento, in quanto rappresenta una formula matematica di semplice comprensione, composta allo stesso tempo da calcoli intensivi, adatti a saturare la CPU del sistema; si può così simulare efficacemente un workload che richiede risorse computazionali notevoli. Il caricamento degli elementi delle matrici *operandi*  $\mathbf{A}$  e  $\mathbf{B}$  e la memorizzazione della matrice *prodotto*  $\mathbf{C}$  possono invece simulare un workload composto da letture e scritture su disco.

### 2.3.2 Generazione di numeri pseudo-casuali

Sia data una *sequenza* di numeri  $\{s_1, s_2, \dots, s_n\}$  e si ipotizzi di doverne *estrarre* un elemento alla volta. Questa si dice caratterizzata da **distribuzione uniforme** se la probabilità di *estrarre* un suo elemento è la stessa per *ognuno* di questi, ovvero ogni numero  $s_i$  ha probabilità  $\frac{1}{n}$  di essere *estratto*.

Se dato un elemento  $s_i$  estratto dalla sequenza precedente, caratterizzata da distribuzione uniforme, è *impossibile* predire il prossimo elemento che verrà estratto da essa, allora la sequenza è detta **di numeri casuali**.

Una sequenza con queste proprietà è, ad esempio,  $\{1, 2, 3, 4, 5, 6\}$  nel caso di un comune *dado da gioco*, nell'ipotesi che non vi siano elementi che portino a preferenze di un numero rispetto ad un altro durante i lanci. Gli elementi della sequenza sono quindi detti *numeri casuali*.

Nell'informatica la generazione di sequenze che presentano questa proprietà è molto difficile. Sebbene vi siano fenomeni *fisici* in grado di generare segnali, impulsi o di manifestare delle proprietà con pattern irriproducibili, un *software* in tal senso non è ancora stato inventato.

Un **generatore di numeri pseudo-casuali** è un algoritmo che genera sequenze di numeri il più possibile simili a quelle di numeri casuali. Sebbene siano in grado di *non far ripetere l'estrazione di nessun numero della sequenza già estratto* finché ogni elemento non lo è stato, il *pattern* con cui questa avviene è *completamente determinato* all'inizio dell'algoritmo.

La cardinalità della sequenza è chiamata *periodo* del generatore, ovvero il numero degli elementi diversi che esso è in grado di produrre prima che l'intera sequenza si ripeta. Questa definizione è adeguata solo nel caso di numeri espressi in forma aritmetica, assumendone l'infinità delle cifre che li compongono. Le limitate risorse a disposizione in un calcolatore, come l'uso della virgola mobile in doppia precisione, possono approssimare numeri diversi alle stesse quantità. In questo caso l'uguaglianza di due numeri  $s_i$  e  $s_j$  della sequenza non segna la fine del periodo, in quanto il numero  $s_{j+1}$  sarà diverso da  $s_{i+1}$ .

Un generatore è in grado di produrre una sequenza di numeri a partire da un certo *stato*, che ne determina l'intero pattern. Generalmente esso è rappresentato da uno o più numeri, chiamati *seed*. Nelle librerie software implementanti algoritmi di generazione di numeri pseudo-casuali è spesso richiesto che l'utilizzatore fornisca un seed per impostare una certa condizione iniziale all'algoritmo. Stessi seed causano lo stesso pattern di estrazione.

Un'opzione comune per la sua scelta è la rappresentazione della data e dell'ora correnti, che garantiscono per natura risultati diversi ad ogni inizializzazione del generatore. Si noti che, sebbene esso non sia effettivamente scelto "a caso", i numeri successivi seguiranno le proprietà discusse in precedenza.

Il benchmark utilizza ampiamente la generazione di numeri pseudo-casuali per determinare i valori degli elementi delle matrici con cui opera o il tipo di operazione da eseguire qualora sia necessario compiere una scelta. Tra i diversi algoritmi a disposizione si è scelto di utilizzare il *Mersenne Twister*<sup>2</sup> [24], sviluppato nel 1998 da Matsumoto e Nishimura.

La scelta è stata motivata dalla sua ampia diffusione (Python, Ruby, PHP e MATLAB lo utilizzano come generatore di default, ad esempio), dalla sua comprovata velocità e dalla sua nota efficienza. Il suo periodo è pari a  $2^{19937} - 1 \approx 4,3 \times 10^{6001}$ , ben oltre ogni possibilità di ripetizione del pattern in tutte le applicazioni informatiche, sebbene vi possano essere numeri "duplicati" per i motivi menzionati in precedenza.

Gli autori forniscono nella loro pagina web un'implementazione del loro algoritmo in *C* liberamente utilizzabile per qualunque scopo. Il suo funzionamento è piuttosto complesso; si rimanda a [24] per la sua comprensione.

Il benchmark utilizza tre funzioni in grado di generare numeri pseudo-casuali. Due di esse consentono, previa inizializzazione, la generazione di un *double* compreso nell'intervallo  $[0, 1)$  o  $[0, 1]$  rispettivamente (nel secondo caso il numero 1 è generabile, nel primo no).

Da questa quantità, qui denotata con  $n$ , è possibile ottenere un numero pseudo-casuale  $\bar{n}$  in un intervallo arbitrario  $[min, max)$  o  $[min, max]$ , a seconda della funzione utilizzata, con l'intuitiva formula

$$\bar{n} = (max - min) \cdot n + min.$$

<sup>2</sup><http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

Nel caso sia necessario un numero intero  $\tilde{n}$  nell'intervallo  $[min, max]$  è possibile utilizzare la prima funzione e la formula precedente per ottenere un numero in  $[min, max + 1)$ , quindi procedere con l'arrotondamento all'intero inferiore, in modo da avere  $\tilde{n} = \lfloor \bar{n} \rfloor$ .

La terza funzione, invece, consente la generazione in una variabile di tipo `unsigned long long` di un numero intero compreso nell'intervallo  $[0, 2^{64} - 1]$ . Si utilizzerà tale funzione nel codice 15.



## Capitolo 3

### Il benchmark realizzato

In questo capitolo si descrive la composizione del benchmark oggetto della tesi. Si delinearanno dapprima gli obiettivi del progetto, poi si approfondirà la struttura del prototipo realizzato.

Si tenga presente che questo rappresenta un'implementazione funzionante e in linea con lo scopo desiderato, ma limitato in alcune sue parti, che si elencano e discutono nel corso del capitolo.

Le importanti considerazioni sui meccanismi del sistema operativo, di cui si è parlato nel corso del secondo capitolo, verranno considerate nuovamente in questa sede per motivare alcune scelte che sono state prese durante l'implementazione, assieme a possibili varianti di essa che possono fornire risultati diversi, in base alle necessità.

#### 3.1 Descrizione del progetto

##### 3.1.1 Obiettivi

Il progetto ha lo scopo di realizzare un'applicazione, assimilabile ad un *benchmark*, che consenta di *generare* un *workload* arbitrario, deciso dall'utilizzatore dello strumento.

Tale workload può essere composto da:

- Operazioni **CPU-bound**, ovvero utilizzanti esclusivamente le risorse computazionali del sistema e la memoria volatile.
- Operazioni **I/O-bound**, ovvero utilizzanti in maniera intensiva funzioni di input/output, come letture e scritture su disco. Sono concesse operazioni su memoria volatile e l'uso della CPU nella stretta misura necessaria alla normale gestione di tali funzioni.
- Operazioni di **LOOP**, ovvero la ripetizione di una certa porzione di workload precedentemente generata.
- Operazioni di **FORK**, ovvero la generazione di un *processo* che eseguirà a sua volta un certo workload.

- Operazioni di **JOIN**, ovvero l'attesa di un *processo* precedentemente creato.

L'utente dell'applicazione può decidere la composizione di esse nel tempo, impostando il loro ordine e la loro durata.

Assieme alla *generazione del traffico* desiderato, lo strumento rendiconta quanto eseguito in modo grafico e testuale, tramite dei precisi profili temporali.

### 3.1.2 Innovazione e possibili usi

L'innovazione di un simile progetto è la possibilità di caratterizzare intuitivamente e formalmente il carico di lavoro da eseguire.

A differenza dei regolari benchmark di cui si è parlato a pagina 5, che eseguono uno *specifico* workload noto a priori, con questo strumento si vuole permettere all'utilizzatore di specificare e creare un *esatto* tipo di traffico, con la possibilità di analizzarlo in seguito con metriche non limitate al semplice *tempo di risposta*.

Lo stesso workload può essere generato e testato in sistemi diversi, anche senza privilegi amministrativi, per poi comparare i risultati. È possibile analizzare se quanto modellato si può riflettere nel sistema, e in che modo questo avviene. Ad esempio, una durata *attesa* può non essere effettivamente rispettata per via della convivenza di altre operazioni indipendenti dall'applicazione o più operazioni di I/O programmate contemporaneamente possono non essere eseguite assieme per via di limiti strutturali dell'architettura.

È possibile anche il meccanismo opposto, ovvero il perfezionamento del workload nell'unico sistema a disposizione. In caso di algoritmi realizzabili con carichi di lavoro di struttura diversa, è possibile utilizzare il benchmark con ognuno di essi e quindi studiare le loro prestazioni anziché replicare l'algoritmo in più versioni, una con uno specifico workload, e quindi testarle tutte. Questa procedura permette un notevole risparmio di tempo e l'estrazione della migliore struttura dei processi, delle loro dipendenze e del grado di parallelismo ottimale.

## 3.2 Considerazioni

### 3.2.1 Limiti

Gli obiettivi delineati, seppur innovativi, non sono *completi* al punto di caratterizzare un workload in ogni sua parte. Lo scopo del lavoro è, piuttosto, la *formazione una solida base* su cui espandere il progetto, che consenta l'aggiunta, in un secondo momento, di ulteriori funzionalità, troppo onerose per essere incluse in un lavoro di tesi. Si discuteranno i limiti dell'implementazione e alcuni possibili sviluppi futuri nelle conclusioni di questo lavoro.

### 3.2.2 Usabilità del prototipo

L'implementazione è correttamente funzionante; è stata testata in più sistemi Linux e produce i risultati desiderati (si veda, a tal scopo, il quarto capitolo). Nonostante ciò, il traffico da generare deve essere attualmente *codificato nel programma* tramite codice *C*, in quanto non è stato prodotto alcun sistema per definire un linguaggio formale con cui specificarlo, e quindi nessun *parser* dedicato.

Questo costituisce uno dei limiti fondamentali del prototipo, che verrà gestito prioritariamente nelle prossime versioni dell'applicazione.

### 3.2.3 Durata e calibrazione

In una prima versione del progetto si è pensato di far decidere all'utente dell'applicazione la *durata in millisecondi* di ogni operazione. Le deludenti implementazioni iniziali del prototipo con i *timer software* (si veda a pagina 57) hanno portato alla revisione del concetto di *durata da intervallo temporale a numero di iterazioni* di una certa operazione.

L'utilizzatore del software non deve specificare il tempo desiderato per un certo tipo di operazione (ad esempio 100ms di computazioni CPU-bound) bensì il numero di queste (ad esempio 30 operazioni CPU-bound).

Sarebbe possibile implementare un meccanismo, a cui ci si riferirà con il termine *calibrazione*, che calcoli la durata media in *ms* di una *singola operazione* di un certo tipo, e che quindi converta una specifica temporale in una iterativa. Ad esempio, con una durata media per un'operazione CPU-bound di 4ms sarebbe possibile convertire i 100ms dell'esempio precedente in  $\frac{100ms}{4ms} = 25$  iterazioni che il benchmark eseguirà senza l'uso di timer software.

La calibrazione potrebbe avvenire prima della generazione del traffico e continuare durante essa, in modo da aggiornare dinamicamente i tempi medi.

Questo meccanismo non è stato implementato nel prototipo e necessita di uno studio approfondito per determinare il modo migliore per realizzarlo. La sua mancanza costituisce un altro limite importante del progetto.

### 3.2.4 Gestione dell'I/O

Nella sezione 2.1.2 si è discusso di come le operazioni di input/output possano essere gestite dal sistema operativo in modo contro intuitivo, risultando in letture o scritture da disco non conformi a quanto voluto dall'utente. In particolare si è parlato di *page cache*, *read-ahead* e *write asincrona*.

È indubbio che la generazione di un'operazione I/O-bound deve necessariamente considerare la loro presenza. Se si intende eseguire una *vera* operazione di I/O è sicuramente opportuno disabilitare i primi due meccanismi e procedere con l'attivazione del *sincronismo* delle write.

L'utilizzatore potrebbe tuttavia voler analizzare il comportamento di un'applicazione che *usa* questi meccanismi, che d'altronde sono stati creati esplicitamente per ottimizzare le costose operazioni su disco.

Nell'implementazione del prototipo si è scelto di *utilizzare tutti i meccanismi*, in accordo con la seconda visione tra le precedenti. Si potrebbe comunque procedere alla loro disattivazione in modo molto semplice, inserendo delle opzioni aggiuntive alla funzione di apertura dei file (si veda a pagina 48) e configurando opportunamente le quantità di byte letti e scritti.

### 3.2.5 Matrici e loro uso

Le matrici, di cui si è parlato a pagina 67, costituiscono l'elemento su cui poggia l'intera composizione delle fasi CPU-bound e I/O-bound. Le prime eseguono il prodotto tra due matrici i cui elementi sono contenuti in memoria, mentre le seconde leggono i due operandi memorizzati sul disco o vi scrivono il loro prodotto.

La struttura delle matrici in memoria e su disco non è rilevante ai fini della realizzazione del benchmark, e pertanto verrà ignorata. Per comprenderne il funzionamento, basti sapere che i loro elementi sono memorizzati in strutture apposite e che le loro dimensioni sono generate come desiderato dall'utente o con numeri pseudo-casuali.

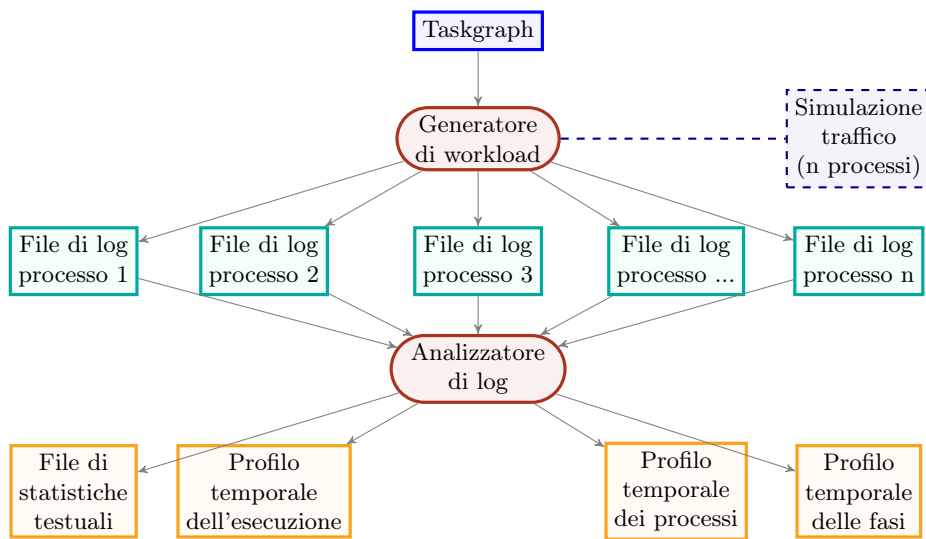
Gli elementi delle matrici sono contenuti in variabili `double` e generati anch'essi pseudo-casualmente. L'utilizzatore specifica se, prima della generazione del workload da simulare, le matrici devono essere create o se sono già esistenti su disco.

Processi *diversi* utilizzano matrici *diverse*, evitando ogni problema di sincronizzazione e di conflitto tra loro. Si noti che *non* è obiettivo del prototipo generare una *completa* moltiplicazione tra matrici: questa viene simulata, ma al termine del programma i risultati memorizzati su disco *non corrispondono* alla corretta moltiplicazione delle due matrici operande in memoria.

## 3.3 Architettura

Il benchmark è organizzato secondo lo schema illustrato nella figura 5, che si commenta di seguito:

- Il *taskgraph* è un file contenente, in un opportuno formato, la descrizione del workload da riprodurre e analizzare.
- Il *generatore di workload* riproduce il carico di lavoro come indicato nel *taskgraph*, e al termine dell'esecuzione produce *un file di log* per *ogni* processo specificato.



**Figura 5:** Schema architetturale del benchmark.

- I *file di log* contengono informazioni relative al comportamento di ogni operazione eseguita nel sistema, in un formato adatto alla loro elaborazione.
- L'*analizzatore di log* elabora tali file per produrre dei risultati comprensibili dall'utilizzatore.
- Il *file di statistiche testuali* e i *profili temporali* permettono la descrizione, rispettivamente testualmente e graficamente, dell'andamento nel tempo delle operazioni eseguite dal generatore di workload.

Generatore di workload e analizzatore di log sono due applicazioni *separate*. È dunque possibile utilizzare una sola delle due, in base alle necessità dell'utente.

Il resto del capitolo descrive nel dettaglio i vari componenti che compongono il benchmark.

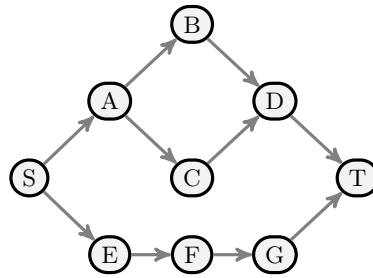
## 3.4 Il taskgraph

### 3.4.1 Definizione

Nella teoria dell'informazione, un *taskgraph*<sup>1</sup> è un grafo senza cicli i cui *nod*i rappresentano delle computazioni (chiamate anche *task*), mentre i suoi archi orientati rappresentano le dipendenze tra i nodi.

La figura 6 ne mostra un esempio. In questa, le lettere maiuscole denotano dei generici *task* composti da operazioni arbitrarie. Ogni *task* situato al

<sup>1</sup>Anche chiamato *task graph* o *Directed Acyclic Graph (DAG)*.



**Figura 6:** Esempio di taskgraph secondo la definizione classica.

termine di una freccia non può iniziare finché quello posto all’inizio di essa non ha concluso. Nel caso di più frecce *entrant*i in un nodo, il task associato deve attendere la conclusione di tutti i task che le originano; mentre in caso di più frecce *uscent*i da un nodo tale task abilita l’esecuzione di più operazioni a seguire.

Ad esempio, il task F non può eseguire fino alla conclusione di E, mentre C non può iniziare se non al termine di A. D può iniziare solo al termine sia di B sia di C.

È anche possibile determinare i *task* con possibilità di esecuzione parallela, ad esempio B e C possono essere eseguiti contemporaneamente mentre E e F non possono.

Il benchmark utilizza una sorta di “taskgraph arricchito”, di struttura simile, ma in grado di fornire informazioni più dettagliate. In questo testo se ne propone una diversa definizione, che si utilizzerà nel resto dell’elaborato.

Definiamo *taskgraph* un *insieme di liste* di elementi atomici chiamati *fasi*. Ogni *fase* è caratterizzata da informazioni diverse in base al suo *tipo*, che può essere CPU-bound, I/O-bound, LOOP, FORK o JOIN con lo stesso significato dato a pagina 71.

### 3.4.2 Implementazione

Nel benchmark ogni fase è memorizzata in una *struttura* adeguata. Più strutture sono collocate in memoria formando una *lista*. Ogni lista descrive il workload che *un singolo processo* dovrà eseguire. L’insieme delle liste di ogni processo costituisce il taskgraph.

Sono state definite quattro strutture diverse, ognuna adatta ad un certo *tipo* di fase e commentate nel resto della sezione.

Le fasi di tipo CPU-bound e I/O-bound condividono la struttura denominata `OrdinaryPhase`, i cui campi sono visibili nel dettaglio nel codice 10.

I primi due elementi della struttura, `type` e `id`, sono presenti anche in tutte le altre strutture che saranno esaminate in seguito. Il tipo `PhaseType` è un `enum` composto dai valori `CPU`, `IO`, `LOOP`, `FORK` e `JOIN`, ed è necessario per

**Codice 10:** Elementi della struttura `OrdinaryPhase`.

---

```

1 typedef struct OrdinaryPhase {
2     PhaseType type;           //"CPU o "IO"
3     int id;
4     long int value;
5     void *next;
6 } OrdinaryPhase;

```

---

caratterizzare il tipo di fase che contiene. In `OrdinaryPhase`, per costruzione, sono ammessi solo i primi due valori.

Ogni fase è inoltre caratterizzata da un *identificatore* unico, contenuto in `id`. Fasi diverse hanno identificatori diversi.

Il campo `value` è utilizzato per specificare il numero di *iterazioni* che il simulatore di workload dovrà effettuare durante l'elaborazione della fase, ovvero il numero di operazioni di un certo tipo che la fase specifica. Il puntatore `next` rende possibile collegare in una lista più strutture diverse. Il suo tipo è un puntatore generico, per permettere l'indirizzamento di una qualunque tra le diverse strutture definite per le altre tipologie di fasi.

Una fase di tipo LOOP è memorizzata nella struttura `LoopPhase`, descritta nel codice 11.

**Codice 11:** Elementi della struttura `LoopPhase`.

---

```

1 typedef struct LoopPhase {
2     PhaseType type;           //"LOOP"
3     int id;
4     long int counter;
5     long int iterationsNumber;
6     void *loopTo;
7     void *next;
8 } LoopPhase;

```

---

La definizione di una fase di LOOP comporta la deviazione del flusso di esecuzione del programma ad una fase definita in precedenza anziché l'elaborazione della seguente. Questo avviene un numero di volte pari al contenuto del campo `iterationsNumber`, ed è possibile specificare la struttura della fase che è necessario *ripetere* tramite il suo indirizzo, memorizzato in `loopTo`.

Alla creazione della struttura, il campo `counter` viene inizializzato con valore pari a `iterationsNumber`, per essere poi diminuito di un'unità ad ogni ripetizione. Quando questo raggiunge lo zero, la fase memorizzata in `loopTo` non viene più ripetuta e l'esecuzione prosegue con la fase successiva nel taskgraph, ovvero quella indirizzata da `next`.

In questa struttura il campo `type` contiene il valore `LOOP`.

Il simulatore di traffico inizia la propria esecuzione con *un singolo processo*. Se l'utente vuole generare nuovi processi deve includere nel taskgraph una fase di tipo `FORK`, memorizzata nella struttura `ForkPhase` e illustrata nel codice 12.

**Codice 12:** Elementi della struttura `ForkPhase`.

---

```

1 typedef struct ForkPhase {
2     PhaseType type;           //"FORK"
3     int id;
4     PointerToArrayOfPointersToVoid newPhases;
5     int numberOfNewProcesses;
6     void *next;
7 } ForkPhase;

```

---

Quando il simulatore incontra una fase di questo tipo crea un numero di processi pari a `numberOfNewProcesses`. Gli indirizzi delle strutture delle fasi *iniziali* di ognuno di questi è contenuto in un *array* di puntatori memorizzato all'indirizzo `newPhases`. Dopo la creazione di tutti i processi specificati, l'esecuzione prosegue alla prossima fase, di struttura memorizzata all'indirizzo contenuto in `next`.

Intuitivamente, `type` per questo tipo di struttura è sempre pari a `FORK`.

Infine, una fase di tipo `JOIN` è memorizzata in una struttura `JoinPhase`, mostrata nel codice 13.

**Codice 13:** Elementi della struttura `JoinPhase`.

---

```

1 typedef struct JoinPhase {
2     PhaseType type;           //"JOIN"
3     int id;
4     PointerToArrayOfPointersToVoid initialPhases;
5     int numberOfProcessesToJoin;
6     void *next;
7 } JoinPhase;

```

---

Come si può notare, la sua composizione è molto simile alla struttura precedente. Una `JOIN` ferma l'esecuzione corrente finché i processi *le cui fasi iniziali* sono contenute in `initialPhases` non sono terminati. Quest'ultima variabile è di tipo analogo a `newPhases` definito per le `FORK`, e il numero di processi da attendere è specificato in `numberOfProcessesToJoin`. Il campo `type` è fissato a `JOIN` per questa struttura.



### 3.4.3 Un esempio operativo

Si riporta un esempio grafico di taskgraph nella figura 7, con l'intento di consolidare i concetti fino ad ora introdotti.

Questa contiene in forma grafica tutte le informazioni che l'utente deve specificare per ogni fase.

Le fasi sono espresse tramite cerchi colorati, in base al loro tipo (si veda la didascalia della figura). Nella porzione superiore di ogni fase vi sono gli *id*, assegnati manualmente dall'utilizzatore del programma. Nell'esempio si è scelto di assegnare ad ogni *id* 3 cifre, la prima indicante il *numero del processo* e le altre il *numero della fase* relativo ad esso. Ad esempio, un *id* pari a 402 indica la seconda fase (02) definita per il quarto processo (4) da generare.

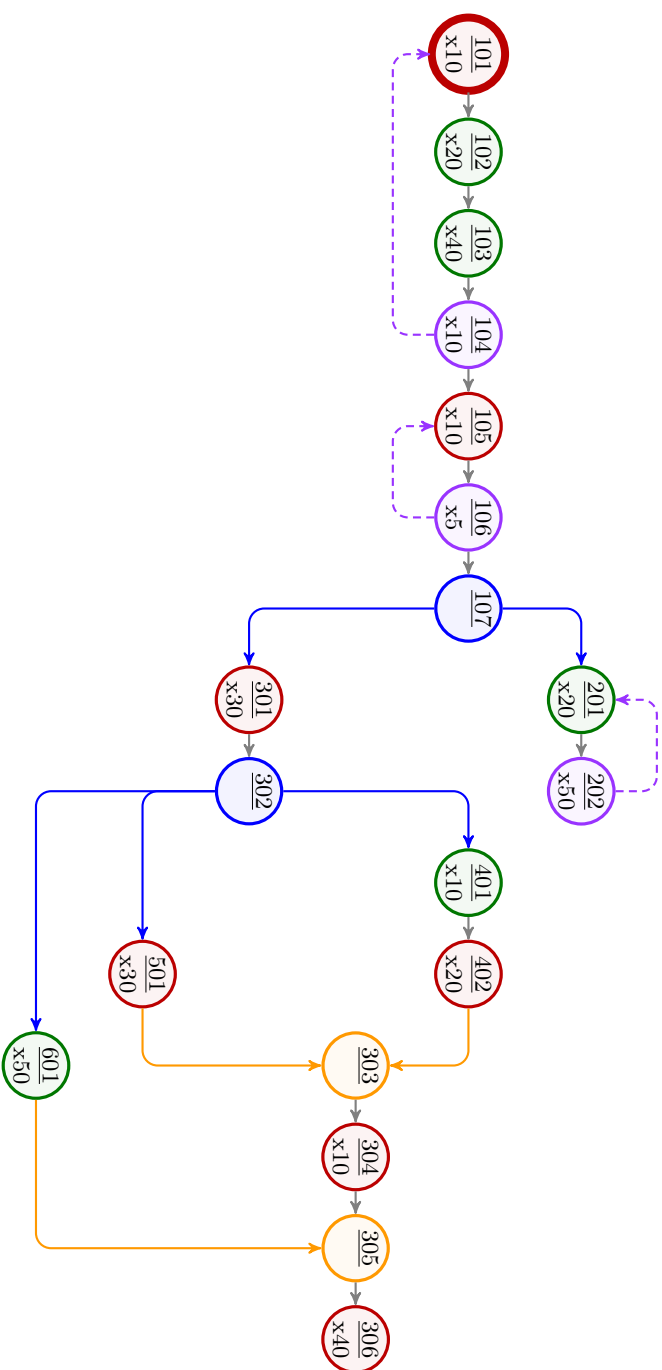
Nella parte inferiore di ogni cerchio è contenuto il numero di *iterazioni* per le fasi CPU-bound e I/O-bound, o il numero di *ripetizioni* per quelle di tipo LOOP. Ad esempio, la fase 105 eseguirà 10 operazioni di I/O (x10).

La freccia tratteggiata nelle fasi LOOP specifica a quale fase l'esecuzione deve essere deviata se **counter** contiene un valore superiore a 0, mentre le frecce uscenti dalle fasi FORK o entranti nelle fasi JOIN del colore corrispondente ad esse specificano le fasi iniziali dei processi da creare o i processi da attendere, rispettivamente. Ad esempio, la fase di LOOP 104 specifica che al suo raggiungimento e per 10 volte il flusso dell'esecuzione deve essere ripreso dalla fase 101; la fase di FORK 302 crea tre nuovi processi, iniziati rispettivamente con le fasi 401, 501 e 601; mentre la fase di JOIN 303 causa l'attesa dei processi 4 e 5.

La prima fase del taskgraph (101) è caratterizzata da un bordo più spesso rispetto agli altri, mentre le frecce in grigio specificano l'ordine tra le fasi, ovvero la struttura dei puntatori **next** definiti. Un cerchio senza una freccia entrante di questo tipo definisce una fase iniziale per un processo, mentre uno senza freccia uscente ne segnala l'ultima. Ad esempio la fase 401, senza freccia grigia entrante, è una fase iniziale, mentre la fase 202, senza freccia uscente, è finale. La fase 601, sprovvista di ogni freccia grigia, è l'unica che il sesto processo eseguirà.

Si noti come, sebbene l'utente abbia la possibilità di riferirsi alle fasi tramite il loro *id*, queste siano referenziate in memoria tramite *indirizzi*. Ad esempio, l'utente specificherà nel taskgraph che la fase di LOOP 106 deve ripetere l'esecuzione dalla fase 105, ma la struttura in memoria di **LoopPhase** prevede per il campo **loopTo** il *puntatore* alla struttura della fase 105, e non il suo *id*. La conversione tra indirizzi e *id* è possibile tramite una *lookup table* che associa i due campi. È il benchmark che gestisce il suo mantenimento durante la creazione delle strutture.

Come già accennato, il taskgraph dovrebbe essere definito in un file separato e letto all'avvio del generatore di workload da un *parser*. Dovrebbero, inoltre, essere compiti di quest'ultimo ricostruire in memoria le varie fasi nelle



**Figura 7:** Descrizione grafica di un taskgraph di esempio. Il colore dei cerchi rappresenta il tipo di fase, secondo la legenda: CPU, I/O, LOOP, FORK e JOIN. Per commenti e ulteriori spiegazioni della figura, si veda a pagina 79.

strutture esaminate nella sezione precedente, mantenere *per ogni processo* l'indirizzo della struttura della sua prima fase e quindi permettere l'esecuzione della *prima fase* del *primo processo* (la 101 nell'esempio) passando il suo indirizzo al generatore di workload.

Nel prototipo attuale è cura dell'utilizzatore inizializzare le strutture in memoria, mentre il resto è svolto automaticamente dal benchmark.

### 3.5 Il generatore di workload

Una volta ricostruito in memoria il *taskgraph*, il **generatore di workload** riceve l'indirizzo della struttura della prima fase da eseguire. Le operazioni previste per essa vengono riprodotte, quindi la lista viene scorsa per consentire l'esecuzione della fase successiva, ripetendo quindi il procedimento. Il generatore si preoccupa inoltre di memorizzare alcune informazioni relative all'esecuzione delle fasi, che verranno scritte nei *file di log* al termine della vita di ogni processo.

Nel resto della sezione si approfondiscono le operazioni compiute dal generatore di workload. Oltre ad una descrizione testuale, si fornirà l'implementazione delle funzioni più importanti di questo componente. Il codice è semplificato, volto a mettere in evidenza il metodo utilizzato, piuttosto che la sua esatta implementazione. Si tralasceranno dunque dettagli superflui ai fini della comprensione, come la gestione dei possibili errori o la struttura delle matrici. Talvolta si introdurranno delle funzioni senza approfondire il relativo codice, si ometteranno i loro parametri se non fondamentali o si introdurranno delle funzioni adatte a pseudo-codici per riassumere operazioni in realtà più complesse, ma non importanti.

In tutti i codici si è scelto di evidenziare in viola le funzioni che verranno analizzate in seguito o che già lo sono state in questo capitolo, in rosso quelle messe a disposizione dal sistema operativo e analizzate nel secondo capitolo e in verde le funzioni fittizie o il cui codice non viene mostrato, accontentandosi di una descrizione testuale del loro funzionamento.

#### 3.5.1 Riproduzione del traffico

L'operazione fondamentale del generatore di workload è la riproduzione del traffico descritto dal *taskgraph*. La procedura centrale che gestisce il flusso di esecuzione di ogni processo è chiamata `simulate`, ed è mostrata nel codice 14.

Si noti innanzitutto che la sua *signature* è adatta affinché possa essere impostata come funzione iniziale di un nuovo processo tramite la funzione `clone`, in quanto ammette un puntatore generico come unico argomento (si veda a pagina 46).

La funzione si preoccupa di scorrere la lista delle fasi in memoria per *un processo* ed eseguire il tipo di traffico appropriato. La *prima fase* del

**Codice 14:** Codice semplificato della procedura `simulate`.

---

```

1 void simulate(void *root) {
2     OrdinaryPhase *pointer;
3     void *generic = root;
4     Statistics *firstStatistics = malloc(sizeof(Statistics));
5     Statistics *currentStat = firstStatistics;
6
7     do {
8         pointer = generic;
9         if (pointer->type == CPU) {
10            OrdinaryPhase *current = generic;
11            setOrdinaryStartTimestamp(currentStat, current->id, CPU,
12                current->value);
13            executeCPUPhase(current);
14            currentStat = setOrdinaryEndTimestamp(currentStat);
15            generic = current->next;
16        }
17        else if (pointer->type == IO) {
18            OrdinaryPhase *current = generic;
19            setOrdinaryStartTimestamp(currentStat, current->id, IO,
20                current->value);
21            executeIOPhase(current);
22            currentStat = setOrdinaryEndTimestamp(currentStat);
23            generic = current->next;
24        }
25        else if (pointer->type == LOOP) {
26            LoopPhase *current = generic;
27            generic = executeLOOPPhase(current);
28        }
29        else if (pointer->type == FORK) {
30            ForkPhase *current = generic;
31            setForkJoinStartTimestamp(currentStat, current->id, FORK);
32            PointerToArrayOfPIDs result = executeFORKPhase(current);
33            currentStat = setForkJoinEndTimestamp(currentStat, result);
34            generic = current->next;
35        }
36        else if (pointer->type == JOIN) {
37            JoinPhase *current = generic;
38            setForkJoinStartTimestamp(currentStat, current->id, JOIN);
39            PointerToArrayOfPIDs result = executeJOINPhase(current);
40            currentStat = setForkJoinEndTimestamp(currentStat, result);
41            generic = current->next;
42        }
43    } while (generic != NULL);
44    writeLogFile(firstStatistics);
45 }

```

---

processo è indirizzata dall'argomento `root`, di tipo generico. L'indirizzo della fase *corrente* è invece memorizzato tramite i puntatori `pointer`, `current` e `generic`. La differenza tra essi è il loro tipo: il primo è sempre di dominio `OrdinaryPhase`, il secondo cambia in base al reale tipo della struttura mentre il terzo è sempre un generico `void *`.

Poiché i campi `type` e `id` sono i primi due in ogni struttura, la funzione può assumere che una nuova fase sia sempre di tipo `OrdinaryPhase` (linea 8) leggendo quindi il campo `type` della struttura anche se questo contiene valori differenti da `CPU` o `IO` (linee 9, 17, 25, 29 e 36). Poiché il contenuto potrebbe risultare diverso dai due valori precedenti (ovvero la fase potrebbe non essere CPU-bound o I/O-bound) un puntatore del tipo corretto (`current`) viene dichiarato in seguito alla determinazione della fase (linee 10, 18, 26, 30 e 37) e viene utilizzato nelle operazioni a seguire. La lettura del campo `next` di una struttura, di tipo `void *`, avviene invece tramite il puntatore `generic`, adatto a leggere un puntatore generico (linee 15, 23, 27, 34 e 41).

Per quasi tutte le fasi il generatore di workload esegue tre operazioni fondamentali, in sequenza:

1. Memorizza l'*uptime* del sistema *prima* di iniziare l'esecuzione della fase.
2. Esegue il traffico adeguato.
3. Memorizza l'*uptime* del sistema *dopo* aver eseguito la fase, assieme ad altre informazioni se necessario.

Il primo punto (linee 11, 19, 31 e 38) e il terzo (linee 14, 22, 33 e 40) verranno commentati nella sezione 3.5.2, mentre il secondo (linee 13, 21, 27, 32 e 39) comporta la chiamata di funzioni descritte nel seguito di questa sezione.

La fine della lista è concretizzata dal valore `NULL` in uno dei puntatori `next` delle strutture. Alla sua occorrenza avviene la memorizzazione su disco di tutte le informazioni collezionate dai punti 1 e 3 descritti in precedenza, tramite la funzione `writeLogFile` (linea 44). Anche questa, assieme alla struttura `Statistics` (linee 4 e 5), sarà esaminata nella sezione successiva.

### Fasi CPU-bound

La funzione `executeCPUPhase` genera un workload di tipo CPU-bound, composto, come già accennato, da moltiplicazioni di matrici formate da elementi di tipo `double`.

Il suo funzionamento consiste nel generare due numeri pseudo-casuali interi,  $i$  e  $j$ , nell'intervallo  $[1, m]$  e  $[1, n]$  rispettivamente (dove con  $m$  si denota il numero di righe della matrice prodotta definita per il processo, mentre con  $n$  le sue colonne) che identificano in questo modo l'elemento  $c_{ij}$  della formula 2.1 a pagina 68.

Una volta scelto l'elemento, tale formula è applicata per produrre il risultato. Questo procedimento è ripetuto *per ogni iterazione*, il cui numero è specificato nella variabile `current->value`. La funzione calcola dunque il valore corretto di *un* elemento della matrice prodotto *per iterazione*.

Si è scelto di non mostrare la sua implementazione in quanto contiene del codice altamente specifico alla modalità di memorizzazione delle matrici in memoria, inutile ai fini della trattazione. Si ricordi comunque che, come commentato nella sezione 3.2.5, queste sono già presenti in memoria al momento dell'esecuzione della fase, composte da elementi di valore pseudo-casuale.

### Fasi I/O-bound

La generazione di un traffico adatto alle fasi I/O-bound è ad opera della funzione `executeIOPhase`, mostrata nel codice 15.

Questa genera un numero pseudo-casuale intero `rand1` (linea 26) nell'intervallo  $[0, 2^{64} - 1]$  tramite l'invocazione del Mersenne Twister (si veda la sezione 2.3.2) qui concretizzata dall'intuitiva funzione `randomInteger`, quindi controlla la sua parità o disparità tramite `isEven`. Nel primo caso esegue una lettura da disco (linea 27), mentre nel secondo una scrittura (linea 40).

La distribuzione dei numeri nell'intervallo garantisce la stessa probabilità di estrazione di numeri pari e dispari, rendendo così equiprobabili scritture e letture. Mentre l'unica matrice a poter essere scritta su disco è quella prodotto, la lettura può interessare uno qualsiasi dei due operandi. In questo caso un ulteriore numero pseudo-casuale, `rand2`, è estratto similmente a prima (linea 28) per decidere da quale delle due matrici effettuare la lettura (linee 29 e 34). Nel codice si sono indicati con A e B i due operandi; mentre C designa la matrice prodotto.

Come tutte le funzioni che generano traffico, questa riceve come parametro il puntatore alla struttura della fase corrente, qui chiamato `pointer`. La funzione itera un numero di volte pari a `pointer->value` (linea 25) eseguendo quanto descritto in precedenza ad ogni iterazione.

Dopo aver letto un elemento su disco da una matrice operanda, la funzione lo memorizza in una struttura appropriata affinché una operazione CPU-bound possa accedervi (linee 32 e 37). La procedura `memorizeElement` schematizza questo procedimento.

Similmente, la scrittura di un elemento della matrice prodotto su disco è preceduta dalla sua selezione con procedimento pseudo-casuale e quindi dalla sua memorizzazione in una struttura adeguata per la scrittura su disco (linea 41). La funzione `retrieveElement` si preoccupa, oltre a selezionare l'elemento da scrivere, anche di comunicare la sua posizione (tramite le variabili `r` e `c`), in modo che una scrittura possa avvenire a partire dal giusto *offset* del file.

La funzione `readMatrix` gestisce una lettura da un file contenente una delle due matrici operandi. La funzione `executeIOPhase`, dopo aver determinato

**Codice 15:** Esecuzione delle fasi I/O-bound.

---

```

1 double readMatrix(int fileDescriptor, int rows, int columns) {
2     double *value = malloc(sizeof(double));
3     int row = randomInteger(1,rows);
4     int column = randomInteger(1,columns);
5     long int offset = computeOffset(column,row,...);
6     pread(fileDescriptor,(double *)value,sizeof(double),offset);
7     double val = *value;
8     free(value);
9     return val;
10 }
11
12 void writeMatrix(int fileDescriptor, double val, int row, int column)
13 {
14     double *value = malloc(sizeof(double));
15     *value = val;
16     long int offset = computeOffset(column,row,...);
17     pwrite(fileDescriptor,(double *)value,
18           sizeof(double),offset);
19     free(value);
20 }
21 void executeIOPhase(OrdinaryPhase *pointer) {
22     double value;
23     int r,c;
24     unsigned long long rand1, rand2;
25     for (long int i = 0; i < pointer->value; i++) {
26         rand1 = randomInteger(0,2^64-1);
27         if (isEven(rand1)) { //Leggi
28             rand2 = randomInteger(0,2^64-1);
29             if (isEven(rand2)) { //Leggi da A
30                 value = readMatrix(getDescriptor(A,current),
31                                   getRows(A,current), getColumns(A,current));
32                 memorizeElement(value,...);
33             }
34             else { //Leggi da B
35                 value = readMatrix(getDescriptor(B,current),
36                                   getRows(B,current), getColumns(B,current));
37                 memorizeElement(value,...);
38             }
39         }
40         else { //Scrivi su C
41             value = retrieveElement(&r,&c,...);
42             writeMatrix(getDescriptor(C,current), value, r, c);
43         }
44     }
45 }

```

---

**Codice 16:** Esecuzione delle fasi LOOP.

---

```
1 void * executeLOOPPhase(LoopPhase *pointer) {
2   if (pointer->iterationsNumber > 0) {
3     if (pointer->counter > 0) {
4       pointer->counter--;
5       return pointer->loopTo;
6     }
7     else if (pointer->counter == 0) {
8       pointer->counter = pointer->iterationsNumber;
9       return pointer->next;
10    }
11  }
12 }
```

---

quale matrice sia da leggere, ne richiede il *descrittore* del file e il numero di righe e di colonne, procedimento schematizzato dalle funzioni `getDescriptor`, `getRows` e `getColumns` (linee 30-31, 35-36). La `readMatrix` può quindi generare pseudo-casualmente una riga `row` e una colonna `column` compresa tra intervalli corretti (linee 3-4), che costituiscono la posizione dell'elemento da leggere all'interno della matrice.

A questo punto è necessario convertire `row` e `column` nella posizione in cui iniziare la lettura della sequenza di byte memorizzata su disco (si veda a pagina 49), calcolando il valore della variabile `offset`. La funzione `computeOffset` (linea 5) si preoccupa di ciò e non viene mostrata in quanto è specifica al formato di memorizzazione delle matrici.

Ora che tutto il necessario è disponibile, la funzione invoca la `pread`. Viene letto un `double`, memorizzato nell'area di memoria indirizzata da `value` (linea 6), il cui valore viene, quindi, restituito al chiamante (linea 9) ed è necessario per la sua collocazione in memoria nelle strutture adeguate, ad opera della `memorizeElement`.

La scrittura avviene con un meccanismo simile alla lettura tramite la funzione `writeMatrix`, con la differenza che l'elemento da memorizzare è già stato scelto dalla `retrieveElement` che fornisce il suo valore in `value` e la sua posizione `r` e `c`. La funzione si limita, dunque, a calcolare la posizione corretta in cui inserirlo nel file (linea 15) e quindi a chiamare la `pwrite` (linee 16-17; si veda a pagina 51).

## Fasi LOOP

Una fase di tipo LOOP causa un cambiamento nel flusso dell'esecuzione di un processo, determinando esplicitamente la prossima fase prevista per esso. Il suo funzionamento è mostrato nel codice 16.



**Codice 17:** Esecuzione delle fasi FORK.

---

```

1 PointerToArrayOfPIDs executeFORKPhase(ForkPhase *pointer) {
2   PointerToArrayOfPIDs pidArray =
3     malloc(sizeof(pid_t)*pointer->numberOfNewProcesses);
4   for (int i=0; i<pointer->numberOfNewProcesses; i++) {
5     void (*pf) (void *);
6     pf = &simulate;
7     void *stack = malloc(STACK_SIZE);
8     void *arg = (*pointer->newPhases)[i];
9     pid_t result = clone(pf,stack + STACK_SIZE, SIGCHLD|CLONE_FS|
10      CLONE_VM|CLONE_FILES, arg);
11     (*pidArray)[i] = result;
12     setLookupTable(result, (*pointer->newPhases)[i]);
13   }
14   return pidArray;
15 }

```

---

A differenza di tutte le altre, al termine di una fase LOOP la procedura `simulate` non scorre la lista. L'indirizzo della prossima fase da eseguire è determinato dalla stessa `executeLOOPPhase`, che restituisce come valore di ritorno il puntatore ad essa (linea 27 del codice 14).

Come già accennato, al momento della generazione in memoria della struttura `LoopPhase`, i campi `counter` e `iterationsNumber` sono entrambi inizializzati al numero di ripetizioni deciso dall'utente. Se `counter` è maggiore di zero (linea 3) la funzione restituisce il puntatore `loopTo` così da permettere la ripetizione di questa (linea 5). Come operazione aggiuntiva, decrementa `counter` di un'unità (linea 4) in modo che, una volta esaurito il numero di ripetizioni desiderato, la fase possa restituire il puntatore `next` (linea 9), proseguendo con l'esecuzione del resto del taskgraph. Prima di ciò "ricarica" `counter` al suo valore iniziale, in modo che eventuali "loop innestati" possano avvenire senza problemi (linea 8).

## Fasi FORK

Una fase FORK causa la generazione di un nuovo processo tramite l'esecuzione della funzione `executeFORKPhase`, mostrata nel codice 17.

I `pid` dei processi creati (si veda a pagina 42) sono restituiti dalla funzione in modo che possano essere utilizzati dalla `setForkJoinEndTimestamp`, che li memorizzerà assieme alle informazioni temporali associate alla fase, come mostrato nella sezione 3.5.2 (linee 32 e 33 del codice 14).

Questi sono contenuti in un *array di variabili* `pid_t`, il cui puntatore (indicato intuitivamente dal tipo `PointerToArrayOfPIDs`) costituisce l'effettivo valore di ritorno della funzione. L'inizializzazione dell'array è la prima operazione compiuta (linee 2-3).

Si è accennato in precedenza che una fase FORK crea un nuovo *light-weight process* con il grado di condivisione delle strutture con il padre il più alto possibile, tramite la funzione `clone` (si veda a pagina 46). La `executeFORKPhase` si preoccupa di gestire quanto necessario alla sua chiamata, come specificato dal codice 1.

Poiché ogni nuovo *light-weight process* dovrà eseguire la funzione `simulate`, la `executeFORKPhase` crea un puntatore `pf` ad essa, memorizzando al suo interno l'indirizzo di questa (linee 5 e 6). Alloca quindi un'area di memoria che costituirà lo stack del nuovo processo, memorizzando in `stack` il suo indirizzo iniziale (linea 7). La macro `STACK_SIZE` specifica la sua dimensione, in byte.

L'argomento della funzione `simulate` è l'indirizzo della prima fase che il nuovo processo dovrà eseguire, contenuto nell'array indirizzato da `newPhases`. Questo è letto e quindi memorizzato in `arg` (linea 8).

A questo punto la funzione `clone` può essere chiamata (linee 9 e 10). Il suo terzo parametro è composto da una maschera di bit ottenuta tramite l'or *bitwise* di alcune macro e specifica le opzioni da trasmettere al figlio. Quelle indicate nel codice hanno il seguente significato:

- `SIGCHLD` specifica l'invio dell'omonimo segnale al padre quando il processo termina. È necessario per permettere il funzionamento di una `waitpid`, ma l'argomento non verrà approfondito ulteriormente.
- `CLONE_FS` permette di condividere con il padre le informazioni del *file system*.
- `CLONE_VM` permette di condividere con il padre la memoria operativa, mantenendo comunque separato lo stack.
- `CLONE_FILES` permette di condividere con il padre i *descrittori dei file*.

Grazie al valore di ritorno della funzione `clone` è possibile conoscere il `pid` del processo creato, che verrà memorizzato in `pidArray` (linea 11) in attesa di essere restituito alla `simulate`.

La funzione `executeFORKPhase` si preoccupa, inoltre, di associare il `pid` del processo creato, ora noto, all'indirizzo della sua *prima fase* tramite una lookup table con la funzione `setLookupTable` (linea 12). Questa tabella verrà utilizzata dalle fasi di tipo JOIN per conoscere il `pid` da aspettare, in quanto, come già detto, la struttura `JoinPhase` contiene l'indirizzo della sua *prima fase* e non il `pid` stesso.

Questo procedimento è compiuto per ogni processo da generare nella fase (linea 4), e al termine l'elenco dei `pid` generati è restituito al chiamante (linea 14).

**Codice 18:** Esecuzione delle fasi JOIN.

---

```

1 PointerToArrayOfPIDs executeJOINPhase(JoinPhase *pointer) {
2   PointerToArrayOfPIDs pidArray =
3     malloc(sizeof(pid_t)*pointer->numberOfProcessesToJoin);
4   for (int i=0;i<pointer->numberOfProcessesToJoin; i++) {
5     pid_t pidToWait =
6       getLookupTable((*pointer->initialPhases)[i]);
7     waitpid(pidToWait,NULL,0);
8     (*pidArray)[i] = pidToWait;
9   }
10  return pidArray;
11 }

```

---

**Fasi JOIN**

Una fase JOIN causa l'attesa di processi generati in precedenza. La funzione `executeJOINPhase` è responsabile del suo funzionamento, ed è mostrata nel codice 18.

Similmente alla `executeFORKPhase` anche questa restituisce un puntatore ad un array di `pid`. Mentre per la prima funzione questo identificava i processi *creati*, in questa rappresenta i processi *aspettati*. Anche questa informazione verrà associata alla fase e memorizzata su disco dalla funzione `setForkJoinEndTimestamp` (linee 39 e 40 del codice 14).

La struttura `JoinPhase` contiene al suo interno `initialPhases`, ovvero il puntatore all'array delle *fasi iniziali* di ogni processo da aspettare. L'indirizzo della fase è convertito nel corrispondente `pid` tramite la `getLookupTable`, che rappresenta la funzione duale alla `setLookupTable` chiamata da una fase FORK nel codice 17 (linee 5-6).

Il `pid` può quindi essere utilizzato dalla funzione `waitpid` (si veda a pagina 47) che pone il processo corrente nello stato *blocked* fino al termine del processo `pid`. L'identificatore è quindi memorizzato in `pidArray` (linea 8) in attesa di essere restituito al chiamante (linea 10).

Il procedimento è ripetuto per ogni processo da attendere specificato (linea 4).

**3.5.2 Gestione delle informazioni operative**

Come già mostrato nel codice 14 il generatore di workload memorizza, per ogni fase ad eccezione delle LOOP, alcune informazioni relative alla loro esecuzione. In questa sezione si mostra come queste vengono collezionate e quindi scritte su disco.

Ogni fase di tipo I/O-bound, CPU-bound, FORK e JOIN ha una struttura di tipo `Statistics` associata, la cui composizione è mostrata nel codice 19.

**Codice 19:** Elementi della struttura `Statistics`.

---

```
1 typedef struct Statistics {
2     pid_t pid;
3     int id;
4     int initialCPU;
5     int finalCPU;
6     PhaseType type;
7     long int iterations;
8     struct timespec start;
9     struct timespec end;
10    PointerToArrayOfPIDs pids;
11    int numberOfPids;
12    struct Statistics *next;
13 } Statistics;
```

---

Intuitivamente, i campi `pid`, `id` e `type` specificano rispettivamente il *processo*, la *fase* e il suo tipo a cui la struttura si riferisce. `initialCPU` e `finalCPU` ospitano un numero che rappresenta rispettivamente la CPU che ha eseguito la fase al momento della sua creazione e della sua terminazione. Un calcolatore con  $n$  unità di calcolo identifica con numeri interi nell'intervallo  $[0, n - 1]$  le sue CPU. Le due variabili possono differire, ad esempio, se il processo viene tolto dall'esecuzione su decisione dello scheduler (si veda a pagina 45) mentre questo sta eseguendo la fase `id`, per poi venire ripreso in esecuzione da una CPU differente da quella iniziale.

Il campo `iterations` memorizza il numero di iterazioni specificato per una fase CPU-bound o I/O-bound, mentre `numberOfPids` e `pids` specificano il numero e i processi creati da una FORK o attesi da una JOIN. Dove non rilevanti a causa del tipo della fase, i campi non sono utilizzati.

Le strutture di tipo `timespec` (si veda il codice 7) chiamate `start` e `end` contengono l'*uptime* del sistema nei momenti di inizio e fine della fase, rispettivamente.

Anche le strutture `Statistics`, analogamente a quelle che descrivono il taskgraph, sono memorizzate sotto forma di *liste*. Il puntatore `next` rende possibile la formazione di queste.

All'inizio della procedura `simulate` viene creata la prima struttura di tipo `Statistics`, relativa alla prima fase del processo (linee 4 e 5). Ogni fase diversa dalla LOOP riempie le informazioni della propria struttura sia prima sia dopo la sua esecuzione, tramite le funzioni `setOrdinaryStartTimestamp` e `setOrdinaryEndTimestamp` se di tipo CPU-bound o I/O-bound, oppure tramite `setForkJoinStartTimestamp` e `setForkJoinEndTimestamp` se di tipo FORK o JOIN. Le due funzioni chiamate al termine delle fasi si preoccupano di allocare in memoria lo spazio necessario alla prossima struttura di tipo `Statistics` che conterrà le informazioni della prossima fase e avanza il

**Codice 20:** Scrittura dei file di log su disco.

---

```

1 void writeAndCheckErrors(int file, void * buffer, size_t size) {}
2
3 void writeLogFile(Statistics * pointer) {
4     Statistics * current = pointer;
5     file = getDescriptor(getpid());
6     while(current->next!=NULL) {
7         writeAndCheckErrors(file,&(current->pid),sizeof(pid_t));
8         writeAndCheckErrors(file,&(current->id),sizeof(int));
9         writeAndCheckErrors(file,&(current->initialCPU),sizeof(int));
10        writeAndCheckErrors(file,&(current->finalCPU),sizeof(int));
11        writeAndCheckErrors(file,&(current->type),sizeof(PhaseType));
12        writeAndCheckErrors(file,&(current->iterations),sizeof(long int));
13        writeAndCheckErrors(file,&((current->start).tv_sec),sizeof(time_t));
14        writeAndCheckErrors(file,&((current->start).tv_nsec),sizeof(long));
15        writeAndCheckErrors(file,&((current->end).tv_sec),sizeof(time_t));
16        writeAndCheckErrors(file,&((current->end).tv_nsec),sizeof(long));
17        writeAndCheckErrors(file,&(current->numberOfPids),sizeof(int));
18        for (int i=0; i<current->numberOfPids;i++)
19            writeAndCheckErrors(file,&((*current->pids)[i]),sizeof(pid_t));
20        current = current->next;
21    }
22 }

```

---

puntatore alla struttura corrente `currentStat`.

Al termine dell'esecuzione, il processo scrive su disco la propria lista di informazioni, tramite la funzione `writeLogFile`, costruendo così il proprio *file di log*.

Si noti che per costruzione ogni processo ha la *propria* lista di `Statistics`, pertanto verrà scritto su disco un file per ogni processo generato dall'applicazione.

## Memorizzazione delle informazioni

Le varie strutture `Statistics` sono riempite dalle quattro funzioni mostrate nel codice 21.

Le funzioni che vengono eseguite *prima* dell'esecuzione di una fase, ovvero `setOrdinaryStartTimestamp` e `setForkJoinStartTimestamp` memorizzano in `Statistics` tutte le informazione note anche senza l'esecuzione della stessa.

Queste includono il `pid` del processo, ottenibile dal sistema operativo tramite la funzione `getpid` (linee 5 e 16), l'`id` della fase associata (linee 6 e 17) e il suo tipo (linee 7 e 18). Tali informazioni sono passate esplicitamente dalla funzione `simulate`. Per le sole fasi CPU-bound e I/O-bound è prevista la memorizzazione del numero di iterazioni specificato (linea 8).

**Codice 21:** Funzioni di memorizzazione delle informazioni operative.

---

```

1  #define CLOCK_TYPE CLOCK_BOOTTIME
2
3  void setOrdinaryStartTimestamp(Statistics *stat, int id,
4      PhaseType type, long int iterations) {
5      stat->pid = getpid();
6      stat->id = id;
7      stat->type = type;
8      stat->iterations = iterations;
9      stat->numberOfPids = 0;
10     stat->initialCPU = sched_getcpu();
11     clock_gettime(CLOCK_TYPE,&(stat->start));
12 }
13
14 void setForkJoinStartTimestamp(Statistics *stat, int id,
15     PhaseType type) {
16     stat->pid = getpid();
17     stat->id = id;
18     stat->type = type;
19     stat->initialCPU = sched_getcpu();
20     clock_gettime(CLOCK_TYPE,&(stat->start));
21 }
22
23 Statistics* setOrdinaryEndTimestamp(Statistics *stat) {
24     clock_gettime(CLOCK_TYPE,&(stat->end));
25     stat->finalCPU = sched_getcpu();
26     stat->next = malloc(sizeof(Statistics));
27     return stat->next;
28 }
29
30 Statistics* setForkJoinEndTimestamp(Statistics *stat,
31     PointerToArrayOfPIDs pids) {
32     clock_gettime(CLOCK_TYPE,&(stat->end));
33     stat->finalCPU = sched_getcpu();
34     stat->pids = pids;
35     stat->numberOfPids = getNumber(pids);
36     stat->next = malloc(sizeof(Statistics));
37     return stat->next;
38 }

```

---

La CPU corrente è ottenibile grazie alla funzione `sched_getcpu`, anche questa messa a disposizione dal sistema operativo (linee 10 e 19), mentre la `clock_gettime` richiede l'uptime del sistema (linee 11 e 20; si veda a pagina 54). Si noti come la richiesta di queste due informazioni sia lasciata appositamente *al termine* delle due funzioni, per minimizzare i possibili fenomeni di cui si è parlato a pagina 55.

Al *termine* di una fase le due funzioni `setOrdinaryEndTimestamp` e `setForkJoinEndTimestamp` collezionano nuovamente uptime e CPU di esecuzione (linee 24, 25 e 32, 33). Questa volta le operazioni sono compiute il prima possibile, per gli stessi motivi di prima.

Nel caso di fasi FORK e JOIN i `pid` dei processi creati o attesi e il loro numero sono memorizzati nelle variabili corrispondenti (linee 34 e 35). Una nuova struttura di tipo `Statistics` è quindi creata in ogni caso (linee 26 e 36) e la lista è fatta avanzare (linee 27 e 37).

### Creazione dei file di log

Al termine dell'esecuzione di tutte le fasi previste per un processo la `simulate` chiama la funzione `writeLogFile`, responsabile della creazione del *file di log* del processo. La sua implementazione è mostrata nel codice 20 a pagina 91.

L'intera lista di strutture `Statistics` a partire da `pointer` è memorizzata ordinatamente nel file di log del processo, ottenuto nel codice dalla funzione `getDescriptor`. Ogni informazione è, dunque, scritta in tale file tramite la funzione `writeAndCheckErrors`, la cui implementazione non è mostrata, poiché ha un funzionamento intuitivo. La sua interfaccia è comunque visibile nella linea 1.

L'operazione continua fino alla fine della lista. La terminazione della funzione `writeLogFile` coincide con il termine del processo.

## 3.6 L'analizzatore di log

Il compito dell'*analizzatore di log* è leggere i file prodotti dal *generatore di workload* per poi realizzarne di nuovi che permettono la ricostruzione grafica o testuale delle informazioni collezionate per ogni fase.

Poiché i file di log sono composti da sequenze di strutture `Statistics`, l'analizzatore usa una funzione simile, ma opposta alla `writeLogFile` per leggere ognuna di queste. Ogni file viene analizzato e la lista originale di `Statistics` viene ricostruita in memoria.

A differenza del generatore di workload, in cui *ogni processo* ha la propria lista, in questo componente la lista mantenuta è *una sola*, contenente le strutture di *tutti* i processi. Durante l'inserimento le strutture sono ordinate per *uptime iniziale*, contenuto nel campo `start` (si veda il codice 19, linea 8). Da questa unica lista è possibile produrre tutti i file desiderati. A tutti

i tempi presenti nelle strutture viene sottratto l'uptime iniziale della prima fase, in modo che sia possibile creare un sistema di riferimento temporale che parta da zero. I tempi che ne conseguono, dunque, non sono più degli *uptime* a tutti gli effetti, in quanto non sono relativi al tempo di avvio del sistema ma a quello della *fase iniziale*.

Nel seguito si esaminano i vari *output* prodotti da questo componente e si descrive come è possibile realizzarli. Tutti gli esempi forniti sono relativi ad un'esecuzione del workload descritto dal taskgraph di esempio già mostrato nella figura 7.

Si tenga presente che i file prodotti dall'analizzatore *non sono* di per se comprensibili dall'utente, ma sono *nel giusto formato* per essere utilizzati da un programma esterno, in grado di realizzare grafici o tabelle, in modo che lo sforzo che un utilizzatore deve compiere per visualizzarli sia minimo.

Spesso questi programmi consentono di generare dei segmenti in un grafico tramite una sequenza di linee del tipo  $(x_{\text{iniziale}}, y_{\text{iniziale}}), (x_{\text{finale}}, y_{\text{finale}})$  letta da un file, con qualche variazione di formato. L'analizzatore di log si preoccupa di generare i file nel formato appropriato, in modo che *un certo* programma possa elaborarlo. L'onere di avviare il programma e di configurarlo nel modo corretto per la lettura dei file prodotti resta a carico dell'utente.

Sono state implementate le procedure per produrre dei file adeguati a due programmi di elaborazione grafica: *gnuplot*<sup>2</sup> e  $\text{\LaTeX}$ <sup>3</sup>. Poiché quest'ultimo è stato utilizzato per la redazione di questo documento, nel seguito si mostrano grafici e tabelle prodotte a partire dai file che l'analizzatore ha realizzato nel formato adatto a  $\text{\LaTeX}$ .

Non si mostrerà la *struttura* dei file prodotti in quanto altamente specifica ai programmi esterni e non rilevante ai fini della comprensione; si è preferito, invece, concentrarsi sugli *algoritmi* che consentono di ottenerli.

### 3.6.1 File di statistiche testuali

Il *file di statistiche testuali* contiene tutte le informazioni memorizzate in ogni struttura `Statistics` scritte in formato testuale, leggibile senza fatica da un analista umano anche senza ulteriori rielaborazioni. È ordinato per *tempo di inizio*.

La lettura di questo file è opportuna se si desidera analizzare nel dettaglio il traffico generato nel sistema, in quanto consente di capire con precisione gli istanti di inizio e fine di ogni fase, la loro durata o la durata media di ogni iterazione tramite delle semplici elaborazioni delle metriche a disposizione.

Se ne presenta un esempio nella tabella 5, il cui contenuto è stato elaborato dal *file di statistiche testuali* prodotto dall'analizzatore di log in un formato adatto alla visualizzazione in forma tabellare tramite  $\text{\LaTeX}$ . A causa dei

---

<sup>2</sup><http://www.gnuplot.info/>

<sup>3</sup><http://www.latex-project.org/>



limiti di spazio, le informazioni riportate sono solo una parte di quanto prodotto dall'analizzatore.

La costruzione del file è molto facile, consistendo nella semplice lettura dell'intera lista di `Statistics`, già ordinata, accompagnata dalla memorizzazione su disco delle informazioni rilevanti.

**Tabella 5:** Esempio di un file di statistiche testuali. Il contenuto è stato rielaborato da  $\text{\LaTeX}$  per poter essere visualizzato tramite una tabella, ed include solo parte delle informazioni a disposizione per limiti di spazio. P.I., P.F. e ITER. sono abbreviazioni di “Processore iniziale”, “Processore finale” e “Numero di iterazioni” rispettivamente. La colonna NOTE riporta, dove rilevante, i PID dei processi creati o aspettati dalla fase. I dati sono relativi ad un'esecuzione del *generatore di workload* derivato dal taskgraph descritto dalla figura 7.

PID	ID	TIPO	P.I.	P.F.	ITER.	INIZIO [ms]	FINE [ms]	NOTE
12153	101	I/O	2	2	10	0.000000	30.496086	-
12153	102	CPU	2	2	20	30.497314	30.904720	-
12153	103	CPU	2	2	40	30.905677	51.284912	-
12153	101	I/O	2	2	10	51.293137	51.704863	-
12153	102	CPU	2	2	20	51.705406	63.317358	-
12153	103	CPU	2	2	40	63.323090	81.568403	-
12153	101	I/O	2	2	10	81.576300	81.959618	-
12153	102	CPU	2	2	20	81.960185	112.525394	-
12153	103	CPU	2	2	40	112.527056	142.294900	-
12153	101	I/O	2	2	10	142.302965	142.674879	-
12153	102	CPU	2	2	20	142.675503	158.056883	-
12153	103	CPU	2	2	40	158.058483	177.179681	-
12153	101	I/O	2	2	10	177.187683	177.572655	-
12153	102	CPU	2	2	20	177.573342	189.184100	-
12153	103	CPU	2	2	40	189.185522	208.392989	-
12153	101	I/O	2	2	10	208.400876	208.777590	-
12153	102	CPU	2	2	20	208.778283	220.366719	-
12153	103	CPU	2	2	40	220.368045	239.300140	-
12153	101	I/O	2	2	10	239.308354	251.238391	-
12153	102	CPU	2	2	20	251.239158	269.942118	-
12153	103	CPU	2	2	40	269.943575	282.157457	-
12153	101	I/O	2	2	10	282.161457	300.564088	-
12153	102	CPU	2	2	20	300.565437	300.941198	-
12153	103	CPU	2	2	40	300.941927	330.726731	-
12153	101	I/O	2	2	10	330.734841	344.169753	-
12153	102	CPU	2	2	20	344.170573	362.526365	-
12153	103	CPU	2	2	40	362.528081	375.127922	-
12153	101	I/O	2	2	10	375.131949	393.387178	-

*Continua nella prossima pagina*

*Continua dalla pagina precedente*

PID	ID	TIPO	P.I.	P.F.	ITER.	INIZIO [ms]	FINE [ms]	NOTE
12153	102	CPU	2	2	20	393.388335	393.744111	-
12153	103	CPU	2	2	40	393.744765	423.819505	-
12153	101	I/O	2	2	10	423.827441	437.112922	-
12153	102	CPU	2	2	20	437.113888	456.174298	-
12153	103	CPU	2	2	40	456.175674	468.203721	-
12153	105	I/O	2	2	10	468.211126	468.553671	-
12153	105	I/O	2	2	10	468.560149	487.195876	-
12153	105	I/O	2	2	10	487.203412	487.541125	-
12153	105	I/O	2	2	10	487.547222	487.884450	-
12153	105	I/O	2	2	10	487.890623	499.257974	-
12153	105	I/O	2	2	10	499.261542	518.051194	-
12153	107	FORK	2	2	-	518.058151	518.208420	Creati: 12162 12163
12162	201	CPU	3	3	20	518.172456	518.581385	-
12162	201	CPU	3	3	20	518.589877	530.304857	-
12163	301	I/O	4	4	30	520.196063	612.850745	-
12162	201	CPU	3	3	20	530.308700	548.264967	-
12162	201	CPU	3	3	20	548.268937	561.220573	-
12162	201	CPU	3	3	20	561.224274	579.131166	-
12162	201	CPU	3	3	20	579.136134	593.265672	-
12162	201	CPU	3	3	20	593.282427	612.947811	-
12163	302	FORK	4	4	-	612.852085	612.991831	Creati: 12164 12165 12166
12162	201	CPU	3	3	20	612.953555	625.258982	-
12165	501	I/O	7	7	30	612.960585	705.451227	-
12163	303	JOIN	4	4	-	612.992600	737.529721	Aspettati: 12164 12165
12164	401	CPU	6	6	10	613.166499	645.509890	-
12166	601	CPU	0	0	50	613.200221	705.518768	-
12162	201	CPU	3	3	20	625.266360	687.158293	-
12164	402	I/O	6	6	20	645.511278	737.070724	-
12162	201	CPU	3	3	20	687.164040	705.781677	-
12162	201	CPU	3	3	20	705.785863	737.652115	-
12163	304	I/O	4	4	10	737.530764	749.112870	-
12162	201	CPU	3	3	20	737.657909	749.244149	-
12163	305	JOIN	4	4	-	749.113989	749.166390	Aspettato: 12166
12163	306	I/O	4	4	40	749.166824	861.572504	-
12162	201	CPU	3	3	20	749.249984	768.353713	-
12162	201	CPU	3	3	20	768.359872	780.273700	-
12162	201	CPU	3	3	20	780.278247	829.694191	-
12162	201	CPU	3	3	20	829.699091	829.917586	-
12162	201	CPU	3	3	20	829.920968	861.160818	-
12162	201	CPU	3	3	20	861.171155	872.204885	-

*Continua nella prossima pagina*

*Continua dalla pagina precedente*

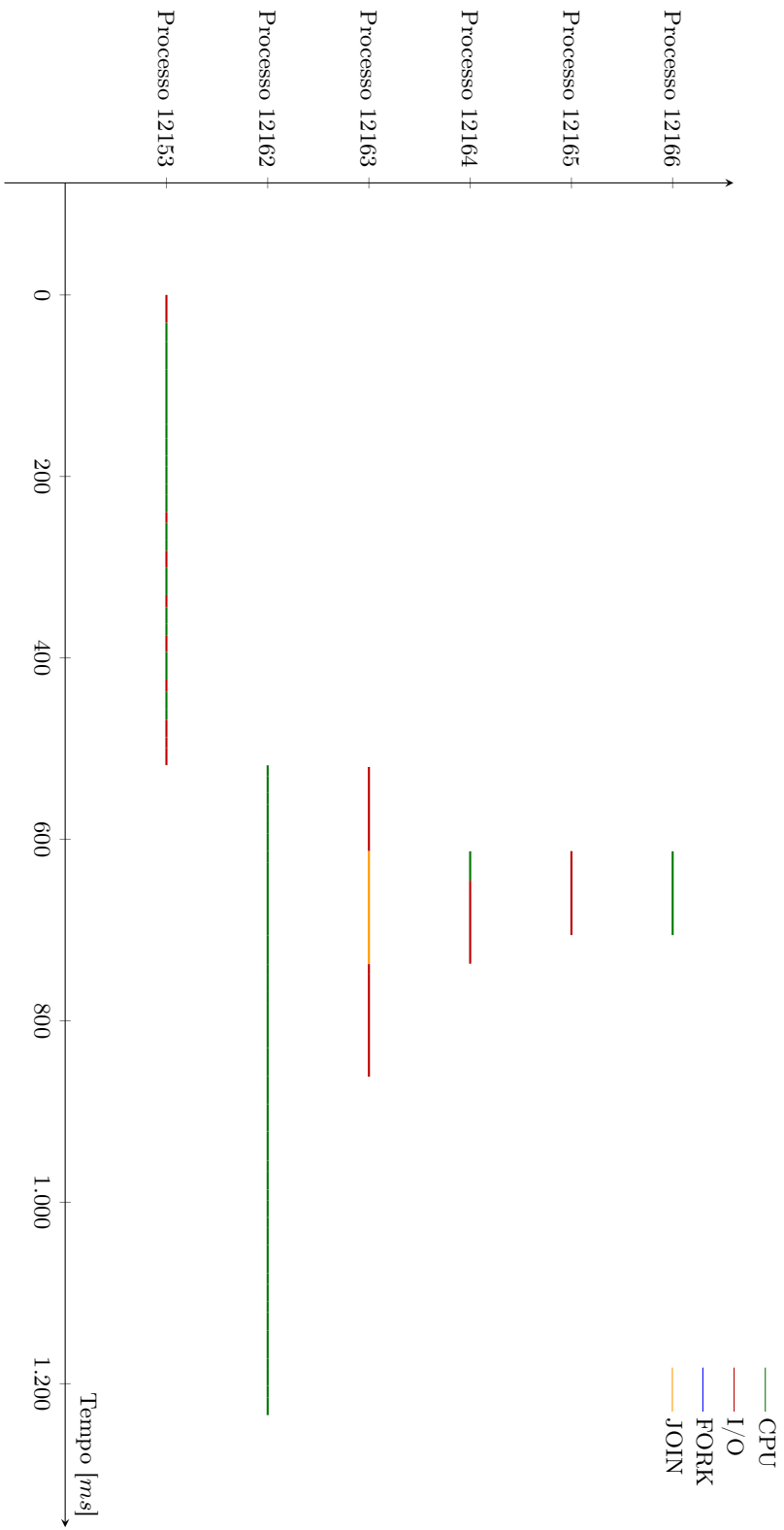
PID	ID	TIPO	P.I.	P.F.	ITER.	INIZIO [ms]	FINE [ms]	NOTE
12162	201	CPU	3	3	20	872.210109	891.376912	-
12162	201	CPU	3	3	20	891.387239	891.859066	-
12162	201	CPU	3	3	20	891.867443	921.725645	-
12162	201	CPU	3	3	20	921.735779	922.235929	-
12162	201	CPU	3	3	20	922.244520	935.386951	-
12162	201	CPU	3	3	20	935.396509	953.914235	-
12162	201	CPU	3	3	20	953.924229	954.425753	-
12162	201	CPU	3	3	20	954.434379	966.180838	-
12162	201	CPU	3	3	20	966.185292	985.925051	-
12162	201	CPU	3	3	20	985.951549	986.430754	-
12162	201	CPU	3	3	20	986.439290	997.720154	-
12162	201	CPU	3	3	20	997.729570	1016.389424	-
12162	201	CPU	3	3	20	1016.399334	1016.881641	-
12162	201	CPU	3	3	20	1016.889505	1028.221086	-
12162	201	CPU	3	3	20	1028.225751	1047.326360	-
12162	201	CPU	3	3	20	1047.336586	1077.855959	-
12162	201	CPU	3	3	20	1077.866076	1078.365587	-
12162	201	CPU	3	3	20	1078.373959	1090.116061	-
12162	201	CPU	3	3	20	1090.125831	1090.593406	-
12162	201	CPU	3	3	20	1090.601304	1109.353799	-
12162	201	CPU	3	3	20	1109.363814	1121.099593	-
12162	201	CPU	3	3	20	1121.109658	1121.589052	-
12162	201	CPU	3	3	20	1121.597057	1141.343499	-
12162	201	CPU	3	3	20	1141.353155	1141.815796	-
12162	201	CPU	3	3	20	1141.823946	1152.518785	-
12162	201	CPU	3	3	20	1152.528121	1171.885209	-
12162	201	CPU	3	3	20	1171.895578	1172.385102	-
12162	201	CPU	3	3	20	1172.393677	1183.475369	-
12162	201	CPU	3	3	20	1183.484839	1202.216590	-
12162	201	CPU	3	3	20	1202.226572	1202.698811	-
12162	201	CPU	3	3	20	1202.706899	1215.004708	-
12162	201	CPU	3	3	20	1215.014168	1234.171639	-

*Si conclude dalla pagina precedente*

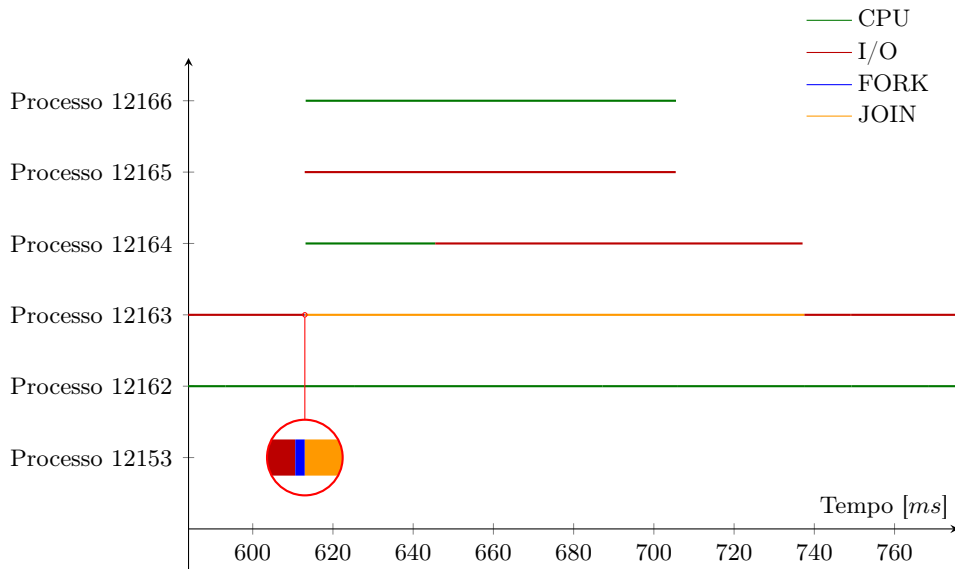
### 3.6.2 Profilo temporale dell'esecuzione

Il *profilo temporale dell'esecuzione* consente di realizzare un grafico in cui vengono mostrate le fasi eseguite da ogni processo nel tempo.

Ogni valore discreto riportato sull'asse delle ordinate corrisponde ad un processo, mentre il tempo scorre sull'asse delle ascisse. Esempi del grafico sono le figure 8 e 9. In particolare la seconda è un dettaglio della prima, in



**Figura 8:** Esempio di un *profilo temporale dell'esecuzione*. La figura riporta in forma grafica l'andamento delle fasi contenuto nella tabella 5. La legenda indica la corrispondenza di ogni colore al relativo tipo di fase. Sull'ordinata sono riportati i PID dei processi come dall'omonima colonna della tabella precedente.



**Figura 9:** Dettaglio del profilo temporale mostrato nella figura 8. Il grafico mostra in forma allargata il comportamento dei processi nell'intervallo temporale  $[600, 760]$ , evidenziando inoltre la presenza di una fase FORK, eseguita nell'intervallo  $[612.852085, 612.991831]$  dal processo 12163.

cui si evidenzia il comportamento mostrato dall'esecuzione in un intervallo temporale più ristretto. Entrambe le figure riportano gli stessi dati contenuti nella tabella 5.

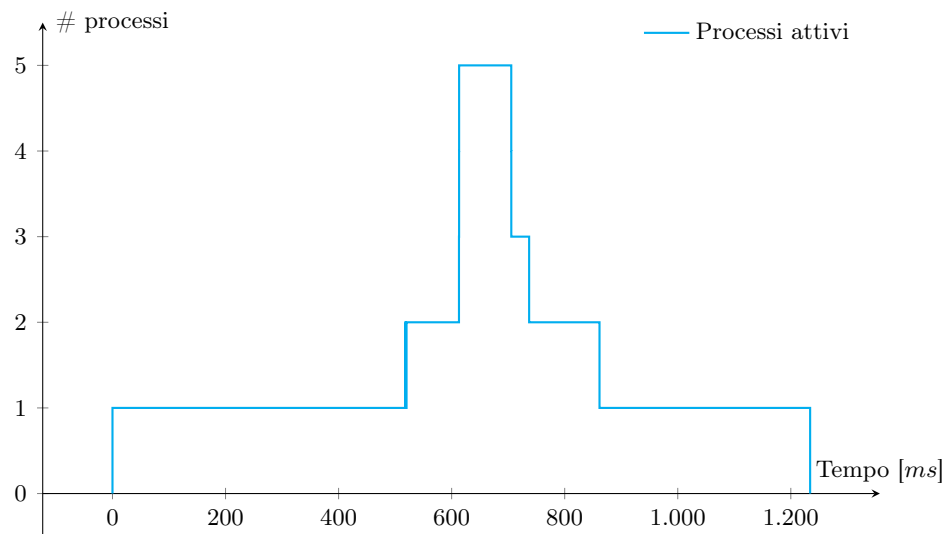
La realizzazione di questo grafico è simile a quanto necessario per produrre il file di statistiche testuali, in quanto, per comporre i segmenti che rappresentano la vita di una fase, è sufficiente leggere l'intera lista di `Statistics` e, per ogni struttura, estrarre il contenuto di `start` ed `end` per poi memorizzarli su disco nel formato adeguato per il programma di elaborazione grafica esterna (L<sup>A</sup>T<sub>E</sub>X in questo caso).

### 3.6.3 Profilo temporale dei processi

Il *profilo temporale dei processi* consente di visualizzare in forma grafica il *numero* dei processi in esecuzione nel tempo. Un esempio ne è la figura 10.

Il grafico è simile ad una funzione *a gradino* che cambia il proprio valore in corrispondenza della creazione o della terminazione di un processo. Sebbene l'ordinata possa assumere esclusivamente valori interi, per esigenze grafiche si riportata anche la transizione verticale.

Per produrre questo file è necessario individuare gli intervalli temporali nel quale il numero di processi resta costante. Ad esempio, se nell'intervallo temporale  $[0, 200]$  si ha una FORK di un processo all'istante 80, una JOIN



**Figura 10:** Esempio di un *profilo temporale dei processi*. Il grafico è stato elaborato dalle informazioni contenute nella tabella 5.

all'istante 120 e all'inizio dell'intervallo vi era un solo processo in esecuzione, gli intervalli da individuare sono  $[0, 80]$ ,  $[80, 120]$  e  $[120, 200]$ , composti rispettivamente da un numero di processi attivi pari a 1, 2 e 1.

Tali intervalli devono quindi essere trasformati in segmenti grafici, situati ad un valore di ordinata uguale al numero di processi attivi in essi. Ad esempio, i punti  $(80, 2)$  e  $(120, 2)$  descrivono il secondo degli intervalli precedenti, e permettono la visualizzazione del segmento corrispondente.

L'analizzatore ricava intervalli e segmenti nel modo seguente. La lista di `Statistics` viene letta interamente in modo da individuare l'*istante di inizio* e l'*istante di fine* di ogni processo, contenuti rispettivamente nel campo `start` della prima fase e nel campo `end` dell'ultima fase di ognuno di essi.

Le informazioni ottenute vengono memorizzate in due array distinti, uno dedicato agli istanti di inizio e l'altro agli istanti di fine. Questi sono quindi ordinati separatamente.

Il codice 23 mostra come il benchmark proceda alla creazione degli intervalli. L'obiettivo è la creazione di una lista composta da strutture del tipo `ProcessInterval`, la cui composizione è mostrata nel codice 22.

La struttura è in grado di descrivere l'intervallo tramite i due istanti temporali `start` e `end`, di tipo `double` per facilitare il loro confronto. Il campo `nProcesses` contiene il numero costante di processi mostrati dall'esecuzione durante l'intervallo. Il puntatore `next` rende possibile il concatenamento delle strutture per formare una lista.

Nell'algoritmo si suppone di disporre degli istanti iniziali e finali negli array `startingTimes` ed `endingTimes`, già ordinati (linee 2-3). La variabile

**Codice 22:** Elementi della struttura `ProcessInterval`.

---

```

1 typedef struct ProcessInterval {
2     double start;
3     double end;
4     int nProcesses;
5     struct ProcessInterval *next;
6 } ProcessInterval;

```

---

`totalProcesses` contiene il numero di processi *totali* creati dal generatore di traffico durante l'esecuzione del workload (linea 1).

La lista da creare sarà ordinata per *istanti di inizio*, pertanto il primo intervallo ha per definizione il campo `start` pari al primo istante di inizio in assoluto (linea 8) e `nProcesses` pari a 1, in quanto il generatore di traffico è inizialmente composto da un singolo processo (linea 9).

L'algoritmo deve ora determinare se il *prossimo istante* (considerando *globalmente* i due array) è di inizio o di fine. Nel primo caso (linee 15-17) un nuovo processo è stato creato nell'istante contenuto in `startingTimes[startIndex]`, pertanto tale valore coincide con la fine dell'intervallo corrente (linea 18) e con l'inizio del successivo (linea 23), che avrà un numero di processi pari alla quantità precedente, aumentata di un'unità (linee 19 e 22).

Il secondo caso è simile al primo, con la differenza che il numero di processi del nuovo intervallo sarà diminuito anziché aumentato, in quanto l'occorrenza segna la terminazione di un processo (linea 34).

Il resto dell'algoritmo non dovrebbe essere di difficile comprensione. Si noti che un'implementazione più intuitiva e con meno controlli avrebbe potuto fondere i due array discriminando tramite un booleano il "tipo di istante" considerato (ovvero se di inizio o di fine). L'uso di un singolo array ordinato semplifica i controlli in quanto può essere utilizzato un solo indice.

La procedura è ripetuta fino all'esaurimento degli istanti temporali in entrambi gli array. Al termine dell'algoritmo, la lista di `ProcessInterval` può essere memorizzata su un file rispettando il formato appropriato al programma di elaborazione grafica prescelto.

### 3.6.4 Profilo temporale delle fasi

L'ultimo tipo di file prodotto dall'analizzatore di log è il *profilo temporale delle fasi*. La sua composizione è molto simile al tipo di grafico precedente, con la differenza che questo mostra l'andamento delle *singole fasi* nel tempo anziché il numero dei processi.

L'analizzatore produce un file separato *per ogni tipo di fase*, in modo che questi possano essere studiati separatamente (figure 11, e 12) o contemporaneamente (figura 13).

**Codice 23:** Algoritmo di creazione degli intervalli temporali.

---

```

1 // int totalProcesses;
2 // double startingTimes[totalProcesses];
3 // double endingTimes[totalProcesses];
4
5 int startIndex=0, endIndex=0;
6
7 ProcessInterval *processInterval = malloc(sizeof(ProcessInterval));
8 processInterval->start = startingTimes[startIndex];
9 processInterval->nProcesses = 1;
10 startIndex++;
11
12 int prevProcess;
13
14 while(startIndex < totalProcesses || endIndex < totalProcesses) {
15     if ((startIndex < totalProcesses &&
16         startingTimes[startIndex] < endingTimes[endIndex]) ||
17         endIndex >= totalProcesses) {
18         processInterval->end = startingTimes[startIndex];
19         prevProcess = processInterval->nProcesses;
20         processInterval->next = malloc(sizeof(ProcessInterval));
21         processInterval = processInterval->next;
22         processInterval->nProcesses = prevProcess + 1;
23         processInterval->start = startingTimes[startIndex];
24         startIndex++;
25     }
26     else
27     if ((endIndex < totalProcesses &&
28         startingTimes[startIndex] > endingTimes[endIndex]) ||
29         startIndex >= totalProcesses) {
30         processInterval->end = endingTimes[endIndex];
31         prevProcess = processInterval->nProcesses;
32         processInterval->next = malloc(sizeof(ProcessInterval));
33         processInterval = processInterval->next;
34         processInterval->nProcesses = prevProcess - 1;
35         processInterval->start = endingTimes[endIndex];
36         endIndex++;
37     }
38     else {
39         startIndex++;
40         endIndex++;
41     }
42 }

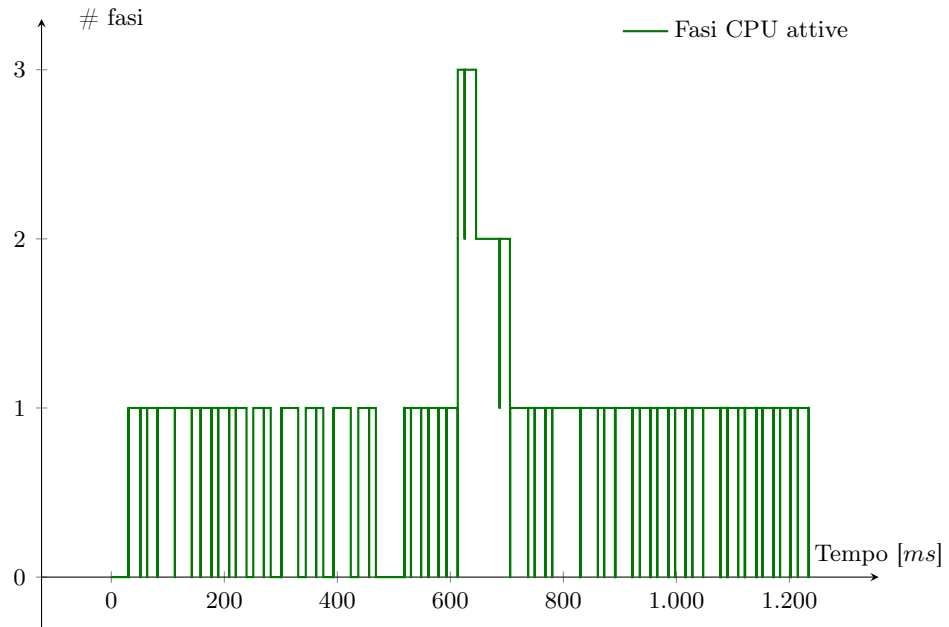
```

---

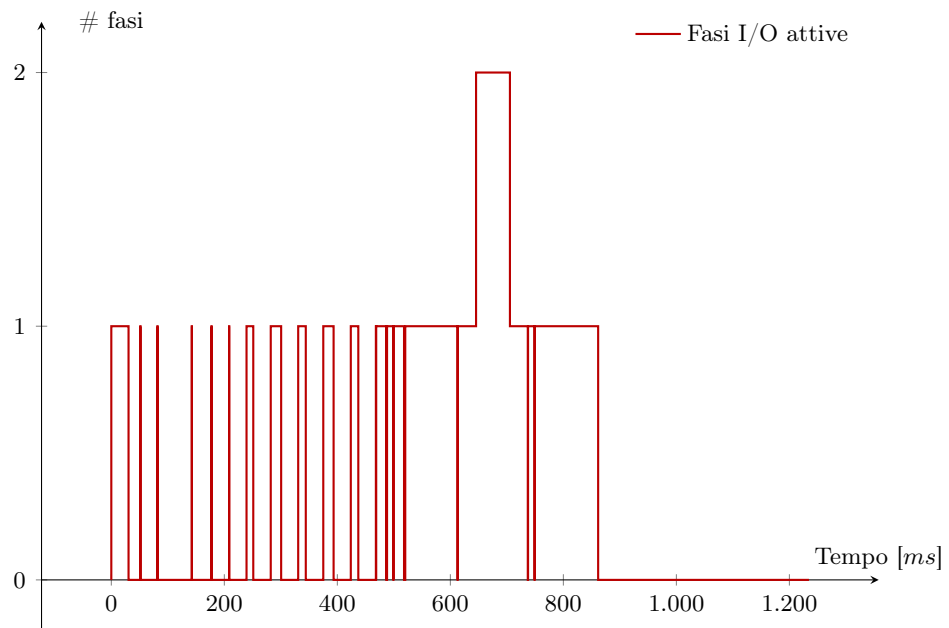


Anche in questo caso è necessario individuare gli istanti temporali in cui il numero di fasi attive di un certo tipo rimane costante. Il procedimento è analogo a quello utilizzato per il profilo temporale dei processi, con la differenza che la lista di **Statistics** contiene già gli istanti di inizio e di fine necessari, senza ulteriori rielaborazioni. È dunque possibile scansionare la lista e memorizzare gli istanti *del solo tipo di fase desiderato* in due array, per poi ripetere la procedura descritta in precedenza.

La struttura **ProcessInterval** può essere utilizzata per questo scopo se si ipotizza che nel campo **nProcesses** vi sia il numero di *fasi* attive anziché il numero dei processi.

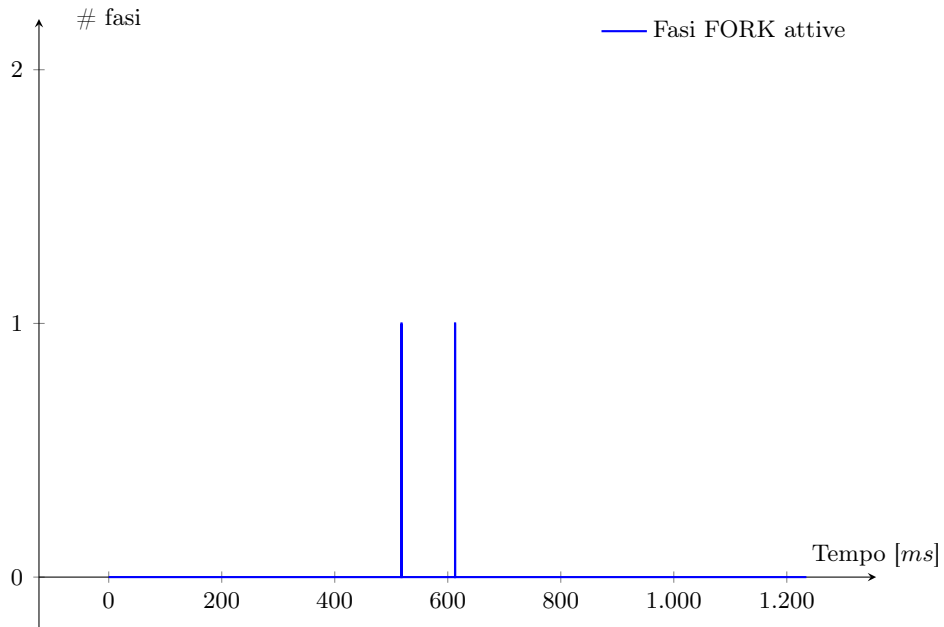


(a) Fasi CPU.

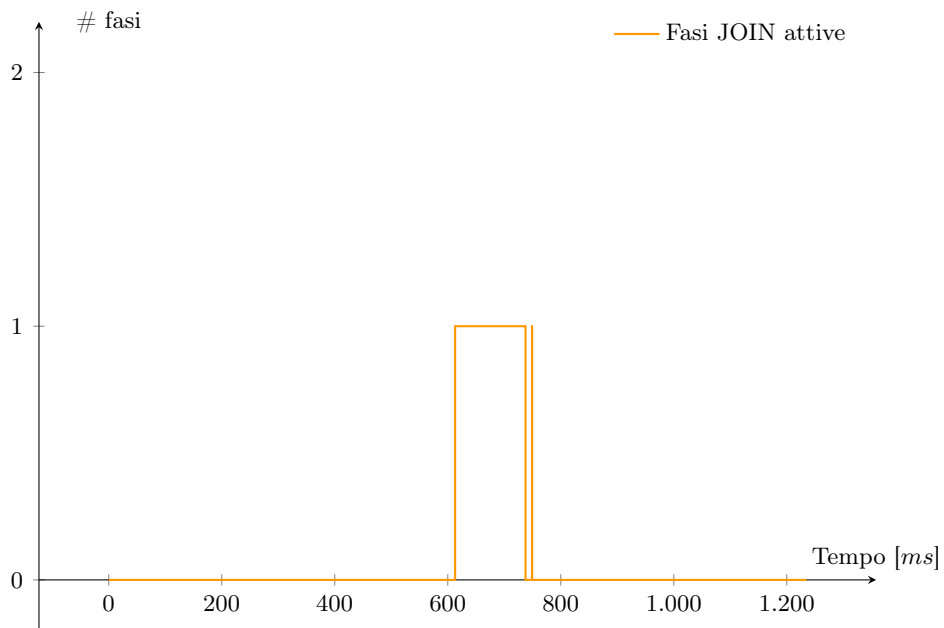


(b) Fasi I/O.

**Figura 11:** Esempi di *profili temporali delle fasi* di tipo CPU e I/O. I grafici sono stati elaborati dalle informazioni contenute nella tabella 5.

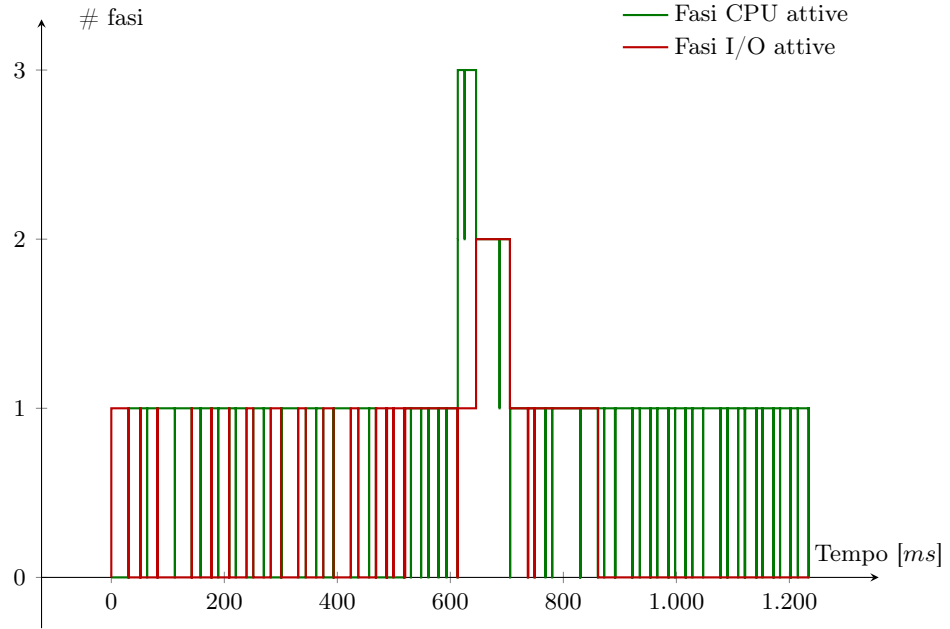


(a) Fasi FORK.

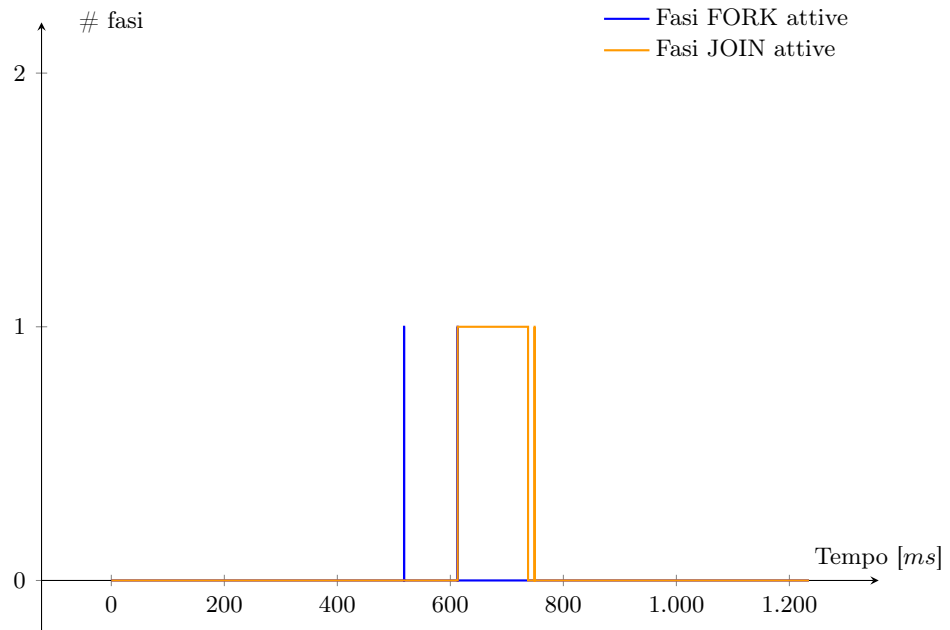


(b) Fasi JOIN.

**Figura 12:** Esempi di *profili temporali delle fasi* di tipo FORK e JOIN. I grafici sono stati elaborati dalle informazioni contenute nella tabella 5.



(a) Fasi IO e CPU.



(b) Fasi FORK e JOIN.

**Figura 13:** Esempi di *profili temporali delle fasi* di ogni tipo.

## Capitolo 4

### Validazione sperimentale

In questo capitolo si mostra il corretto funzionamento del benchmark sviluppato nel lavoro di tesi.

Un possibile approccio in questa direzione consiste nel considerare un algoritmo ben conosciuto e dal workload facilmente ricavabile, implementarlo aggiungendo le stesse funzionalità di raccolta e analisi delle informazioni temporali utilizzate nel benchmark, quindi costruire un taskgraph configurato in modo da consentirne una fedele riproduzione, per poi confrontare i due risultati ottenuti. La loro somiglianza costituirebbe una prova dell'effettiva efficacia dello strumento prodotto.

Nonostante questo tipo di validazione sia sicuramente più facile da realizzare, in questo lavoro si è preferito procedere diversamente, studiando e riproducendo un'applicazione MapReduce utilizzando Apache Hadoop (si veda la sezione 2.2) liberamente disponibile in rete e di facile utilizzo. Si procederà con l'analisi semplificata del suo workload, per poi riprodurlo tramite il benchmark realizzato.

La scelta è motivata dalla crescente importanza del framework e dalla mancanza di studi analoghi. L'autore ritiene che questo procedimento costituisca un'utile analisi di uno strumento dal funzionamento complesso e che probabilmente verrà sempre più utilizzato in futuro. In questo scenario è importante caratterizzare almeno nei tratti principali il traffico generato da Hadoop, in modo da poterne tenere conto in caso si voglia ottimizzare il carico di un'applicazione MapReduce.

In ogni caso non si dimentichi che lo scopo del capitolo è mostrare che un workload noto è riproducibile correttamente dal benchmark. La complessità di Hadoop ha reso necessaria l'introduzione di alcune semplificazioni che consentono di rendere più agile la derivazione del carico di lavoro e la sua riproduzione. Queste limitano la completezza dell'analisi, pur cogliendo fedelmente le operazioni più importanti e il modo in cui queste avvengono.

Nelle sezioni seguenti si descrivono le ipotesi e le semplificazioni effettuate nello studio dell'applicazione, in che modo sia stato possibile ricavarne un

workload e un profilo di esecuzione e che caratteristiche presenta il taskgraph che la riproduce per mezzo del benchmark.

#### 4.1 Il programma *pi*

Il programma studiato e riprodotto ai fini di validare lo strumento implementato è chiamato *pi*, ed è un'applicazione MapReduce inclusa nella distribuzione di Hadoop<sup>1</sup> con finalità di esempio e di test del sistema. Il lettore non dovrebbe incontrare difficoltà nell'identificarla all'interno del *bundle* standard disponibile sul sito di Apache, e i suoi codici sorgenti sono liberamente consultabili<sup>2</sup>.

L'applicazione *pi* implementa un algoritmo di approssimazione del numero  $\pi$  tramite un metodo *Monte Carlo*<sup>3</sup>, basato sull'intuizione che si descrive di seguito.

Poiché l'area di un quadrato di lato  $l$  è pari a

$$A_q = l^2$$

mentre quella di un cerchio di raggio  $r$  è pari a

$$A_c = \pi r^2$$

se il cerchio è iscritto al quadrato valgono le relazioni

$$r = \frac{1}{2}l \quad A_c = \frac{1}{4}\pi l^2$$

da cui

$$\pi = \frac{4A_c}{l^2} = \frac{4A_c}{A_q}.$$

Per approssimare i due valori  $A_c$  e  $A_q$  è possibile generare pseudo-casualmente dei numeri all'interno del quadrato, supposto per semplicità di lato unitario, quindi verificare quanti di questi sono anche interni al cerchio. Al crescere del numero dei punti si avrà che

$$\frac{\text{punti interni al cerchio}}{\text{punti totali}} \approx \frac{A_c}{A_q}.$$

È quindi possibile calcolare un valore *approssimato di*  $\pi$  tramite la quantità

$$\pi \approx 4 \cdot \frac{\text{punti interni al cerchio}}{\text{punti totali}}. \quad (4.1)$$

---

<sup>1</sup>La validazione è stata eseguita sulla versione 1.1.2 di Apache Hadoop, l'ultima release stabile al momento dello studio, che si garantisce contenere il programma.

<sup>2</sup>I codici sorgenti possono essere consultati all'indirizzo <http://grepcode.com/file/repository.cloudera.com/content/repositories/releases/com.cloudera.hadoop/hadoop-examples/0.20.2-320/org/apache/hadoop/examples/PiEstimator.java>.

<sup>3</sup>Un metodo *Monte Carlo* utilizza approcci pseudo-casuali per ottenere il proprio risultato, ammettendo che questo *possa non coincidere* con l'esatto valore desiderato.

Al crescere del numero di punti generati l'esattezza del valore calcolato migliora, in quanto questi riescono a "riempire" sempre più spazio all'interno delle due forme, approssimando più fedelmente la loro area.

In ambiente MapReduce, *pi* implementa il ragionamento precedente in questo modo:

1. Il *client*, prima di chiedere l'inizio del *job* al *JobTracker*, crea un file per ogni *mapper* nell'HDFS specificando al loro interno il *seed iniziale* per il generatore di numeri pseudo-casuali, e il *numero di punti* da generare per ognuno di essi, deciso dall'utente. Tutti i *mapper* generano lo stesso numero di punti.
2. Dopo la richiesta da parte del *client*, il *JobTracker* inizializza tanti *mapper* quanti specificati dall'utente e *un singolo reducer*.
3. Ogni *mapper*:
  - (a) Legge il proprio file di input dall'HDFS e inizializza il generatore di numeri pseudo-casuali.
  - (b) Per ogni punto da generare richiede la creazione di *due numeri pseudo-casuali* nell'intervallo  $[0, 1]$ , rappresentanti il punto all'interno del quadrato, quindi controlla se questo è interno o esterno al cerchio. In entrambi i casi incrementa un contatore dedicato.
  - (c) Al termine della generazione, scrive sull'HDFS due coppie del tipo  $(numInterni, valore)$  e  $(numEsterni, valore)$ , rappresentanti l'output dei *mapper*.
4. Il singolo *reducer*:
  - (a) Legge i vari file di input, contenenti le due coppie scritte dal *mapper* corrispondente.
  - (b) Aggrega i risultati da questi forniti e scrive i valori finali in due coppie come le precedenti.
5. L'applicazione può dunque leggere l'output del *reducer*, per poi procedere al calcolo approssimato di  $\pi$  utilizzando la formula (4.1).

Il comportamento dell'algoritmo è stato estratto direttamente dal codice sorgente disponibile online.

Come si può notare, *pi* non è un buon candidato per l'utilizzo del paradigma MapReduce, in quanto *non* opera direttamente su *dati* scritti sull'HDFS e sarebbe molto più proficuo implementarlo su un singolo calcolatore anziché su cloud. La lettura di file separati avviene soltanto per permettere la creazione del numero di *mapper* desiderato; infatti, il programma crea appositamente un blocco per ognuno di essi.

Tuttavia la possibilità di generare il numero di mapper voluto, la presenza di un singolo reducer e la semplicità delle sue operazioni lo rendono adeguato allo scopo di analisi, considerato che la struttura di un job MapReduce è identica a prescindere dall'applicazione.

L'uguaglianza del numero di punti da generare per ogni mapper semplifica, inoltre, la ricostruzione del loro workload.

## 4.2 Ipotesi e semplificazioni

Le applicazioni operanti su Apache Hadoop presentano un workload complesso data la intrinseca natura del framework, strutturato in diversi componenti a sé stanti in continua comunicazione tra loro.

Lo scopo della validazione *non* consiste nello studiare e nel tentare di replicare *ogni* operazione di Hadoop nel sistema, bensì di individuarne e riprodurne solamente le principali. Questa sezione descrive le semplificazioni e le ipotesi che sono state fatte per facilitare l'analisi, avendo comunque cura di non alterare la validità del procedimento.

### 4.2.1 Ipotesi generali

Un'applicazione MapReduce è solita operare su *sistemi distribuiti*, costituiti da più calcolatori eterogenei; al contrario, il benchmark realizzato si concentra su *sistemi paralleli*.

Per la validazione, Hadoop è stato installato su una *singola macchina*, che quindi comprende *tutti* i vari componenti del framework (*JobTracker*, *TaskTracker*, *NameNode* e *DataNode*). Una tale configurazione prende il nome di *stand-alone* ed è generalmente utilizzata con finalità di test.

La convivenza di tutti i componenti in un singolo calcolatore comporta la presenza di *più processi* diversi in continua interazione tra loro. Più esattamente, in sistemi distribuiti i vari componenti comporterebbero la creazione di *demoni*<sup>4</sup> in continua comunicazione tramite rete. In configurazioni *stand-alone* le richieste non lasciano la macchina, ma i processi continuano ad essere demoni a se stanti, la cui vita ed esecuzione sono indipendenti tra loro.

Modellare questo workload comporterebbe l'introduzione di più processi in parallelo (uno per ogni componente), composti in maniera preponderante da *continue attese* di messaggi dalla rete, da operatività limitata (ad esempio, solo il *DataNode* potrà compiere operazioni I/O-bound, solo il *TaskTracker* potrà compiere delle FORK) e sempre attivi (non esistono infatti vere e proprie operazioni di JOIN, piuttosto *esplicite segnalazioni* della fine di un task).

---

<sup>4</sup>Con il termine *demone* si definisce un processo in esecuzione nel sistema *senza* interazione attiva con l'utente, che risponde solo a richieste o messaggi a lui esplicitamente indirizzati, ad esempio tramite connessioni di rete.



Si è quindi preferito ricreare la *visione logica* del workload da parte del *client*, immaginando che questo costituisca il processo principale dell'applicazione. Sarà il client a creare e attendere i vari *task* che nella realtà sarebbero gestiti dal *JobTracker*, e ogni altro processo ingloberà le funzionalità di *TaskTracker*, *NameNode* o *DataNode*, eseguendo operazioni di I/O o computazionali in base alle necessità.

Questa visione, oltre a semplificare la derivazione e la riproduzione del traffico, fornisce anche un flusso logico delle operazioni più intuitivo.

Il framework ha inoltre una natura intrinsecamente complessa. Operazioni *non* controllabili sono portate a termine senza possibilità di previsione (come la replicazione dei blocchi) e numerose scritture e letture di file ausiliari accompagnano mapper e reducer costantemente. Benché sia possibile tracciarle e replicarle, si è preferito tralasciare tutto ciò che non sia strettamente riconducibile a delle normali richieste di tipo MapReduce, in quanto si rischia di creare confusione e complicare il taskgraph risultante.

#### 4.2.2 Semplificazione dei tempi

Come sarà illustrato più avanti, l'analisi del workload è possibile grazie ai *log* che i componenti di Hadoop utilizzano per rendicontare all'amministratore quanto avvenuto durante la loro esecuzione.

Sebbene utili, i log forniscono talvolta delle informazioni soltanto parziali, soprattutto riguardo agli istanti di inizio e fine delle operazioni. Ad esempio, di solito viene riportato il *successo* di una certa operazione, specificando *solo a volte* la sua durata. In altri casi è difficile capire se la stringa riportata nel file si riferisce ad un *comando di inizio* o ad una *rendicontazione finale*.

Laddove possibile si sono utilizzati altri strumenti per "incrociare" i risultati al fine di garantire una riproduzione fedele all'originale, ma a volte ciò non è stato praticabile, ad esempio per la durata delle operazioni di FORK. In questi casi è stato necessario effettuare delle stime temporali, guidate sia dal buonsenso, sia da operazioni simili svolte in precedenza, per definire delle durate sensate. In altre occasioni si è assunto che l'azione *successiva* riportata nel log segnasse la fine dell'operazione precedente, sempre ammesso che questa ipotesi non fosse in contrasto con altri dati.

In ogni caso, si garantisce che l'ordine delle operazioni è stato rispettato e che gli istanti di inizio e fine sono coerenti con i dati a disposizione, sebbene non si possa sempre essere certi della loro effettiva durata.

La visione logica di cui si è parlato in precedenza comporta che un unico processo, il client, si occupi di ogni operazione che non sia un *task* principale dell'applicazione. In realtà alcune di queste sono state registrate nei log contemporaneamente, in quanto i componenti che le hanno portate a termine sono in esecuzione contemporanea. Ad esempio, la creazione di un reducer *mentre* un mapper è ancora in esecuzione è possibile perché il *JobTracker* che ne comanda la creazione è distinto dal *TaskTracker*. La contemporaneità

delle operazioni non può tuttavia essere modellata quando da quattro processi in parallelo ci si riduce ad uno solo.

In questo caso si è scelto di eseguire sequenzialmente le operazioni non riconducibili a task diversi e sovrapposte, posticipando leggermente quella creata più di recente. Nuovamente, questa scelta fa discostare il workload ricavato da quello effettivo, anche se mai per più di alcuni millisecondi.

Si noti, infine, che non è dato a sapere quanto tempo trascorre dall'inizio o dalla fine dell'operazione alla scrittura del file di log, o se il *timestamp* associato alle informazioni scritte nei file corrisponde comunque all'effettivo istante di avvenimento dell'operazione. La ricostruzione assume quest'ultima ipotesi.

### 4.2.3 Ipotesi sulle operazioni CPU-bound e I/O-bound

Il taskgraph ricavato assume la presenza di una fase I/O-bound se un'operazione di lettura o scrittura è *esplicitamente* riportata da un file di log. In assenza di indicazioni in tal senso una fase è considerata CPU-bound.

Mentre è possibile riscontrare il primo tipo di fase dai log, le fasi CPU non sono monitorate in alcun modo, in quanto nessun componente rendiconta le operazioni effettuate in memoria che non siano la creazione o la terminazione di nuovi *task*. Si supporrà, dunque, che, in mancanza di informazioni memorizzate, il processo stia eseguendo una computazione intensiva.

In caso di più scritture o letture consecutive, verrà creata *una sola fase* comprendente tutte queste operazioni. La situazione è tipica della memorizzazione sull'HDFS degli input per i mapper e della lettura dei loro output da parte dei reducer, composte in realtà da *più file* (e quindi da più operazioni consecutive).

Dai file di log si cercherà di capire il *motivo* di una scrittura o lettura dall'HDFS, utilizzando le informazioni contenute in essi o delle supposizioni basate sul comportamento previsto per un'applicazione MapReduce, ricostruendo tutte le fasi che la teoria prescrive per queste.

### 4.2.4 Semplificazioni su mapper e reducer

Come già accennato, i vari mapper eseguono le stesse operazioni e generano il medesimo numero di punti. Assumere che siano tutti composti dalle stesse fasi e che la loro composizione sia identica non dovrebbe costituire un'ipotesi troppo azzardata, come d'altronde confermato dai file di log.

Con l'aiuto di un'altra applicazione presente nel bundle di Hadoop, chiamata *sleep*<sup>5</sup>, è stato possibile notare che, contrariamente a quanto si potrebbe credere, il *JobTracker* programma l'esecuzione di un reducer a partire dal termine di *un singolo mapper* e non alla conclusione di tutti questi. Sebbene

---

<sup>5</sup>Il programma *sleep* consente di simulare l'esecuzione di un numero di mapper e reducer arbitrario, ognuno attivo per il tempo desiderato.

questo comportamento non sia presente in *pi*, probabilmente per il ridotto intervallo in cui tutti i mapper terminano, in *sleep* questo fenomeno è riscontrabile chiaramente.

Dovrebbe essere impossibile per un reducer operare senza la presenza di tutti i suoi input, anche se è possibile ipotizzare la creazione preventiva di questo, in modo da ridurre il tempo di attesa una volta che tutti i mapper hanno completato il loro compito. Anche se non rilevante al caso specifico, si ritiene interessante segnalare questo comportamento potenzialmente contro intuitivo.

### 4.3 Analisi del workload

Il programma *pi* è stato configurato per l'esecuzione con *dieci* mapper, ognuno con *mille* punti da generare. Il job richiesto dall'applicazione è stato l'unico in esecuzione su Hadoop dal momento dell'attivazione del framework alla sua chiusura, in modo che i file di log non potessero includere operazioni esterne ad esso. Dai due istanti sono inoltre stati fatti trascorrere alcuni minuti, per permettere ad Hadoop di terminare le operazioni ausiliarie programmate alla sua attivazione o alla chiusura senza interferenze.

L'applicazione è stata eseguita in una distribuzione stand-alone di Apache Hadoop, versione 1.1.2, installata su una macchina virtuale Linux configurata con 4,4GB di RAM. L'host opera su un processore Intel i7, composto da 4 core alla frequenza di 3,01GHz e con tecnologia **Hyperthreading** attiva, risultando quindi in 8 processori logici visibili al sistema operativo. I tempi ottenuti fanno riferimento a questo ambiente.

La ricostruzione del workload è stata possibile grazie a questi strumenti:

- I *log* dei componenti di Hadoop che tracciano l'attività di ognuno di essi e ne rendicontano le operazioni principali.
- Un semplice script in grado di memorizzare i processi in esecuzione nel sistema ad intervalli molto fini (10ms), tramite il comando *ps*<sup>6</sup> disponibile in Linux.

Lo script è stato utilizzato per derivare efficacemente il *numero* e il *tipo* di processi in esecuzione nel tempo, mentre sono stati utilizzati i log per comprendere la composizione di ognuno di essi e a scopo di verifica e validazione di quanto collezionato tramite script.

Si è cercato di collegare le operazioni effettuate e segnalate dai log con i *motivi* per il quale queste sono state compiute in base ai commenti scritti in essi, ai nomi dei file acceduti e agli istanti temporali in cui queste avvengono.

---

<sup>6</sup>Il comando *ps* consente di ottenere la lista dei processi in esecuzione nel momento in cui viene utilizzato, assieme alla loro gerarchia.

### 4.3.1 Ricostruzione del workload

La figura 14 mostra il taskgraph ricavato dall'esecuzione di *pi* grazie agli strumenti descritti in precedenza. Nel resto della sezione si discute la sua struttura e come sia stato possibile ricostruirlo, tenendo a mente l'algoritmo implementato dall'applicazione riportato a pagina 109.

Il taskgraph è simile a quello di esempio già analizzato nel terzo capitolo. La riga centrale rappresenta il *client* mentre quelle sopra e sotto rappresentano gli altri *task*. Seguendo la visione logica proposta, questi sono creati esclusivamente dal client, che si avvale dell'operato di nuovi processi per eseguire i vari task attivi in MapReduce, logicamente distinti dall'applicazione.

I task creati sono di quattro tipi distinti: all'inizio e alla fine dell'esecuzione il client esegue una fase di FORK per creare la JobSetup e la JobCleanup (segnalati in figura dai numeri 1 e 4 rispettivamente), mentre gli altri due task presenti sono i dieci mapper e il singolo reducer (numeri 2 e 3). Non è stato possibile trovare alcun messaggio nei log riguardo alle operazioni eseguite dai primi due task, pertanto sono stati caratterizzati da una *singola* fase CPU-bound.

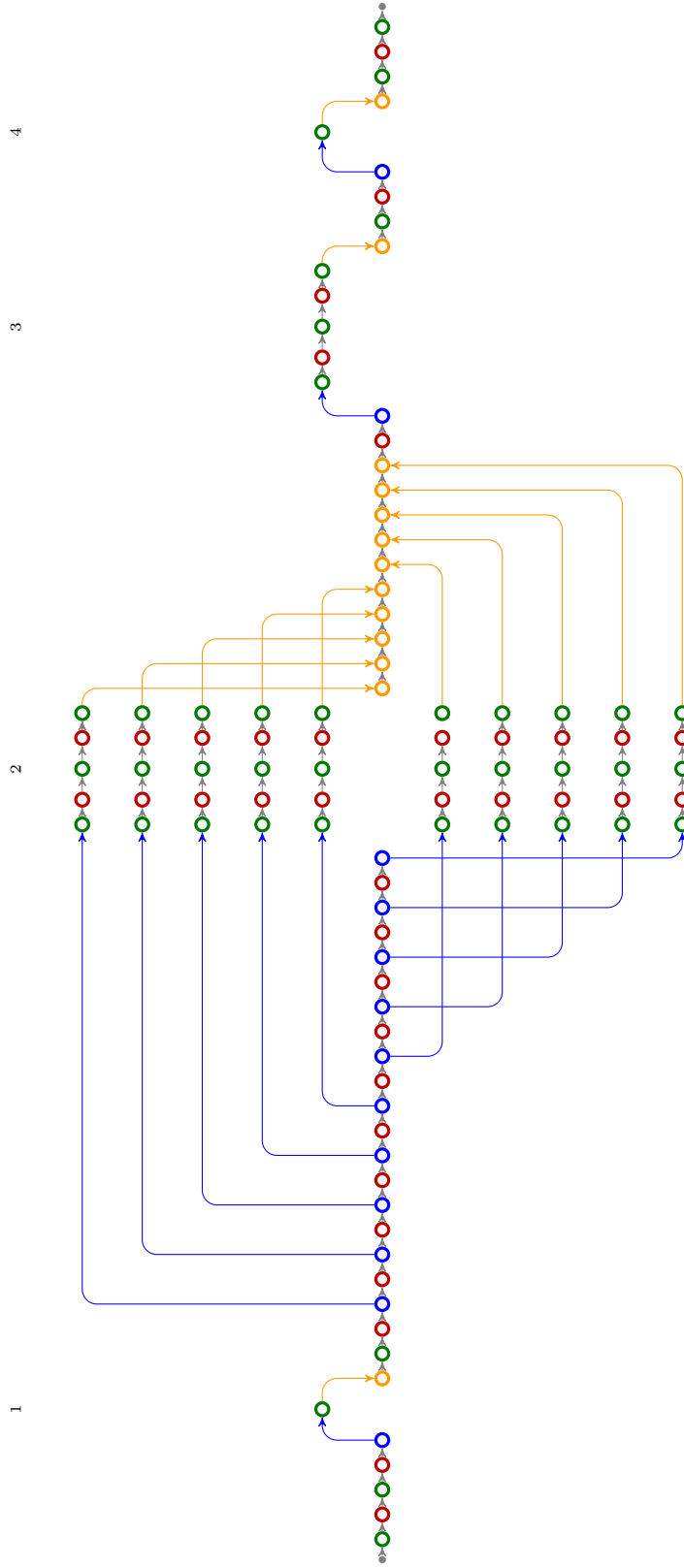
Al contrario, mapper e reducer presentano un comportamento più complesso, composto da cinque fasi, alternando CPU e I/O. Le due fasi I/O-bound consistono nella lettura dei file di *input* per il task e nella scrittura del suo output rispettivamente. La lettura, per entrambi i task, è documentata dal log del *DataNode*, mentre la scrittura dell'output è rendicontata dal *TaskTracker*.

La fase CPU-bound presente tra le due operazioni di input/output rappresenta l'*esecuzione effettiva* del mapper o del reducer, mentre le altre due sono state introdotte in quanto lo script ha registrato la presenza di questi task anche in precedenza e in seguito alle fasi I/O-bound, senza che adeguate informazioni fossero collezionate nei log.

La prima può rappresentare l'inizializzazione del task, in quanto, non avendo ancora letto il file di input, questo non può svolgere alcuna operazione. Come si vedrà dal profilo temporale dell'esecuzione, riportato più avanti, questa fase è *molto lunga*, a riprova che *pi* è uno scarso candidato all'esecuzione in Hadoop: i tempi di creazione e setup di un nuovo task sono molto onerosi, e non vale la pena eseguire un'applicazione il cui tempo di vita è relativamente basso. Il framework è da preferire per applicazioni che impiegano molto tempo e che operano su grandi quantità di dati.

La seconda può, al contrario, rappresentare la fase finale del task, in attesa della sua terminazione. I suoi tempi di esecuzione sono più contenuti e dopo la scrittura dell'output non vi sono altre operazioni da svolgere.

La *prima* e l'*ultima* operazione I/O-bound del *client* rappresentano la scrittura dell'input per i mapper e la lettura dell'output del reducer rispettivamente. Il codice sorgente specifica una cartella temporanea in cui scrivere i



**Figura 14:** Taskgraph ricostruito del programma  $\pi$ , in esecuzione con dieci mapper. La sua struttura è simile a quella del taskgraph riportato nella figura 7 ed è descritto a partire dalla pagina 114. Il colore dei cerchi rappresenta il tipo della fase, secondo la legenda: CPU, I/O, FORK e JOIN. I numeri nella parte alta della figura identificano i differenti task MapReduce con questo significato: 1: JobSetup, 2: mapper, 3: reducer, 4: JobCleanup.

file di input, e il log del *DataNode* conferma una scrittura di dieci file in tale locazione all'inizio del client.

La lettura dell'output finale è necessaria affinché il client possa calcolare il valore approssimato di  $\pi$ . Come già commentato infatti, il reducer si limita a scrivere come output due coppie  $(numInterni, valore)$ ,  $(numEsterni, valore)$ , senza utilizzare la formula (4.1). È il client a leggere i risultati prodotti dal reducer e ad applicarla, come si può vedere dai codici sorgenti. L'ultima operazione del client, CPU-bound, è la comunicazione del valore di  $\pi$  all'utente.

Come si può notare dal taskgraph, immediatamente *prima* di *ogni* FORK il client esegue una fase I/O-bound. Queste sono *tutte* delle scritture su disco, in particolare il log del *TaskTracker* riporta in corrispondenza di queste la memorizzazione del file `taskjvm.sh` che contiene i comandi necessari al lancio di una nuova JVM per il task da eseguire.

In caso di creazione di più task dello stesso tipo il pattern I/O-FORK è ripetuto per ognuno di questi, poiché il *TaskTracker* riporta *per ogni* task la memorizzazione del proprio file dedicato, *quindi* la sua creazione. Questo motiva la presenza di dieci FORK distinte intervallate da una fase I/O-bound anziché una singola FORK creante dieci mapper.

Le fasi di JOIN che ne conseguono sono *logicamente* equivalenti ad una singola aspettante dieci processi diversi. Si è preferito separarle per consentire la memorizzazione dei loro istanti di inizio e di fine al momento della riproduzione del workload.

Le poche fasi CPU-bound rimanenti nel client sono state aggiunte per modellare attività non presenti nei file di log. Generalmente si riferiscono a degli istanti temporali che separano due fasi documentate.

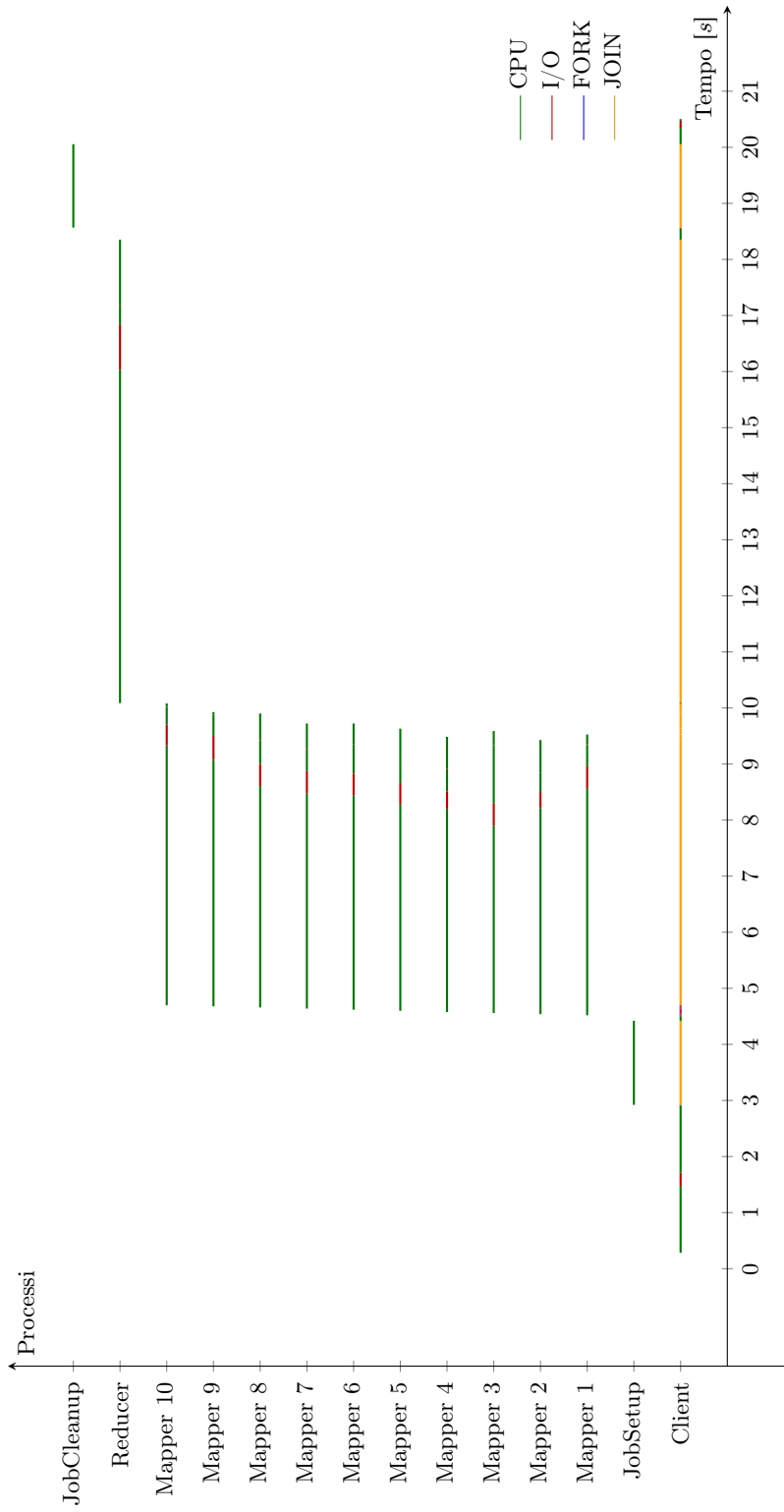
### 4.3.2 Ricostruzione del profilo temporale dell'esecuzione

Grazie alle informazioni contenute nei log e ottenute grazie allo script è stato possibile eseguire un'istanza dell'applicazione *pi* e ricostruire gli istanti di inizio e di fine delle fasi definite nella sezione precedente.

La procedura è stata fatta *manualmente*, eseguendo il programma una sola volta e creando a mano i dati necessari alla creazione del *profilo temporale dell'esecuzione* (si veda la sezione 3.6.2).

Sono stati utilizzati i file di log per determinare i tempi delle operazioni CPU-bound, I/O-bound e la durata della fasi FORK, mentre i dati raccolti dallo script hanno determinato gli istanti di inizio e fine dei processi, con i conseguenti tempi di durata delle fasi JOIN. I risultati ricostruiti sono presenti nella figura 15.

Data la presenza di molte operazioni poco visibili dai grafici, a causa della loro durata contenuta, alla fine del capitolo si presenteranno delle porzioni estese del profilo a confronto con i risultati ottenuti dal benchmark.



**Figura 15:** Ricostruzione del *profilo temporale dell'esecuzione* di  $\pi_i$ , assunto caratterizzato dal taskgraph mostrato nella figura 14 e con le semplificazioni discusse nella sezione 4.2. La legenda indica la corrispondenza di ogni colore al relativo tipo di fase.

#### 4.4 Riproduzione del taskgraph

Il taskgraph derivato per *pi* e mostrato nella figura 14 è stato codificato nella memoria del *generatore di workload* con l'obiettivo di riprodurre un comportamento fedele all'originale. In particolare, tramite un'accurata procedura di *fitting*<sup>7</sup>, si è cercato di riprodurre un carico di lavoro il più possibile conforme al profilo temporale dell'esecuzione ricavato.

La validazione è avvenuta nello *stesso ambiente* in cui i dati per *pi* sono stati ricavati. La tabella 6 mostra i valori con i quali è stato configurato il taskgraph. Similmente all'esempio proposto nel terzo capitolo si è scelto di caratterizzare gli id delle fasi con quattro cifre (o con tre quando si sottintende uno 0 iniziale), le prime due indicanti il *processo* e le ultime due rappresentanti il *numero* della fase relativo al processo stesso.

Laddove possibile si è descritta l'operazione reale modellata da una fase, come descritto nella sezione 4.3.1.

Al termine del capitolo si riportano a titolo dimostrativo i grafici e le statistiche testuali prodotti da un'esecuzione del benchmark, confrontandole con quelle ricavate dai log.

Si è scelto di riportare il grafico di una singola riproduzione del workload. Esecuzioni diverse possono differire lievemente nella durata delle varie fasi per via dei meccanismi di scheduling e del traffico già presente nell'ambiente di analisi, nonché delle ottimizzazioni sulle operazioni di input/output, come si è discusso a pagina 50.

La presenza, in una versione successiva del benchmark, della fase di *calibrazione* può migliorare sensibilmente i risultati.

**Tabella 6:** Configurazione del taskgraph validante *pi*. La colonna "ITER." riporta il numero di iterazioni impostate per una fase CPU-bound o I/O-bound, mentre l'ultima riporta la corrispondente fase di *pi* modellata o i processi creati o aspettati, dove opportuno.

PROCESSO	ID	TIPO	ITER.	DESCRIZIONE
Client	101	CPU	2000	-
Client	102	I/O	100	Scrittura input Mapper
Client	103	CPU	2000	-
Client	104	I/O	50	Scrittura comandi per JobSetup
Client	105	FORK	-	Crea fase 201 (JobSetup)
Client	106	JOIN	-	Aspetta fase 201 (JobSetup)
Client	107	CPU	126	-
Client	108	I/O	3	Scrittura comandi per Mapper 1

*Continua nella prossima pagina*

<sup>7</sup>In questo contesto, si denota con il termine *fitting* il processo di selezione e adattamento dei parametri di un modello, al fine di ottenere con esso la migliore approssimazione possibile della realtà modellata.



*Continua dalla pagina precedente*

PROCESSO	ID	TIPO	ITER.	DESCRIZIONE
Client	109	FORK	-	Crea fase 301 (Mapper 1)
Client	110	I/O	3	Scrittura comandi per Mapper 2
Client	111	FORK	-	Crea fase 401 (Mapper 2)
Client	112	I/O	3	Scrittura comandi per Mapper 3
Client	113	FORK	-	Crea fase 501 (Mapper 3)
Client	114	I/O	3	Scrittura comandi per Mapper 4
Client	115	FORK	-	Crea fase 601 (Mapper 4)
Client	116	I/O	3	Scrittura comandi per Mapper 5
Client	117	FORK	-	Crea fase 701 (Mapper 5)
Client	118	I/O	3	Scrittura comandi per Mapper 6
Client	119	FORK	-	Crea fase 801 (Mapper 6)
Client	120	I/O	3	Scrittura comandi per Mapper 7
Client	121	FORK	-	Crea fase 901 (Mapper 7)
Client	122	I/O	3	Scrittura comandi per Mapper 8
Client	123	FORK	-	Crea fase 1001 (Mapper 8)
Client	124	I/O	3	Scrittura comandi per Mapper 9
Client	125	FORK	-	Crea fase 1101 (Mapper 9)
Client	126	I/O	3	Scrittura comandi per Mapper 10
Client	127	FORK	-	Crea fase 1201 (Mapper 10)
Client	128	JOIN	-	Aspetta fase 301 (Mapper 1)
Client	129	JOIN	-	Aspetta fase 401 (Mapper 2)
Client	130	JOIN	-	Aspetta fase 501 (Mapper 3)
Client	131	JOIN	-	Aspetta fase 601 (Mapper 4)
Client	132	JOIN	-	Aspetta fase 701 (Mapper 5)
Client	133	JOIN	-	Aspetta fase 801 (Mapper 6)
Client	134	JOIN	-	Aspetta fase 901 (Mapper 7)
Client	135	JOIN	-	Aspetta fase 1001 (Mapper 8)
Client	136	JOIN	-	Aspetta fase 1101 (Mapper 9)
Client	137	JOIN	-	Aspetta fase 1201 (Mapper 10)
Client	138	I/O	20	Scrittura comandi per Reducer
Client	139	FORK	-	Crea fase 1301 (Reducer)
Client	140	JOIN	-	Aspetta fase 1301 (Reducer)
Client	141	CPU	326	-
Client	142	I/O	30	Scrittura comandi per JobCleanup
Client	143	FORK	-	Crea fase 1401 (JobCleanup)
Client	144	JOIN	-	Aspetta fase 1401 (JobCleanup)
Client	145	CPU	730	-
Client	146	I/O	300	Lettura output Reducer
Client	147	CPU	100	Calcolo $\pi$ e terminazione

*Continua nella prossima pagina*

*Continua dalla pagina precedente*

PROCESSO	ID	TIPO	ITER.	DESCRIZIONE
JobSetup	201	CPU	2300	-
Mapper 1	301	CPU	600	Inizializzazione
Mapper 1	302	I/O	8	Lettura input
Mapper 1	303	CPU	110	Esecuzione
Mapper 1	304	I/O	2	Scrittura output
Mapper 1	305	CPU	91	Terminazione
Mapper 2	401	CPU	600	Inizializzazione
Mapper 2	402	I/O	8	Lettura input
Mapper 2	403	CPU	110	Esecuzione
Mapper 2	404	I/O	2	Scrittura output
Mapper 2	405	CPU	91	Terminazione
Mapper 3	501	CPU	600	Inizializzazione
Mapper 3	502	I/O	8	Lettura input
Mapper 3	503	CPU	110	Esecuzione
Mapper 3	504	I/O	2	Scrittura output
Mapper 3	505	CPU	91	Terminazione
Mapper 4	601	CPU	600	Inizializzazione
Mapper 4	602	I/O	8	Lettura input
Mapper 4	603	CPU	110	Esecuzione
Mapper 4	604	I/O	2	Scrittura output
Mapper 4	605	CPU	91	Terminazione
Mapper 5	701	CPU	600	Inizializzazione
Mapper 5	702	I/O	8	Lettura input
Mapper 5	703	CPU	110	Esecuzione
Mapper 5	704	I/O	2	Scrittura output
Mapper 5	705	CPU	91	Terminazione
Mapper 6	801	CPU	600	Inizializzazione
Mapper 6	802	I/O	8	Lettura input
Mapper 6	803	CPU	110	Esecuzione
Mapper 6	804	I/O	2	Scrittura output
Mapper 6	805	CPU	91	Terminazione
Mapper 7	901	CPU	600	Inizializzazione
Mapper 7	902	I/O	8	Lettura input
Mapper 7	903	CPU	110	Esecuzione
Mapper 7	904	I/O	2	Scrittura output
Mapper 7	905	CPU	91	Terminazione

*Continua nella prossima pagina*

*Continua dalla pagina precedente*

PROCESSO	ID	TIPO	ITER.	DESCRIZIONE
Mapper 8	1001	CPU	600	Inizializzazione
Mapper 8	1002	I/O	8	Lettura input
Mapper 8	1003	CPU	110	Esecuzione
Mapper 8	1004	I/O	2	Scrittura output
Mapper 8	1005	CPU	91	Terminazione
Mapper 9	1101	CPU	600	Inizializzazione
Mapper 9	1102	I/O	8	Lettura input
Mapper 9	1103	CPU	110	Esecuzione
Mapper 9	1104	I/O	2	Scrittura output
Mapper 9	1105	CPU	91	Terminazione
Mapper 10	1201	CPU	600	Inizializzazione
Mapper 10	1202	I/O	8	Lettura input
Mapper 10	1203	CPU	110	Esecuzione
Mapper 10	1204	I/O	2	Scrittura output
Mapper 10	1205	CPU	91	Terminazione
Reducer	1301	CPU	9300	Inizializzazione
Reducer	1302	I/O	500	Lettura input
Reducer	1303	CPU	1050	Esecuzione
Reducer	1304	I/O	21	Scrittura output
Reducer	1305	CPU	1500	Terminazione
JobCleanup	1401	CPU	2380	-

*Si conclude dalla pagina precedente*

## 4.5 Commenti sui risultati

I grafici prodotti dal benchmark non si discostano in maniera sensibile dal profilo di  $pi$  ricostruito. L'ordine delle operazioni è conservato e gli istanti di inizio e fine sono coerenti, ammettendo una leggera varianza.

Si ricordi, inoltre, che il profilo ricostruito è basato sulle ipotesi e sulle semplificazioni di cui si è parlato in precedenza, rendendo alcuni tempi mostrati nel grafico che lo riguarda leggermente differenti da quelli reali.

Il numero di mapper impostato è volutamente superiore ai processori logici disponibili nel sistema, per mostrare l'effetto discusso a pagina 56: nonostante siano individuabili fino a undici processi in esecuzione contemporanea, solo otto di essi al massimo potranno essere *effettivamente* eseguiti contemporaneamente. L'effetto è visibile nella tabella 7, in cui si nota come per alcune fasi il processore iniziale differisca da quello finale, sintomo di una rimozione temporanea dall'esecuzione da parte dello scheduler.

I confronti tra i due profili sono riportati nelle figure 17, 18 e 19 della prossima sezione. I grafici dell'esecuzione di  $pi$  sono accompagnati da ingrandimenti di alcuni intervalli temporali, per mostrare operazioni altrimenti troppo rapide per essere visualizzate.

Si è scelto di non effettuare la stessa operazione per i profili della simulazione, in quanto le fasi di FORK, spesso oggetto degli ingrandimenti, hanno durata non controllabile e sono generalmente molto più brevi delle corrispettive eseguite da  $pi$ , al punto da non essere visibili neppure in una estensione dedicata. La loro presenza è comunque provata dalla tabella 7.

Il lettore interessato potrà facilmente individuare nei grafici le fasi definite nel taskgraph, verificandone la presenza tramite gli ingrandimenti proposti e i segmenti visibili.

Si sono infine riportati i profili temporali delle fasi prodotti dall'analizzatore di log in seguito alla riproduzione del taskgraph. I confronti della figura 23 mostrano l'alternanza tra le fasi I/O-CPU e FORK-JOIN, come ci si aspetterebbe dal traffico definito.

L'autore ritiene che i risultati ottenuti siano sufficienti a confermare la correttezza del benchmark realizzato.

#### 4.6 Statistiche e grafici

In chiusura di capitolo si riportano i dati dell'esecuzione del taskgraph configurato come indicato dalla tabella 6 ed elaborati dall'analizzatore di log, assieme a dei confronti tra i profili ricostruiti e quelli della simulazione.

La composizione di ogni tabella o figura a seguire è stata commentata in precedenza nella sezione 3.6, cui si rimanda per ulteriori spiegazioni.

**Tabella 7:** File di statistiche testuali prodotto dalla simulazione di  $\pi$ .

PID	ID	TIPO	P.I.	P.F.	ITER.	INIZIO [ms]	FINE [ms]	NOTE
14098	101	CPU	5	2	2000	0.000000	1175.925037	-
14098	102	I/O	2	2	100	1175.927008	1253.190201	-
14098	103	CPU	2	2	2000	1253.190991	2521.015791	-
14098	104	I/O	2	2	50	2521.017306	2571.050256	-
14098	105	FORK	2	2	-	2571.051642	2571.122104	Creato: 14100
14100	201	CPU	3	3	2300	2571.121851	3967.858293	-
14098	106	JOIN	2	2	-	2571.123021	3968.095402	Aspettato: 14100
14098	107	CPU	2	2	126	3968.096425	4048.121272	-
14098	108	I/O	2	2	3	4048.122684	4048.210770	-
14098	109	FORK	2	2	-	4048.211341	4048.261755	Creato: 14101
14098	110	I/O	2	2	3	4048.262504	4066.143127	-
14101	301	CPU	3	1	600	4048.267851	7929.063451	-
14098	111	FORK	2	2	-	4066.145758	4066.215126	Creato: 14102
14098	112	I/O	2	2	3	4066.215892	4066.317014	-
14102	401	CPU	4	1	600	4066.229173	7426.211480	-
14098	113	FORK	2	2	-	4066.317573	4066.346455	Creato: 14103
14098	114	I/O	2	2	3	4066.346959	4098.599045	-
14103	501	CPU	5	4	600	4066.426434	7393.277510	-
14098	115	FORK	2	2	-	4098.600361	4098.648392	Creato: 14104
14098	116	I/O	2	2	3	4098.648929	4198.245482	-
14104	601	CPU	5	2	600	4098.815062	8405.615049	-
14098	117	FORK	2	2	-	4198.248541	4198.303842	Creato: 14105
14098	118	I/O	2	2	3	4198.304415	4266.155715	-
14105	701	CPU	3	0	600	4198.364880	7704.246517	-
14098	119	FORK	2	2	-	4266.157698	4329.472058	Creato: 14106
14106	801	CPU	3	1	600	4266.219353	8603.409063	-
14098	120	I/O	2	2	3	4329.488774	4455.173290	-
14098	121	FORK	2	2	-	4455.174788	4470.995887	Creato: 14107
14107	901	CPU	3	4	600	4455.240414	8535.915906	-
14098	122	I/O	2	2	3	4470.997385	4568.325664	-
14098	123	FORK	2	2	-	4568.326765	4568.363304	Creato: 14108
14098	124	I/O	2	2	3	4568.363822	4586.111047	-
14108	1001	CPU	3	2	600	4568.440298	7085.655607	-
14098	125	FORK	2	2	-	4586.112785	4665.597358	Creato: 14109
14109	1101	CPU	3	6	600	4586.364360	7425.895940	-
14098	126	I/O	2	2	3	4665.599019	4698.307065	-
14098	127	FORK	2	2	-	4698.308347	4698.357886	Creato: 14110
14098	128	JOIN	2	6	-	4698.358595	9159.855378	Aspettato: 14101
14110	1201	CPU	3	0	600	4698.403531	9511.140568	-

*Continua nella prossima pagina*

*Continua dalla pagina precedente*

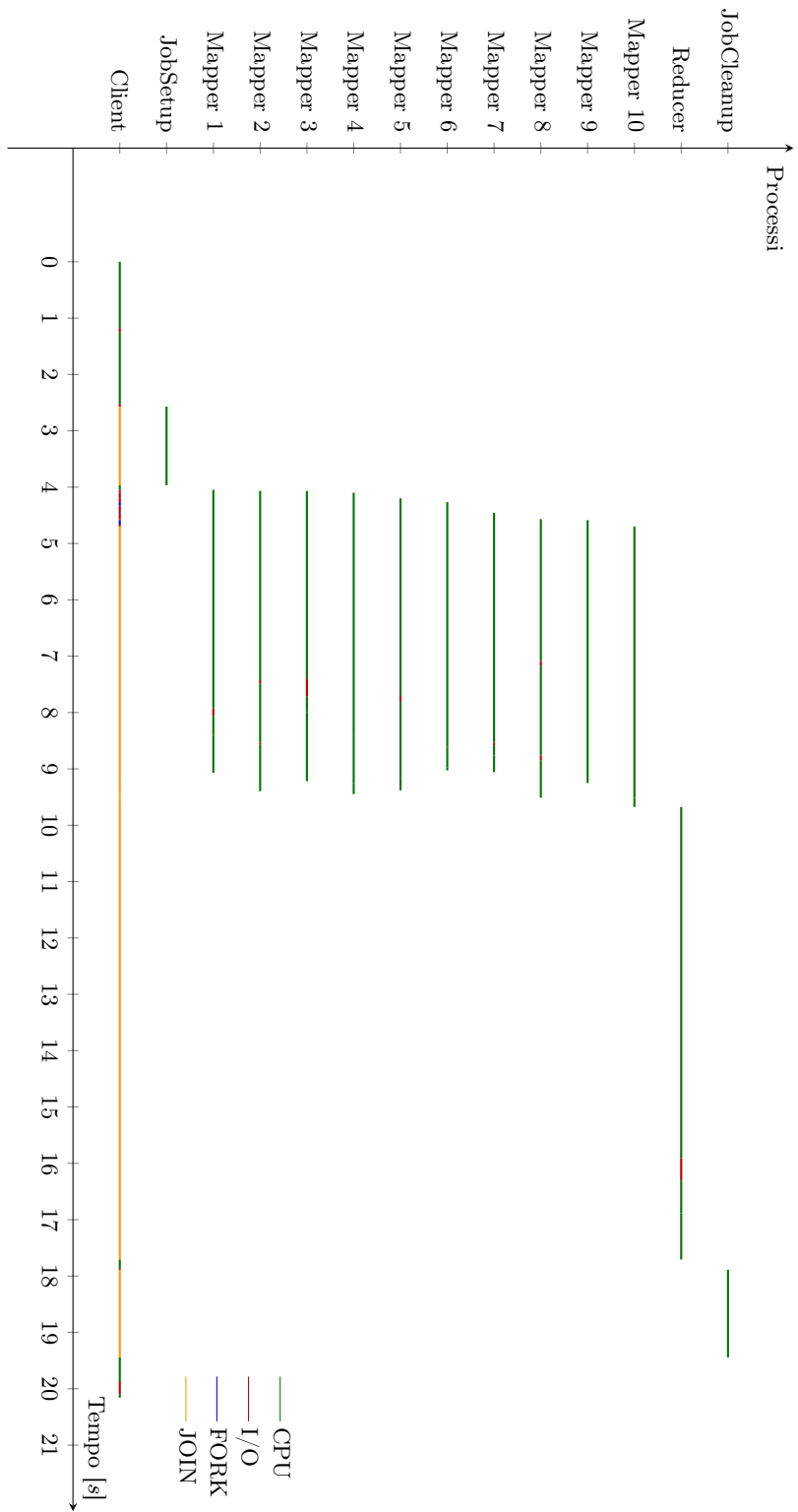
PID	ID	TIPO	P.I.	P.F.	ITER.	INIZIO [ms]	FINE [ms]	NOTE
14108	1002	I/O	2	0	8	7085.657602	7165.637332	-
14108	1003	CPU	0	5	110	7165.638267	8763.277124	-
14103	502	I/O	4	4	8	7393.278941	7718.088353	-
14109	1102	I/O	6	6	8	7425.897425	7426.203732	-
14109	1103	CPU	6	7	110	7426.204324	8090.263603	-
14102	402	I/O	1	5	8	7426.212538	7490.889813	-
14102	403	CPU	5	7	110	7490.890832	8537.644544	-
14105	702	I/O	0	1	8	7704.248102	7813.702013	-
14103	503	CPU	4	0	110	7718.090517	7994.309181	-
14105	703	CPU	1	0	110	7813.703547	9160.131369	-
14101	302	I/O	1	1	8	7929.065154	8058.436662	-
14103	504	I/O	0	0	2	7994.310782	7994.395271	-
14103	505	CPU	0	0	91	7994.395793	9218.885819	-
14101	303	CPU	1	0	110	8058.437802	8392.352427	-
14109	1104	I/O	7	7	2	8090.265249	8090.351595	-
14109	1105	CPU	7	0	91	8090.352128	9251.728016	-
14101	304	I/O	0	0	2	8392.354075	8392.424736	-
14101	305	CPU	0	5	91	8392.425268	9070.880139	-
14104	602	I/O	2	2	8	8405.616614	8405.798822	-
14104	603	CPU	2	2	110	8405.799454	9251.437348	-
14107	902	I/O	4	4	8	8535.917619	8583.494371	-
14102	404	I/O	7	7	2	8537.646373	8568.457611	-
14102	405	CPU	7	4	91	8568.458955	9395.947820	-
14107	903	CPU	4	4	110	8583.495443	8762.875293	-
14106	802	I/O	1	1	8	8603.410602	8615.682107	-
14106	803	CPU	1	1	110	8615.683025	8973.082854	-
14107	904	I/O	4	4	2	8762.877154	8762.988768	-
14107	905	CPU	4	4	91	8762.989354	9057.587921	-
14108	1004	I/O	5	5	2	8763.278405	8843.505054	-
14108	1005	CPU	5	2	91	8843.506043	9511.218504	-
14106	804	I/O	1	1	2	8973.091648	8973.122175	-
14106	805	CPU	1	1	91	8973.122604	9025.685035	-
14098	129	JOIN	6	4	-	9159.856863	9428.122032	Aspettato: 14102
14105	704	I/O	0	0	2	9160.132506	9160.303559	-
14105	705	CPU	0	1	91	9160.304215	9380.775558	-
14104	604	I/O	2	2	2	9251.439129	9251.532115	-
14104	605	CPU	2	2	91	9251.532703	9445.990397	-
14098	130	JOIN	4	4	-	9428.123294	9428.162864	Aspettato: 14103
14098	131	JOIN	4	4	-	9428.163249	9479.212846	Aspettato: 14104
14098	132	JOIN	4	4	-	9479.215285	9479.260099	Aspettato: 14105

*Continua nella prossima pagina*

*Continua dalla pagina precedente*

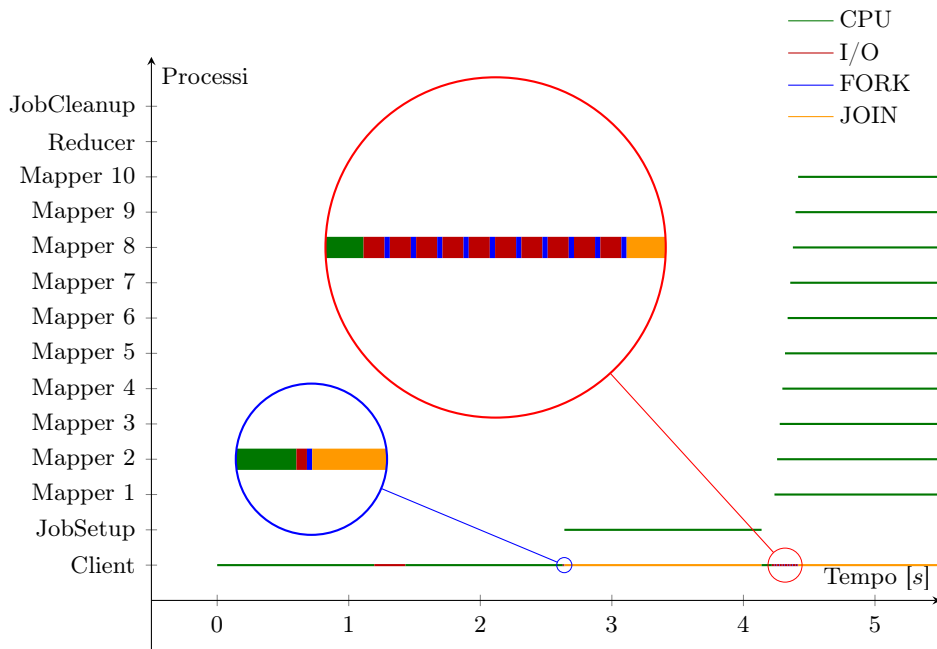
PID	ID	TIPO	P.I.	P.F.	ITER.	INIZIO [ms]	FINE [ms]	NOTE
14098	133	JOIN	4	4	-	9479.260485	9479.382075	Aspettato: 14106
14098	134	JOIN	4	4	-	9479.382745	9479.513299	Aspettato: 14107
14098	135	JOIN	4	2	-	9479.522151	9525.641079	Aspettato: 14108
14110	1202	I/O	0	0	8	9511.142717	9511.303061	-
14110	1203	CPU	0	0	110	9511.303610	9608.756677	-
14098	136	JOIN	2	2	-	9525.642358	9544.188729	Aspettato: 14109
14098	137	JOIN	2	2	-	9544.190628	9674.437710	Aspettato: 14110
14110	1204	I/O	0	0	2	9608.758424	9608.829328	-
14110	1205	CPU	0	0	91	9608.829887	9674.021414	-
14098	138	I/O	2	2	20	9674.438583	9675.068509	-
14098	139	FORK	2	2	-	9675.069402	9675.165983	Creato: 14112
14112	1301	CPU	3	3	9300	9675.154795	15911.143330	-
14098	140	JOIN	2	3	-	9675.166731	17706.086309	Aspettato: 14112
14112	1302	I/O	3	0	500	15911.144977	16298.825188	-
14112	1303	CPU	0	3	1050	16298.826218	16885.659628	-
14112	1304	I/O	3	3	21	16885.661176	16885.990692	-
14112	1305	CPU	3	3	1500	16885.991401	17705.566888	-
14098	141	CPU	3	3	326	17706.086989	17868.370719	-
14098	142	I/O	3	3	30	17868.372321	17888.879001	-
14098	143	FORK	3	3	-	17888.880190	17888.943869	Creato: 14114
14114	1401	CPU	4	5	2380	17888.940003	19445.822540	-
14098	144	JOIN	3	5	-	17888.944568	19445.998890	Aspettato: 14114
14098	145	CPU	5	5	730	19445.999598	19865.204323	-
14098	146	I/O	5	5	300	19865.206115	20094.086298	-
14098	147	CPU	5	5	100	20094.087670	20158.141588	-

*Si conclude dalla pagina precedente*

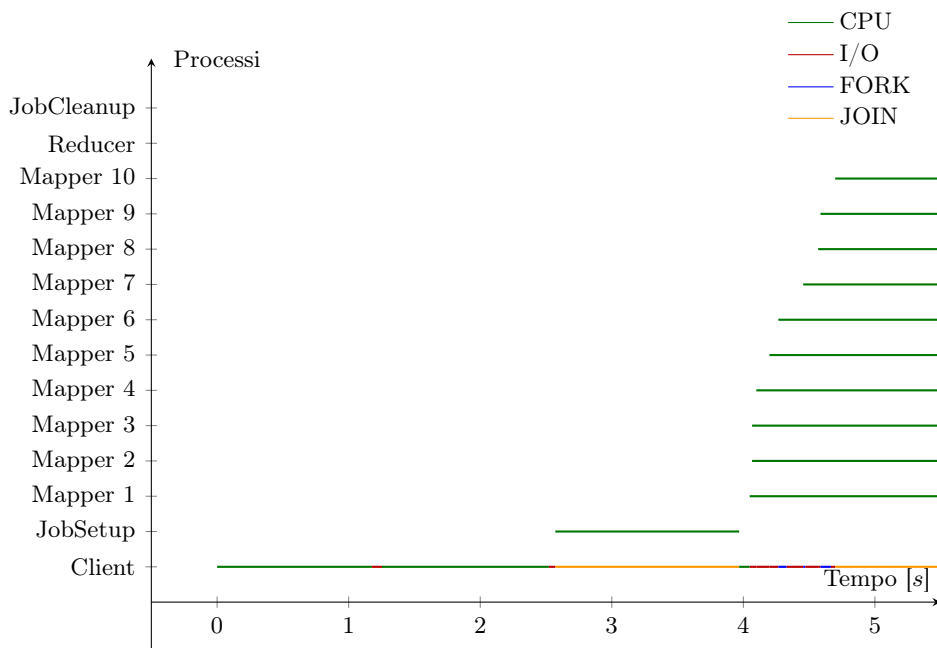


**Figura 16:** *Profilo temporale dell'esecuzione della simulazione di pi.*



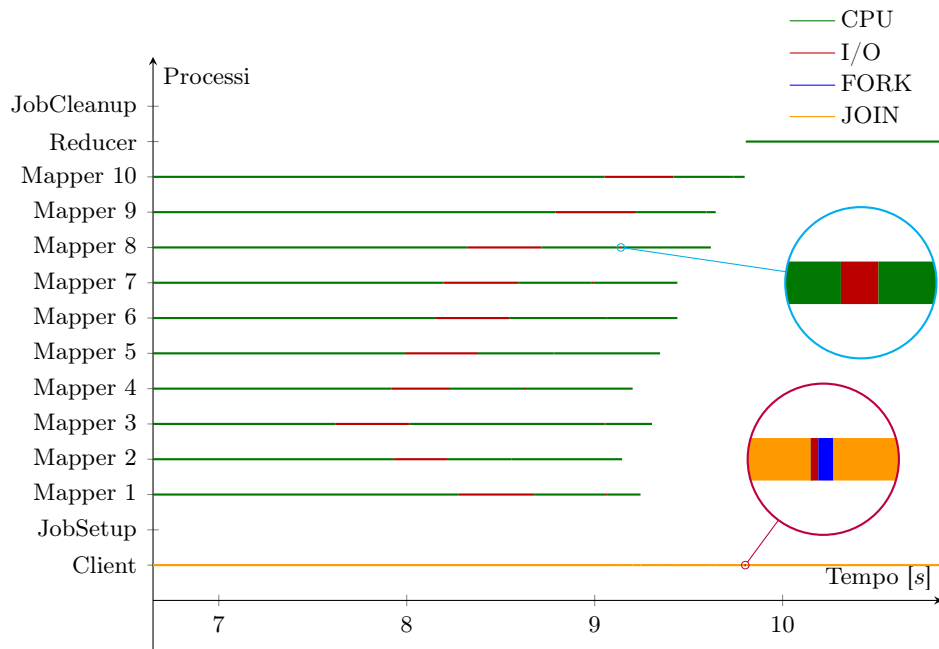


(a) Ricostruzione dell'esecuzione.

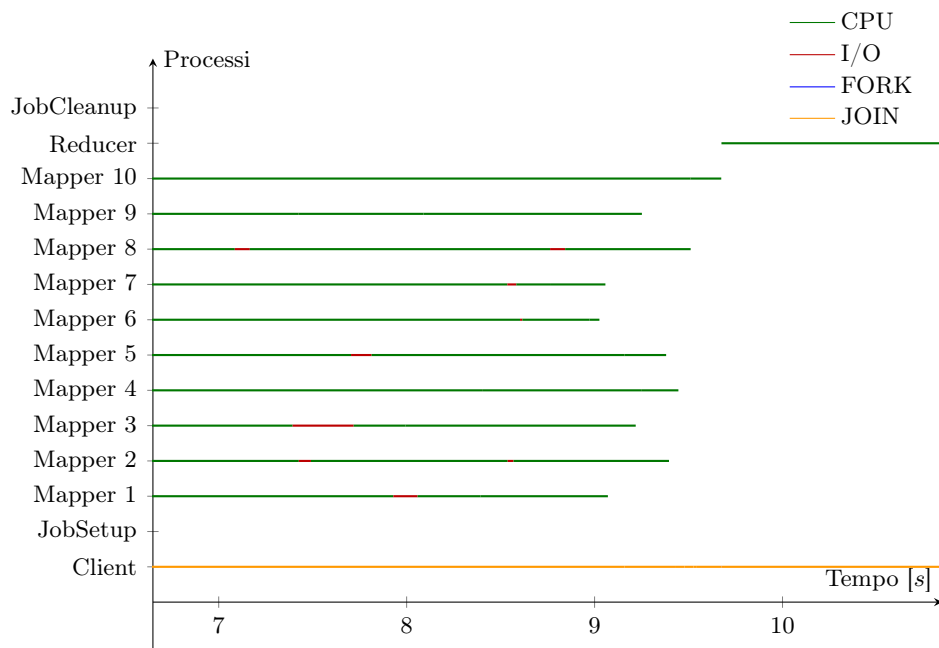


(b) Simulazione tramite benchmark.

**Figura 17:** Confronto tra i *profili temporali dell'esecuzione* di *pi* ricostruiti e simulati, durante la fase iniziale del workload.

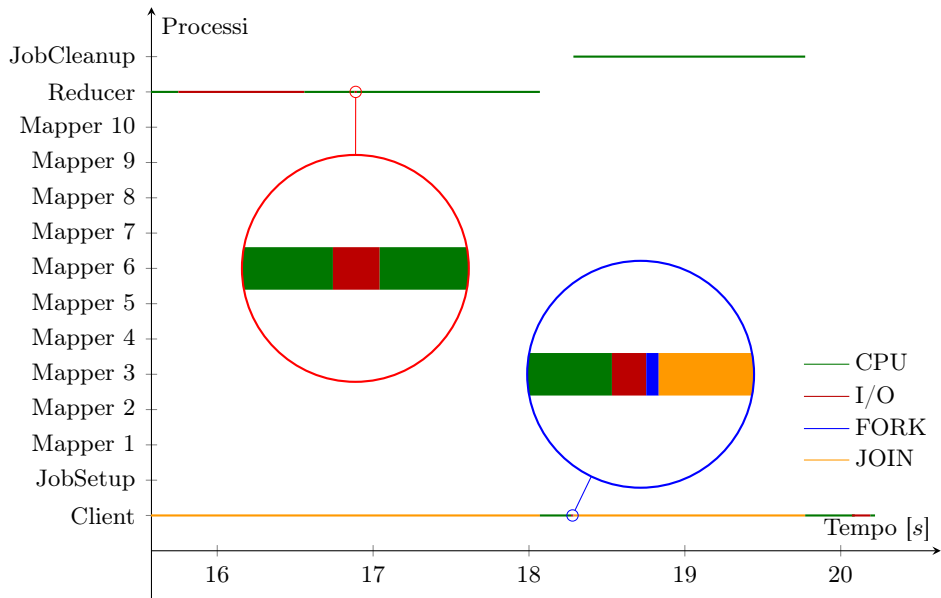


(a) Ricostruzione dell'esecuzione.

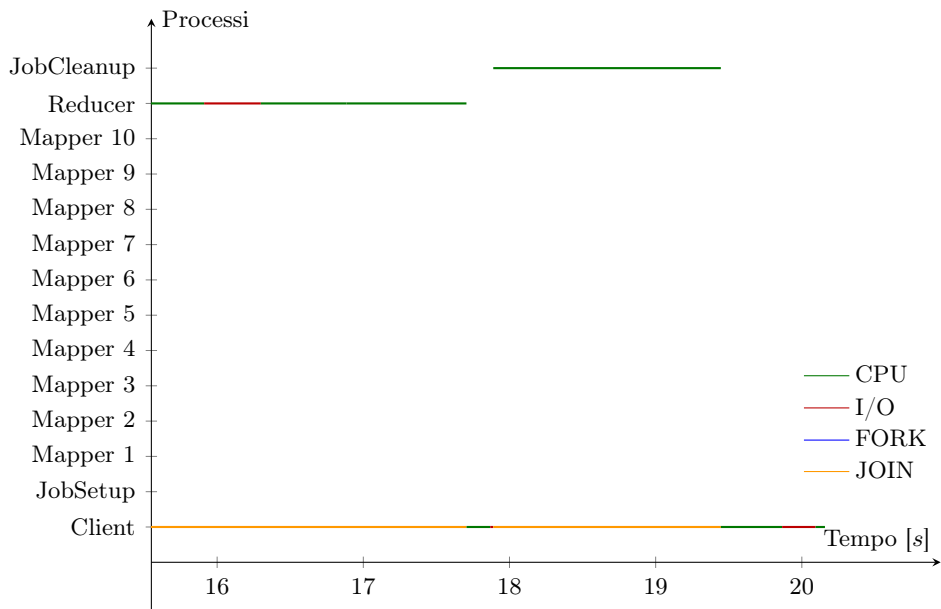


(b) Simulazione tramite benchmark.

**Figura 18:** Confronto tra i *profili temporali dell'esecuzione* di *pi* ricostruiti e simulati, durante la fase centrale del workload.



(a) Ricostruzione dell'esecuzione.



(b) Simulazione tramite benchmark.

**Figura 19:** Confronto tra i *profili temporali dell'esecuzione* di *pi* ricostruiti e simulati, durante la fase finale del workload.

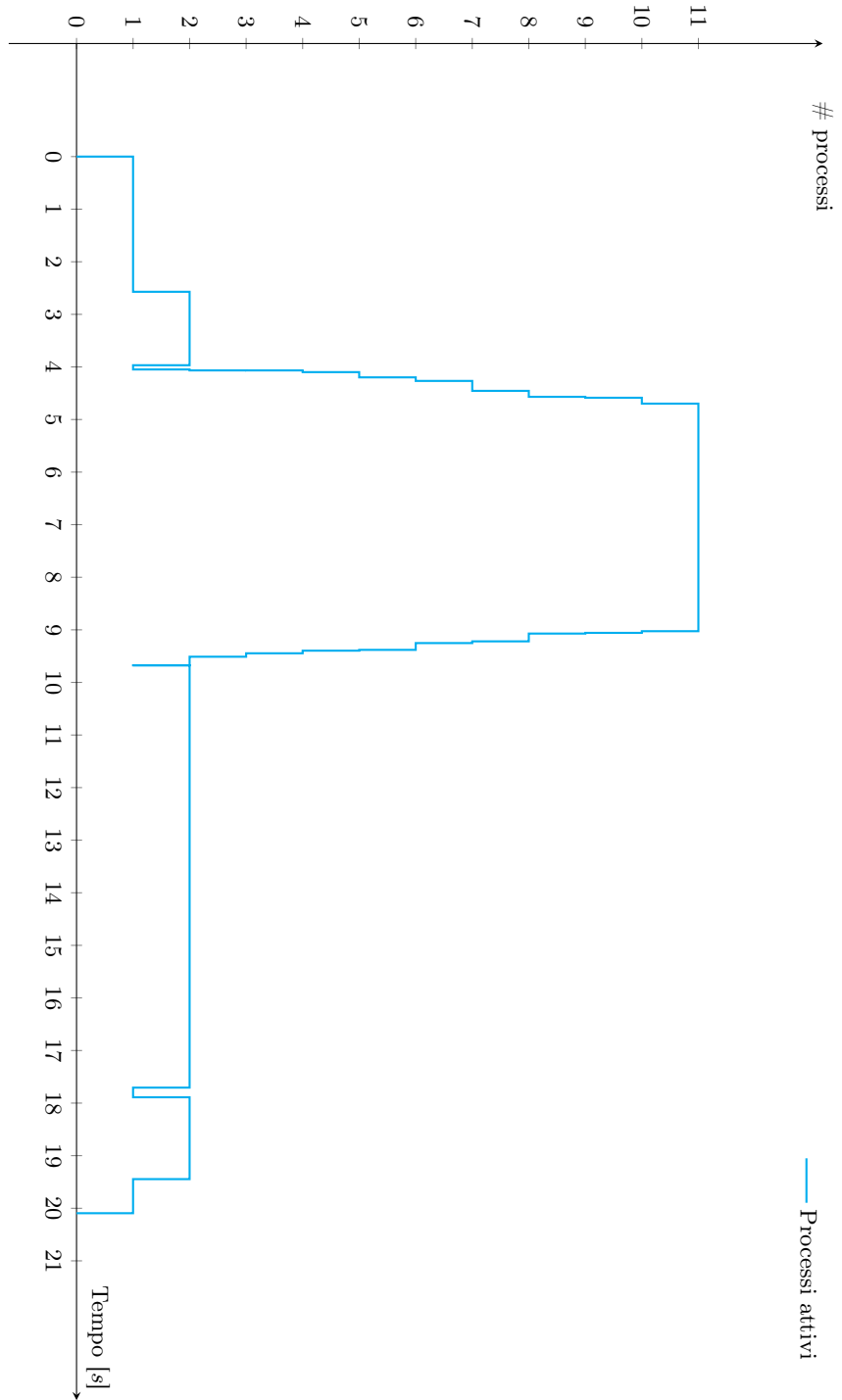
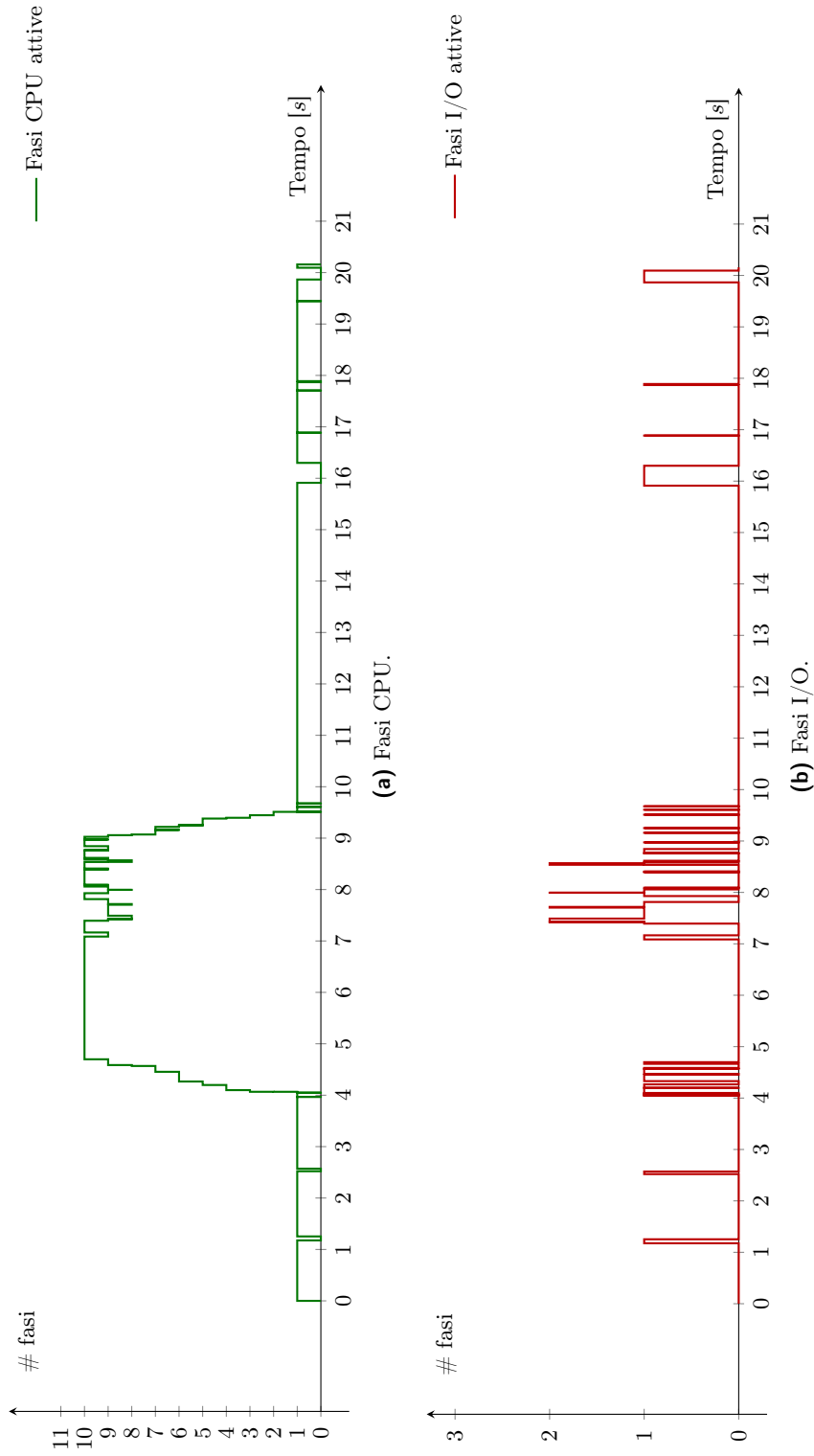
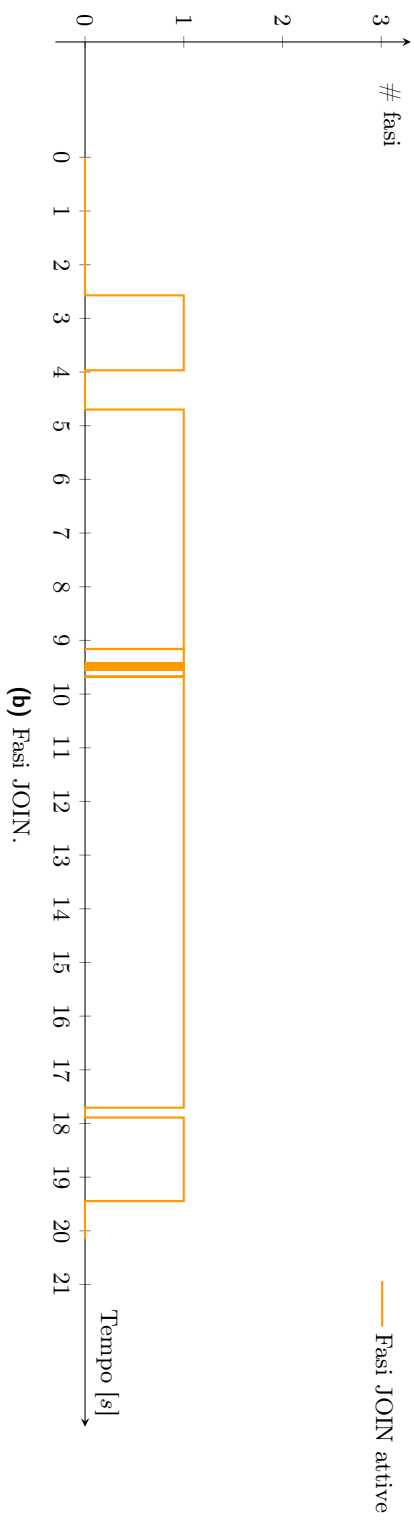
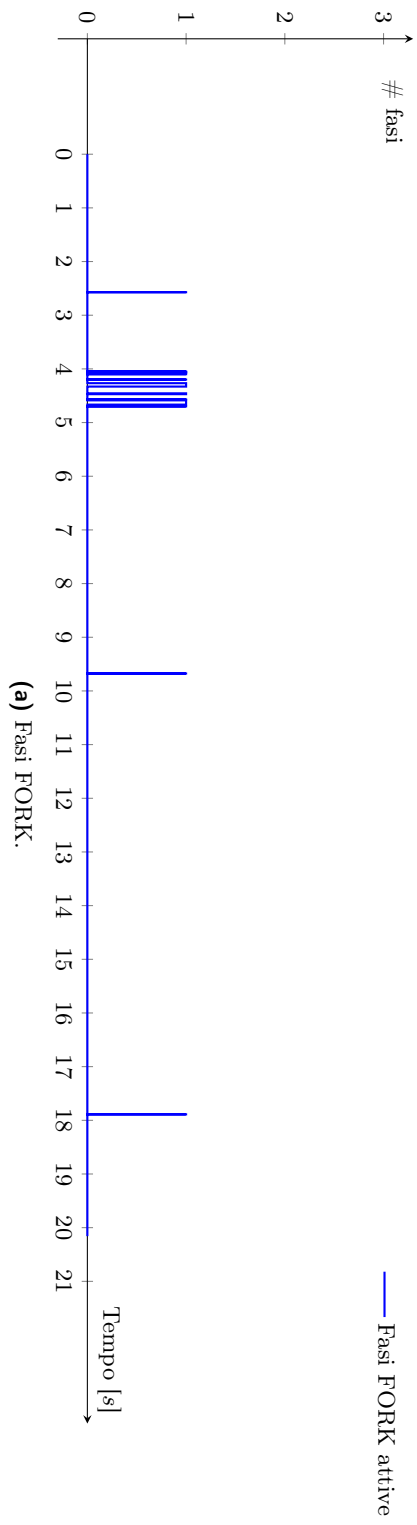


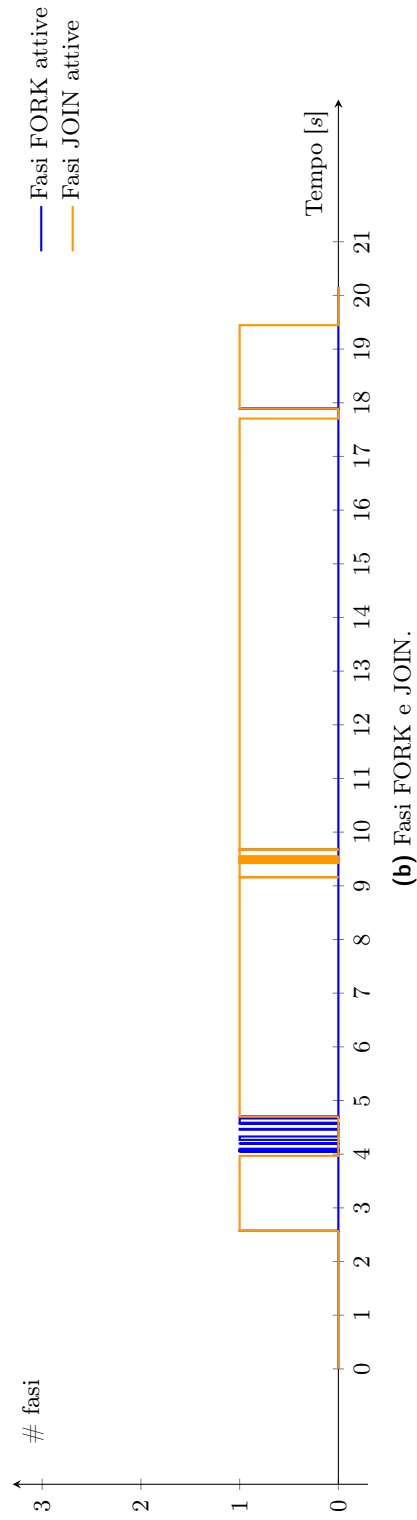
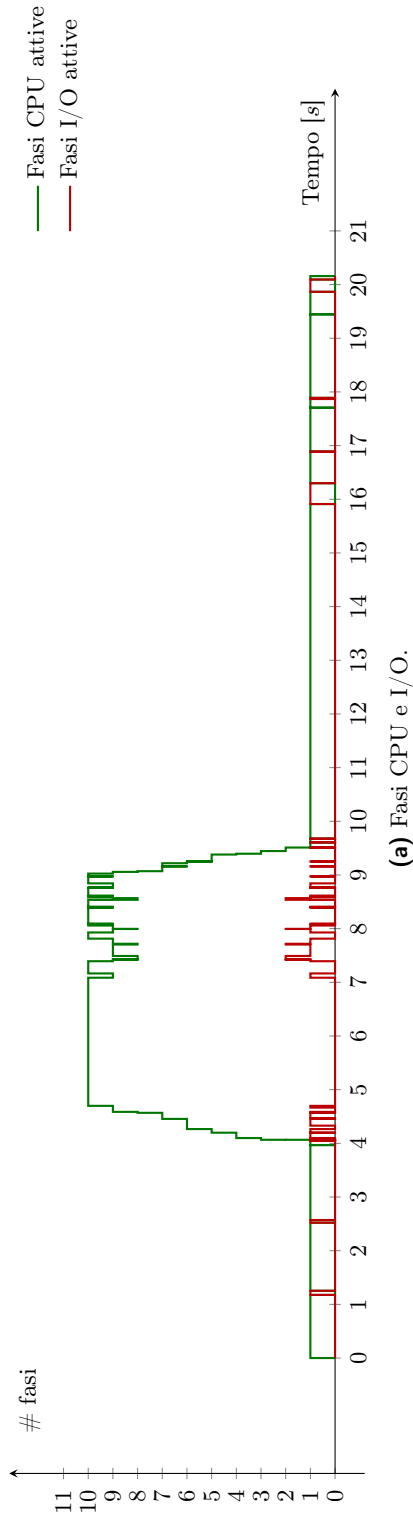
Figura 20: Profilo temporale dei processi della simulazione di *pv*.



**Figura 21:** *Profili temporali delle fasi CPU e I/O della simulazione di  $pi$ .*



**Figura 22:** *Profili temporali delle fasi FORK e JOIN della simulazione di  $p_i$ .*



**Figura 23:** *Profili temporali delle fasi di ogni tipo relativi alla simulazione di pi.*





## Conclusioni

In questo documento sono stati descritti la struttura e il funzionamento di un benchmark innovativo, caratterizzato dalla possibilità di definire flessibilmente il workload da generare e, in seguito, di analizzarlo con metriche non limitate al semplice tempo di risposta.

L'obiettivo di un tale strumento è la possibilità di modellare al meglio un'applicazione arbitraria, grazie ad una precisa descrizione del traffico voluto nel sistema, aumentando in questo modo l'affidabilità delle analisi seguenti.

Queste sono utilizzabili dall'utente per ottimizzare la struttura di un algoritmo in fase di sviluppo, individuando la migliore combinazione di parallelismo e operazioni a disposizione o per confrontare le prestazioni di sistemi eterogenei a parità di workload.

La trattazione è accompagnata da una panoramica sullo stato dell'arte di benchmark, sistemi di elaborazione moderni e nozioni di base, necessarie alla piena comprensione dello strumento realizzato. L'introduzione al framework Apache Hadoop, al modello MapReduce e il loro uso durante la validazione sperimentale del benchmark completano l'elaborato.

### Limiti e sviluppi futuri

Il prototipo sviluppato rappresenta una prima versione funzionante di quanto desiderato, la cui usabilità è tuttavia limitata dalla necessità di codificare direttamente in memoria, in linguaggio *C*, le strutture dati necessarie, e dalla mancanza di una specifica *temporale* della durata delle fasi, descritta attualmente tramite *numero di iterazioni*.

Gli ambiziosi obiettivi che questo lavoro vuole raggiungere richiedono studi, risorse e attenzioni che non possono trovare posto in una tesi di Laurea Magistrale. Il benchmark è stato concepito come primo stadio di un progetto da espandere, con numerose ipotesi di sviluppi futuri che migliorino sempre di più la sua potenza ed efficienza.

Di seguito si illustrano alcune idee che possono essere implementate in versioni successive dello strumento:

- La definizione di un *linguaggio formale*, ad esempio una *grammatica*, che permetta di specificare esattamente e in un modo il più possibile standardizzato il comportamento desiderato di un'applicazione.

- La costruzione di un *parser* che consenta il riconoscimento di tale grammatica, con la conseguente ricostruzione corretta delle strutture necessarie al funzionamento del benchmark. In questo modo l'utente può semplicemente concentrarsi sulla stesura della grammatica, senza preoccuparsi di dover configurare la memoria dello strumento in maniera adeguata.
- L'implementazione di una procedura di *calibrazione* in grado di trasformare una specifica *temporale* in una *iterativa*, affiancata dalla possibilità di definire un carico di lavoro tramite l'*intensità* delle sue operazioni, oltre che dalla loro durata.
- La possibilità di abilitare, disabilitare e configurare i meccanismi di ottimizzazione introdotti dal sistema operativo, quali la page cache, la read-ahead o l'asincronismo delle scritture, confrontando le prestazioni in loro presenza o assenza e misurando il loro impatto sulla macchina.
- La possibilità di impostare un certo livello di *casualità* nelle operazioni da simulare, senza limitarsi al comportamento *determinato* degli eventi. Ad esempio vi sono avvenimenti modellabili in modo molto efficiente tramite l'uso di *distribuzioni statistiche*, approccio largamente utilizzato che semplifica l'elaborazione dei risultati. Può essere interessante permettere la simulazione e l'analisi di un sistema tenendo conto di questi modelli, consentendone un utilizzo avanzato ed efficiente.
- Il potenziamento degli indici di prestazione calcolabili dal programma. Applicazioni e processi in esecuzione contemporanea possono influire sulle analisi raccolte, rendendole non rappresentative del normale traffico caratterizzante l'ambiente sotto osservazione. La ripetizione della simulazione del traffico in più istanze e l'introduzione degli *intervalli di confidenza* possono migliorare sensibilmente i risultati finali, aumentando la loro affidabilità e consentendo la definizione di nuove metriche.
- L'estensione del taskgraph, così da consentire una definizione più approfondita del comportamento di ogni fase. Ad esempio, può essere interessante individuare *più tipi* di fasi CPU-bound o I/O-bound, ognuna caratterizzata da un diverso pattern di operazioni. Sarebbe così possibile aumentare l'adeguatezza del benchmark all'operazione da modellare, migliorando l'attendibilità delle analisi.
- L'introduzione di meccanismi di *sincronizzazione* tra processi, in modo da poter studiare il loro impatto sulle prestazioni del sistema. Questo può includere la definizione di strumenti espliciti (quali *barriere*, attese, semafori o *file lock*) o impliciti (come operazioni di lettura o scrittura sulle stesse risorse).

- La rilevazione del *collo di bottiglia* del sistema, così da individuare la risorsa o il meccanismo che più penalizza le prestazioni.
- L'aggiunta di modelli e interazioni distribuite, come la generazione o la ricezione di messaggi remoti, la definizione di processi demoni o l'attesa e l'invio di dati tramite una rete.



## Lista degli acronimi

**SPEC** Standard Performance Evaluation Corporation.....8

Consorzio no-profit formato nel 1988 che si occupa di creare e mantenere una serie di benchmark di riferimento. Lo scopo delle sue attività è la formazione di uno standard nella misura delle performance.

**ILP** Instruction Level Parallelism ..... 18

Termine indicante tutte le tecniche architetturali che consentono l'esecuzione contemporanea di più istruzioni nella stessa unità di calcolo, senza replicare le CPU a disposizione.

**TLP** Thread Level Parallelism ..... 21

Termine indicante tutte le tecniche architetturali che consentono di definire flussi di istruzioni logicamente indipendenti tra loro, chiamati *thread hardware*, garantendo l'assenza di conflitti tra istruzioni di thread diversi. Anche se possono introdurre una lieve replicazione delle risorse computazionali, tutte le tecniche di questo tipo mantengono comunque una singola CPU.

**SMT** Simultaneous MultiThreading ..... 22

Tecnica appartenente al paradigma TLP che consente di eseguire, in ogni momento e in ogni stadio della pipeline, istruzioni appartenenti a thread hardware arbitrari.

**GPU** Graphics Processing Unit.....24

Nome comunemente utilizzato per indicare un processore ottimizzato per l'elaborazione grafica.

**NIST** National Institute of Standards and Technology.....32

Organizzazione statunitense preposta alla promozione della tecnologia, delle misure e degli standard industriali.

<b>IaaS</b>	Infrastructure as a Service . . . . .	35
	Termine utilizzato per indicare i servizi che, tramite <i>cloud computing</i> , forniscono al richiedente una completa VM configurata opportunamente, virtualmente corrispondente ad una macchina fisica con accesso completo.	
<b>PaaS</b>	Platform as a Service . . . . .	35
	Termine utilizzato per indicare i servizi che, tramite <i>cloud computing</i> , forniscono al richiedente un ambiente dalla limitata configurabilità, preposto per lo sviluppo e l'utilizzo di applicazioni specifiche.	
<b>SaaS</b>	Software as a Service . . . . .	35
	Termine utilizzato per indicare i servizi che, tramite <i>cloud computing</i> , forniscono al richiedente un'applicazione specifica, dall'uso fortemente limitato e poco configurabile.	
<b>EC2</b>	Elastic Compute Cloud . . . . .	37
	Prodotto IaaS di Amazon che consente l'acquisto e la piena gestione di VM configurate in base alle proprie necessità.	
<b>S3</b>	Simple Storage Service . . . . .	37
	Prodotto di Amazon che mette a disposizione, tramite <i>cloud computing</i> , spazio di archiviazione personale.	
<b>VPN</b>	Virtual Private Network . . . . .	38
	Tecnica di virtualizzazione che consente di instaurare una rete locale utilizzando l'infrastruttura di una rete pubblica.	
<b>VM</b>	Virtual Machine . . . . .	39
	Ambiente virtuale che emula le caratteristiche e la composizione di una macchina fisica, consentendo l'esecuzione di sistemi operativi e applicazioni.	
<b>IEEE</b>	Institute of Electrical and Electronics Engineers . . . . .	42
	Famosa e ampia organizzazione di scienziati professionisti impegnata nella promozione delle scienze tecnologiche.	

---

<b>POSIX</b>	Portable Operating System Interface for UniX . . . . .	43
	Termine che indica una famiglia di standard promossi dalla IEEE e specificanti i meccanismi di compatibilità basilari tra sistemi operativi. L'acronimo è spesso preferito alla denominazione formale della famiglia (IEEE 1003).	
<b>TSC</b>	Time Stamp Counter . . . . .	54
	Registro a 64 bit incluso in numerose architetture contemporanee e preposto a memorizzare il numero di impulsi generati dal clock di sistema dall'accensione della macchina o dall'ultimo riavvio.	
<b>HPET</b>	High Precision Event Timer . . . . .	54
	Chip incluso nelle più moderne architetture, composto da oscillatori ad alta frequenza e contatori di elevata precisione. È progettato per essere facilmente programmato dal sistema operativo, e la sua indipendenza da tutti gli altri clock presenti nella macchina consente la creazione di più <i>timer</i> personalizzati, evitando il problema del <i>time warp</i> .	
<b>HDFS</b>	Hadoop Distributed File System . . . . .	58
	Implementazione di Apache del concetto di <i>file system distribuito</i> , che consente la memorizzazione di dati in dispositivi di archiviazione distribuiti, ricreando all'utente l'illusione che esista un'unica periferica preposta allo scopo. È incluso in Hadoop.	
<b>SPOF</b>	Single Point Of Failure . . . . .	59
	Termine utilizzato in un sistema informatico per indicare la risorsa che, se messa fuori operatività, determina la caduta dell'intero sistema.	
<b>DAG</b>	Directed Acyclic Graph . . . . .	75
	Grafo con archi diretti che non presenta cicli, ovvero non è possibile, partendo da un qualunque vertice del grafo, tornare ad esso tramite un percorso descritto dai suoi archi.	





## Bibliografia

- [1] GENE M. AMDAHL. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485.
- [2] CHRISTER BENGTTSSON et al. *Multicore computing—the state of the art*. A cura di KARL-FILIP FAXÉN. Non pubblicato. 2008.
- [3] STEPHEN M. BLACKBURN et al. “The DaCapo benchmarks: java benchmarking development and analysis”. In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pp. 169–190.
- [4] DANIEL P. BOVET e MARCO CESATI. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly Media, 2005.
- [5] DACAPO PROJECT. *DaCapo Benchmarks*. 2012. URL: <http://www.dacapobench.org/>.
- [6] JEFFREY DEAN e SANJAY GHEMAWAT. “MapReduce: simplified data processing on large clusters”. In: *OSDI'04: Proceedings of the 6th conference on symposium on Operating Systems Design and Implementation*. USENIX Association, 2004.
- [7] KAIVALYA M. DIXIT. “Overview of the SPEC Benchmarks.” In: *The Benchmark Handbook*. A cura di JIM GRAY. Morgan Kaufmann, 1993. ISBN: 1-55860-292-5.
- [8] KAIVALYA M. DIXIT, JEFF REILLY e JOHN L. HENNING. *SPEC CPU2006: Read Me First*. SPEC. 2011. URL: <http://www.spec.org/cpu2006/Docs/readme1st.html>.
- [9] SUSAN J. EGGERS et al. “Simultaneous multithreading: a platform for next-generation processors”. In: *Micro, IEEE* 17.5 (1997), pp. 12–19. ISSN: 0272-1732.
- [10] IAN FOSTER e CARL KESSELMAN, cur. *The grid: blueprint for a new computing infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1-55860-475-8.

- [11] IAN FOSTER et al. “Cloud Computing and Grid Computing 360-Degree Compared”. In: *Grid Computing Environments Workshop, 2008. GCE '08*. 2008, pp. 1–10.
- [12] SANJAY GHEMAWAT, HOWARD GOBIOFF e SHUN-TAK LEUNG. “The Google file system”. In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 29–43. ISBN: 1-58113-757-5.
- [13] HADOOP WIKI. *How Map and Reduce operations are actually carried out*. 2013. URL: <http://wiki.apache.org/hadoop/HadoopMapReduce>.
- [14] JOHN L. HENNESSY e DAVID A. PATTERSON. *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN: 0123704901.
- [15] JOHN L. HENNING. “SPEC CPU suite growth: an historical perspective”. In: *SIGARCH Comput. Archit. News* 35.1 (2007), pp. 65–68.
- [16] JOHN L. HENNING. “SPEC CPU2000: measuring CPU performance in the New Millennium”. In: *Computer* 33.7 (2000), pp. 28–35. ISSN: 0018-9162.
- [17] “SPEC CPU2006 benchmark descriptions”. In: *SIGARCH Comput. Archit. News* 34.4 (2006). A cura di JOHN L. HENNING, pp. 1–17.
- [18] MAHESHKUMAR P JAGTAP. “Era of Multi-Core Processors”. In: *DRDO Science Spectrum* (mar. 2009), pp. 87–94.
- [19] MANUALE di LINUX. *clone(2)*. 2013. URL: <http://linux.die.net/man/2/clone>.
- [20] MANUALE di LINUX. *waitpid(3)*. 2013. URL: <http://linux.die.net/man/3/waitpid>.
- [21] MANUALE di LINUX. *open(3)*. 2013. URL: <http://linux.die.net/man/3/open>.
- [22] MANUALE di LINUX. *pread(2)*. 2013. URL: <http://linux.die.net/man/2/pread>.
- [23] MANUALE di LINUX. *clock\_gettime(3)*. 2013. URL: [http://linux.die.net/man/3/clock\\_gettime](http://linux.die.net/man/3/clock_gettime).
- [24] MAKOTO MATSUMOTO e TAKUJI NISHIMURA. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”. In: *ACM Trans. Model. Comput. Simul.* 8.1 (gen. 1998), pp. 3–30. ISSN: 1049-3301.
- [25] PETER MELL e TIM GRANCE. *The NIST Definition of Cloud Computing*. Rapp. tecn. 2009. URL: <http://www.csrc.nist.gov/groups/SNS/cloud-computing/>.

- 
- [26] ROBERT MUNAFO. *The SPEC Benchmarks at MROB*. 2013. URL: <http://mrob.com/pub/comp/benchmarks/spec.html>.
- [27] OWEN O'MALLEY. *Introduction to Hadoop*. Slide di presentazione. ApacheCon: Yahoo!, Inc., 2008.
- [28] *Slide del corso "Architetture Avanzate dei Calcolatori"*. L'immagine è stata adattata dall'autore della tesi per esigenze grafiche. Politecnico di Milano, 2011.
- [29] *Slide del corso "Enterprise Digital Infrastructure"*. Politecnico di Milano, 2012.
- [30] SPEC. *Standard Performance Evaluation Corporation*. 2013. URL: <http://www.spec.org/>.
- [31] SYS-CON MEDIA INC. *Twenty-One Experts Define Cloud Computing*. 2008. URL: <http://www.sys-con.com/node/612375/print>.
- [32] THE APACHE SOFTWARE FOUNDATION. *Apache Hadoop*. 2013. URL: <http://hadoop.apache.org>.
- [33] THE APACHE SOFTWARE FOUNDATION. *HDFS Architecture Guide*. 2013. URL: [http://hadoop.apache.org/docs/stable/hdfs\\_design.html](http://hadoop.apache.org/docs/stable/hdfs_design.html).
- [34] THE APACHE SOFTWARE FOUNDATION. *MapReduce Tutorial*. 2013. URL: [http://hadoop.apache.org/docs/stable/mapred\\_tutorial.html](http://hadoop.apache.org/docs/stable/mapred_tutorial.html).
- [35] DEAN M. TULLSEN, SUSAN J. EGGERS e HENRY M. LEVY. "Simultaneous multithreading: Maximizing on-chip parallelism". In: *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*. 1995, pp. 392–403.
- [36] TOM WHEELER. *Hadoop In 45 Minutes or Less*. Slide di presentazione. Lambda Lounge (St. Louis, USA): Object Computing, Inc., 2009.
- [37] TOM WHITE. *Understanding MapReduce with Hadoop*. Slide di presentazione. SPA 2008.
- [38] WIKIPEDIA. *Instruction pipeline*. 2013. URL: [http://en.wikipedia.org/wiki/Instruction\\_pipeline](http://en.wikipedia.org/wiki/Instruction_pipeline).
- [39] WIKIPEDIA. *Supercomputer*. 2013. URL: <http://en.wikipedia.org/wiki/Supercomputer>.
- [40] DONG YE, JOYDEEP RAY e DAVID KAELI. "Characterization of file I/O activity for SPEC CPU2006". In: *SIGARCH Comput. Archit. News* 35.1 (2007), pp. 112–117. ISSN: 0163-5964.
- [41] QI ZHANG, LU CHENG e RAOUF BOUTABA. "Cloud computing: state-of-the-art and research challenges". In: *Journal of Internet Services and Applications* 1.1 (2010), pp. 7–18. ISSN: 1867-4828.



## Indice analitico

- EC2, 37, 38, 40
  - HDFS, 58–61, 64–66, 109, 112
  - HPET, 54
  - ILP, 18
  - IaaS, 35
  - PaaS, 35
  - S3, 37
  - SMT, 22
  - SaaS, 35
  - TLP, 21
  - TSC, 54
  - VM, 39
  
  - analizzatore di log, 75, 93
  
  - bench-a-thon, 8
  - benchmark, 5
    - punteggio, 6
    - suite di -, 7
  
  - cache, 25
    - hit, 25
    - miss, 25
    - gerarchia, 26
  - calibrazione, 73
  - Central Processing Unit, *vedi* CPU
  - client, 64
    - thick -, 36
    - thin -, 36
  - clock
    - di sistema, 17, 54
    - cicli di -, 17
  - cloud, 34
    - computing, 32
  - cluster, 29
  - core, 17
  - CPU, 17
  
  - DaCapo, 14
    - DaCapo-9.12-bach, 14
  - data center, 36
  - DataNode, 59–62, 64–66, 110, 111, 114, 116
  - demone, 110
  - dispositivi a blocchi, 48
  - distribuzione uniforme, 68
  
  - Elastic Compute Cloud, *vedi* EC2
  
  - fase, 76
    - CPU-bound, 71, 74, 76, 79, 83, 84, 89–91, 112, 114, 116, 118, 136
    - FORK, 71, 76, 78, 79, 87–90, 93, 99, 110, 111, 114, 116, 122
    - I/O-bound, 71, 73, 74, 76, 79, 83–85, 89–91, 110, 112, 114, 116, 118, 136
    - JOIN, 72, 76, 78, 79, 88–90, 93, 99, 110, 116
    - LOOP, 71, 76, 77, 79, 86, 87, 89, 90
  
  - file, 48
    - di log (Hadoop), 66, 111, 113
    - di log (benchmark), 75, 81, 93
    - di statistiche testuali, 75, 94
  - apertura, 48
    - modalità diretta, 51
    - modalità sincrona, 52, 73
  - descrittore, 48
  - lettura, 49
  - scrittura, 51
- file system distribuito, 36, 38, 58
- fitting*, 118
- funzione

- di map, *vedi* map
- di reduce, *vedi* reduce
- clock\_gettime, 54–56, 93
- clone, 43, 46, 47, 81, 88
- computeTime, 55
- executeCPUPhase, 83
- executeFORKPhase, 87–89
- executeIOPhase, 84
- executeJOINPhase, 89
- executeLOOPPhase, 87
- fork, 42, 43, 47
- getpid, 91
- open, 49
- pread, 49–51, 86
- pwrite, 51, 52, 86
- sched\_getcpu, 93
- setForkJoinEndTimestamp, 87, 89, 90, 93
- setForkJoinStartTimestamp, 90, 91
- setOrdinaryEndTimestamp, 90, 93
- setOrdinaryStartTimestamp, 90, 91
- simulate, 81, 82, 87, 88, 90, 91, 93
- waitpid, 47, 48, 88, 89
- writeLogFile, 83, 91, 93
- generatore
  - di numeri pseudo-casuali, 68
  - periodo, 69
  - di workload, 74, 81
- grado di parallelismo, 24
- grid, 30
- guest, 39
- Hadoop, 57, 58, 107, 108, 110, 111, 113, 114, 135
- Hadoop Distributed File System, *vedi* HDFS
- Heartbeat, 60
- High Precision Event Timer, *vedi* HPET
- host, 39
- hypervisor, 39
- idle*, 42
- indici di prestazione, 5
- Infrastructure as a Service, *vedi* IaaS
- input di riferimento, 12
- Instruction Level Parallelism, *vedi* ILP
- istanza, 39
- job, 64
- JobCleanup, 66, 114, 115
- JobSetup, 66, 114, 115
- JobTracker, 61, 64–66, 109–112
- latenza, 17
- macchina di riferimento, 12
- manycore, 24
- map, 61–65
- mapper, 63–66, 109–116, 121
- MapReduce, 36, 58, 61–64, 66, 67, 107–112, 114, 115, 135
- matrici, 67
  - prodotto, 68
- media geometrica, 13
- Mersenne Twister, 69, 84
- metrica, 6
- microprocessore, 17
- migrazione, 39
- multithreading
  - coarse-grained -, 21
  - fine-grained -, 21
  - simultaneous -, *vedi* SMT
- NameNode, 59–61, 64–66, 110, 111
- page cache, 50, 73
- pagina (dispositivi a blocchi), 48
- pi*, 108, 109, 113, 114, 116, 121
- pipeline, 18
  - conflitti, 19
- pipelining, 18
- Platform as a Service, *vedi* PaaS
- processo, 42
  - figlio, 42
  - light-weight, 43, 88
  - padre, 42

- 
- priorità, 45
  - real-time, 46
  - stato, 43
    - blocked*, 44
    - diagramma (degli stati), 43
    - ready*, 44
    - running*, 44
  - processore, 17
  - profilo temporale, 75
    - dei processi, 99
    - dell'esecuzione, 97
    - delle fasi, 101
  - ps*, 113
  - pthread*, 42
  - rack, 36
  - read-ahead, 50, 73
  - reduce, 61–66
  - reducer, 63–67, 110–116
  - replicazione, 60
    - fattore di-, 60
  - scheduler, 45
  - seed, 69
  - sequenza
    - di numeri casuali, 68
    - di numeri pseudo-casuali, 68
  - settore, 48
  - Simple Storage Service, *vedi* S3
  - Simultaneous MultiThreading, *vedi* SMT
  - Single System Image, 30
  - sistemi
    - distribuiti, 29
    - multicore, 23
    - multiprocessore, 27
    - paralleli, 16
    - sequenziali, 16
  - sleep*, 112
  - snapshot, 39
  - Software as a Service, *vedi* SaaS
  - SPEC, 8
    - CPU2006
      - SPECfp\_base2006, 13
      - SPECfp\_rate2006, 13
      - SPECfp\_rate\_base2006, 13
      - SPECint2006, 13
      - SPECint\_base2006, 13
      - SPECint\_rate2006, 13
      - SPECint\_rate\_base2006, 13
      - SPECrate, 13
      - SPECratio, 12
    - CFP2006, 10
    - CINT2006, 10
    - CPUv6, 14
  - struttura
    - ForkPhase, 78
    - JoinPhase, 78, 88, 89
    - LoopPhase, 77, 79, 87
    - OrdinaryPhase, 76, 77, 83
    - ProcessInterval, 100, 101, 103
    - Statistics, 83, 89–91, 93–95, 99, 100, 103
    - timespec, 55, 90
  - supercomputer, 30
    - virtuale, 32
  - task, 64, 111, 112
  - taskgraph, 74–76
  - TaskTracker, 61, 62, 64–66, 110, 111, 114, 116
  - tempo
    - di riferimento, 12
    - di risposta, 6
    - quanto di -, 45
  - thread, 42
    - hardware, 21
    - switching, 21
  - Thread Level Parallelism, *vedi* TLP
  - throughput, 6
  - Time Stamp Counter, *vedi* TSC
  - time warp*, 54
  - timer software, 57
  - uptime, 54, 56, 83, 90, 93
  - utility computing, 33
  - variabile

id, 76, 77, 79, 83, 90, 91, 118  
pid, 43, 47, 87–91, 93  
Virtual Machine, *vedi* VM  
virtual organization, 31  
virtualizzazione, 34, 38  
    hardware, 39  
workload, 6