UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

ANALISI EMPIRICA DI DUAL PIVOT QUICKSORT

EMPIRICAL ANALYSIS OF DUAL PIVOT QUICKSORT

MARCO VIGNINI

Relatore: *Prof. Maria Cecilia Verri*

Anno Accademico 2015-2016

Marco Vignini: *Analisi Empirica di Dual Pivot Quicksort*, Corso di Laurea in Informatica, © Anno Accademico 2015-2016

INDICE

ELENCO DELLE FIGURE

"L'uomo rimane il più straordinario dei computer"
— John Fitzgerald Kennedy

1

INTRODUZIONE AL PROBLEMA DELL'ORDINAMENTO

L'ordinamento dei dati è uno dei fondamentali problemi dell'informatica. Spesso viene riscontrato anche come sotto problema contenuto in problemi molto più complessi e deve essere risolto più efficientemente possibile. Un modello ricorrente utilizzato per ordinare degli elementi è quello basato sul confronto tra oggetti, che è utilizzato nel corso di tutta la tesi. Definiamo un algoritmo di ordinamento come una procedura che viene utilizzata per elencare gli elementi di un insieme secondo una sequenza stabilita da una relazione d'ordine, in modo che ogni elemento sia minore o uguale (oppure maggiore o uguale) rispetto a quello che segue [16]. Assumiamo per tutta la tesi come riferimento un modello che utilizza confronti e numeri interi. L'algoritmo più utilizzato al mondo per risolvere il problema dell'ordinamento è Quicksort [21] [22]. Tale algoritmo fu sviluppato nel 1959 da [20] Tony Hoare ma venne pubblicato solamente nel 1962. Nel corso degli anni è stato uno degli algoritmi più apprezzati per risolvere il problema dell'ordinamento, ed è stato oggetto di miglioramento da parte di vari studiosi che lo hanno portato all'implementazione con la quale il mondo dell'informatica lo conosce oggi. Ma nel 2009 uno studioso russo di nome Vladimir Yaroslavskiy [15] ha proposto una nuova versione dell'algoritmo, sostenendo in particolare una maggiore velocità nel risolvere il problema dell'ordinamento rispetto alla precedente che godeva delle modifiche apportate da Jon Bentley e da Mcllroy del 1975 [21]. Verso la metà degli anni '70 Sedgewick aveva già preso in considerazione una versione di Quicksort a due perni ma l'algoritmo che produsse non risultò comunque più veloce di Quicksort Classico [28].

Andiamo a definire qualche concetto che sarà utilizzato nel corso della tesi. Verrà fatto riferimento ad un ordinamento generico senza specificare

se crescente o decrescente dato che le considerazioni fatte per un ordine valgono simmetricamente anche per l'altro. Definiamo un algoritmo stabile se conserva l'ordine relativo di chiavi uguali (dove per chiavi si intendono numeri interi) [16]. Un algoritmo d'ordinamento lavora sul posto se non usa memoria ausiliaria che supporti l'esecuzione di tale algoritmo permettendogli per esempio di memorizzare temporaneamente alcune chiavi per determinati scopi [22]. Quindi in ogni istante è allocato al massimo un numero costante di variabili oltre al vettore contenente le chiavi da ordinare. Definiamo una sentinella come l'elemento maggiore o minore della collezione di oggetti da ordinare (rispettivamente indicati con -∞ e +∞) [24], utilizzata allo scopo di evitare controlli ridondanti e comunque di garantire la correttezza dell'algoritmo. *chiavi* è la struttura concreta che rappresenta la collezione di elementi da ordinare, la cui denominazione è la stessa scelta nel codice implementato. In riferimento agli insiemi di chiavi da ordinare l'indice `left` sarà sempre riferito all'estremo sinistro di essi e `right` all'estremo destro.

La tesi si pone come obiettivo quello di confrontare da un punto di vista empirico l'algoritmo Quicksort nella sua versione Classica e con scelta del perno come mediano fra tre chiavi candidate, con la versione a doppio perno di Yaroslavskiy.

TONY HOARE

Tony Hoare [27] è un informatico britannico nato a Colombo (Sri Lanka) nel 1934 da genitori inglesi a cui si deve l'invenzione dell'algoritmo Quicksort. Hoare sviluppò Quicksort nel 1959 mentre era in Unione Sovietica, più precisamente alla Moscow State University, dove stava lavorando ad un progetto in cui era richiesto di ordinare delle parole: dopo aver realizzato che Insertion Sort sarebbe stato troppo lento egli inventò Quicksort. La pubblicazione dell'algoritmo risale però al 1962. Hoare si era laureato in lettere nel 1956 ma successivamente ha studiato ad Oxford statistica avanzata. Ha iniziato a lavorare allo sviluppo di algoritmi solo nel 1960 quando venne assunto dalla Elliot Brothers dove inventò il linguaggio di programmazione ALGOL 60. Oggi Tony Hoare è professore all'università di Oxford e ricercatore per conto di Microsoft a Londra.

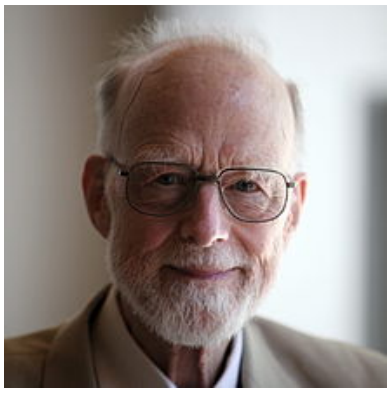

Figura 1.: Tony Hoare

2

QUICKSORT

L'algoritmo Quicksort si basa sul principio del Divide et Impera [17] e sul processo di partizionamento. Il principio del Divide et Impera risale circa al 359 a.C quando Filippo II di Macedonia salì al potere; tale principio afferma che:

> *Il miglior espediente per governare di una tirannide o di un'autorità qualsiasi per controllare e governare un popolo è dividerlo, provocando rivalità e fomentando discordie.*

Tale criterio è stato applicato anche nel mondo degli algoritmi, quindi data l'istanza di un problema il Divide et Impera suggerisce di suddividerla in k sotto istanze (con $k \in R$) allo scopo di avere più sotto istanze del solito problema ma con dimensioni ridotte. In realtà un algoritmo che usa Divide et Impera dovrebbe dividere esattamente a metà l'istanza del problema, Quicksort non lo fa ma è comunque considerato appartenente alla famiglia degli algoritmi che si basano su questo principio. Quindi le sotto istanze vanno risolte e le soluzioni ottenute ricombinate, così da ottenere la soluzione per l'istanza di partenza. Se le sotto istanze ottenute dal partizionamento di quella di partenza risultano comunque relativamente troppo grandi allora tale procedimento può essere ripetuto generando sotto istanze sempre più piccole. Tale procedimento continua fino a che le sotto istanze ottenute non risultano abbastanza piccole da poter essere risolte senza partizionare ulteriormente quelle ottenute al passo precedente. La procedura di partizionamento, per applicare il principio del Divide et Impera seleziona una delle chiavi da ordinare, detta perno, e in base al suo valore crea sotto istanze del problema, i cui dati vengono ordinati in modo indipendente. Tale partizionamento si basa sul valore dei dati e non sulla loro quantità. Il partizionamento ricorsivamente suddivide tutti i dati dell'istanza ottenuta al passo precedente in due sotto istanze: nel primo insieme vengono collocati gli elementi minori del perno e nel secondo quelli maggiori. Dopo questo processo l'elemento

perno risulta nella sua posizione finale, quindi il partizionamento viene richiamato ricorsivamente sulle due sotto istanze da ordinare generate dal partizionamento stesso al passo precedente.

PARTIZIONAMENTO USATO DA QUICKSORT

Coerentemente con quanto detto in precedenza sul principio del Divide et Impera, andiamo a ricordare come Quicksort applichi questo principio, ovvero come partiziona l'istanza composta dalle chiavi da ordinare.

- Viene selezionato l'elemento perno che in realtà può essere scelto con più di un criterio, l'implementazione più semplice prevede di scegliere la chiave più a destra (o più a sinistra) della collezione. Soluzioni più raffinate consistono nel selezionare l'elemento perno come mediano fra tre chiavi candidate, oppure casualmente. Per spiegare il partizionamento di Quicksort assumeremo che il perno sia scelto come elemento più a sinistra dell'istanza (salvo dove diversamente specificato).
- Viene scansionato *chiavi*, partendo da sinistra con un indice `i`, finché non viene trovata una chiave maggiore del perno.
- Viene scansionato *chiavi*, partendo da destra con un indice `j`, finché non viene trovata una chiave minore del perno.
- Vengono scambiate le due chiavi su cui sono state interrotte le scansioni di `i` e `j` perché risultano fuori posto rispetto alla partizione finale che si vuole ottenere.
- Le due scansioni proseguono, arrestandosi per effettuare gli scambi quando si verificano le condizioni indicate, finché i due indici non si incrociano.
- Per completare il partizionamento, la chiave perno che si trova in *chiavi[*`left`*]* deve essere inserita al posto della chiave più a destra del sotto vettore sinistro, cioè *chiavi[*`j`*]*.

Nella sua versione classica Quicksort necessita di una sentinella nella posizione finale, altrimenti non verrebbe arrestato l'incremento di `i`. Il decremento di `j` invece viene arrestato dal perno stesso che funge da sentinella. Vediamo come funziona quindi il processo di partizionamento con un esempio, supponiamo di voler ordinare la seguente collezione di chiavi:

9 3 12 3 15 27 6 18 15 9

Basandoci su quanto detto in precedenza sul partizionamento possiamo selezionare arbitrariamente il perno scegliendolo per esempio, tra le due chiavi agli estremi della collezione. Scegliamo dunque la chiave più a sinistra, ovvero 9. Ricordiamo che durante il partizionamento l'indice i viene incrementato fino a che non trova una chiave maggiore del perno, ovvero di 9 e j viene decrementato fino a che non trova una chiave minore o uguale. Quindi nel primo partizionamento i si ferma su 12 e j su 9. Avviene quindi lo scambio tra 12 e 9, stesso discorso vale per 15 e 6. A questo punto gli indici si incrociano quindi avendo preso come perno l'elemento più a sinistra nella partizione, *chiavi[*j*]* deve essere scambiato con *chiavi[*left*]*, 9 risulta dunque nella posizione finale. Otteniamo così:

6 3 9 3 9 27 15 18 15 12

L'elemento perno relativo all'ultimo partizionamento è evidenziato in rosso, a sinistra e a destra si formano due sotto istanze del problema di ordinamento iniziale coerentemente anche con quanto detto sul principio del Divine et Impera. Il procedimento continua concentrandosi sulla sotto istanza sinistra per la quale è analogo al primo passo di partizionamento:

6 3 9 3

In questo passo vengono scambiati il 9 con il 3 più a destra ed il perno (evidenziato in rosso) con l'altro 3. Nei passi successivi di partizionamento Quicksort viene richiamato ricorsivamente prima sul 3 poi sul 6 poi sul 9, ordinando così la sotto istanza sinistra che si era generata dopo il primo partizionamento. Lo stesso procedimento è applicato alla sotto istanza destra originata dopo il primo partizionamento, tale sotto istanza da ordinare è dunque:

27 15 18 15 12

Il perno (evidenziato in rosso) viene scambiato col 12 in quanto non ci sono chiavi maggiori di 27. Al partizionamento successivo viene preso come perno 12, durante questo passo l'indice j scorre fino al perno dato che non ci sono chiavi minori del perno stesso. Durante il partizionamento successivo l'indice i rimane su 18 e j sul 15 più a destra dell'istanza e tali chiavi vengono scambiate fra di loro, permettendo al 15 più a sinistra (che durante questo passo era il perno) di raggiungere la sua

posizione definitiva nella collezione. Gli ultimi due passi dell'algoritmo sono caratterizzati dal partizionamento che è richiamato su 15 e 18, di cui il primo viene ordinato nel penultimo passo ed il secondo nell'ultimo. Ricombinado quindi le sotto istanze create si ottene la risoluzione al problema dell'ordinamento iniziale.

CARATTERISITCHE PRINCIPALI DI QUICKSORT

- Generalità.
- Lavora sul posto.
- Facile da implementare.
- Mancanza di stabilità.
- Nel caso peggiore esegue n^2 operazioni per ordinare una collezione di n chiavi.
- Nel caso medio effettua circa $0.33n\ln(n)$ scambi per ordinare una collezione di n chiavi [6].
- Nel caso medio esegue circa $2.0n\ln(n)$ confronti per ordinare una collezione di n chiavi [6].
- È un algoritmo d'ordinamento adattivo, ovvero le operazioni di confronto e di scambio sono eseguite dipendentemente dai valori delle chiavi delle collezione [16].
- La fase d'implementazione richiede precisione, basta un piccolo errore nel codice per escludere alcuni casi limite.

L'algoritmo Quicksort è generico perché si comporta bene in un'ampia fascia di configurazioni che gli si presentano davanti. Quicksort viene considerato un algoritmo che lavora sul posto anche se in realtà usa una piccola pila. Spesso viene preferita la sua implementazione ricorsiva dato che quella iterativa è più complicata da realizzare. L'algoritmo Quicksort non è stabile infatti quando vengono scambiate due chiavi non viene tenuto conto di quelle intermedie fra di esse. Anche un algoritmo buono come Quicksort ha dei casi che lo portano a degenerare ad un costo quadratico per ordinare una collezioni di chiavi, infatti anche se mediamente il costo per ordinare una collezione di n chiavi è $n\log_2 n$, nel caso in cui le chiavi della collezione da ordinare siano già ordinate,

oppure ordinate nel senso opposto a quello desiderato, dal processo di partizionamento si vanno a creare partizioni sbilanciate.

APPROCCIO OTTIMO PER SOTTO ISTANZE DI PICCOLE DIMENSIONI

Quando le istanze da ordinare sono molto piccole, è stato studiato che algoritmi più semplici come per esempio Insertion Sort [19] risultano più efficienti rispetto ad altri come Quicksort, che mediamente risultano qualitativamente migliori. Quicksort riduce il tempo di esecuzione di circa il 20% grazie all'integrazione di Insertion Sort, infatti tale algoritmo viene richiamato nel caso in cui le sotto istanze generate dal passo di partizionamento rientrano in una certa cardinalità. La soglia sotto la quale le sotto istanze dei nostri problemi saranno considerate piccole assumeremo essere di 27 chiavi perché è quella che è stata assunta nell'implementazione dell'algoritmo Dual Pivot Quicksort [15] da parte dello studioso russo Vladimir Yaroslavskiy. Quando questa soglia è rispettata è possibile affrontare problemi di ordinamento di sotto istanze con l'algoritmo Insertion Sort, sia quando il problema si presenterà immediatamente, ovvero l'istanza di partenza sarà formata da meno di 27 chiavi da ordinare sia quando la cardinalità delle sotto istanze generate ricorsivamente dal partizionamento rientrerà appunto in tale soglia. La modifica che consiste nell'uso di Insertion Sort usata da Quicksort Classico è applicabile anche a Dual Pivot Quicksort, quindi andiamo a ricordare come funziona Insertion Sort. Insertion Sort è un algoritmo che lavora sul posto ed ha la proprietà che dopo k iterazioni, k chiavi risultano ordinate (con $k \in R$). Infatti ad ogni iterazione viene rimossa una chiave dal sottoinsieme delle chiavi non ordinate. Tale algoritmo utilizza due indici, un indice i che indica la chiave da ordinare e un indice j che scorre il sottoinsieme di chiavi già ordinate a partire da destra verso sinistra (inizialmente tali indici puntano rispettivamente alla seconda chiave della collezione e alla prima). Se j punta ad una chiave maggiore di i, questa viene spostata di una posizione verso destra e j viene decrementato; tale procedimento è ripetuto fino a che non viene trovato il punto nella collezione di chiavi dove l'elemento puntato dall'indice i debba essere inserito. Il funzionamento dell'algoritmo ha l'effetto di spostare gli elementi maggiori verso il fondo della collezione. Di seguito è riportato un esempio di funzionamento dell'algoritmo preceduto dallo pseudocodice.

```
for i = 1 to length[chiavi]:
```

```
val = chiavi[i]
j = i-1
while j >= 0 and chiavi[j] > val:
  chiavi[j+1] = chiavi[j]
  j = j-1
chiavi[j+1] = val
```

Supponiamo di voler ordinare la seguente collezione di chiavi:

22 37 11 9 14 46 2 38

Sono riportati sotto i passi dell'algoritmo con evidenziate in rosso le chiavi che sono state spostate, allo scopo di far collocare la successiva della collezione.

22

22 37

11 22 37

9 11 22 37

9 11 14 22 37

9 11 14 22 37 46

2 9 11 14 22 37 46

2 9 11 14 22 37 38 46

L'implementazione dell'algoritmo è presente nell'appendice.

SCELTA DEL PERNO COME MEDIANO FRA TRE CHIAVI CANDIDATE

Quicksort può essere migliorato con una riduzione di circa il 5% del tempo medio d'esecuzione per ordinare una collezione, grazie ad una scelta diversa dell'elemento perno rispetto a quella più semplice di prendere una delle due chiavi agli estremi dell'istanza [25]. Una possibilità è quella di considerare *chiavi*[left], *chiavi*[right] e *chiavi*[(left + right)/2], ovvero la chiave più a sinistra della collezione, quella più a destra e quella centrale come candidate per essere prese come perno. Vengono eseguiti i seguenti passi:

- I tre candidati vengono ordinati fra di loro.
- Il candidato che risulta di valore medio viene scambio con *chiavi[*$\texttt{right} - 1$*]*.
- Quicksort viene applicato alle chiavi che vanno da $\texttt{left} + 1$ a $\texttt{right} - 2$ usando come perno *chiavi[*$\texttt{right} - 1$*]*.

Gli effetti di questa strategia sono:

- Eliminazione delle sentinelle.
- Riduzione drastica della probabilità che si presentino partizioni sbilanciate.
- Incremento della velocità d'esecuzione dell'algoritmo.

PERCHÉ QUESTA STRATEGIA NON NECESSITA DELL'USO DI SENTINELLE?

Perché *chiavi[*$\texttt{left}$*]* e *chiavi[*$\texttt{right} - 1$*]* svolgono questo ruolo, dato che risultano rispettivamente più piccola e più grande in confronto al perno.

PERCHÉ QUESTA STRATEGIA RIDUCE DRASTICAMENTE LA PROBABILITÀ CHE SI PRESENTINO PARTIZIONI SBILANCIATE?

Perché due delle chiavi candidate a perno dovranno risultare le più piccole o le più grandi all'interno della collezione.

CONSIDERAZIONE FINALE

Possiamo terminare il nostro richiamo su Quicksort ricordando che:

- CASO OTTIMO: Tutte le partizioni dividono *chiavi* esattamente a metà, quindi creando due sotto istanze di uguale cardinalità.
- CASO PEGGIORE: Chiavi già ordinate, oppure chiavi ordinate nel senso opposto a quello che vogliamo ottenere, queste due configurazioni sono difetti di cui l'algoritmo soffre pur essendo considerato buono a livello di generalità.

3

DUAL PIVOT QUICKSORT

Per descrivere l'algoritmo Dual Pivot Quicksort andiamo a definire alcune terminologie che verranno usate nel corso della trattazione:

- P1: Chiave minore tra le due scelte come perno.
- P2: Chiave maggiore tra le due scelte come perno.
- Elemento semplice: Chiave della collezione che non è uno dei due perni.

L'algoritmo Dual Pivot Quicksort rispetta il principio Divide et Impera usando il partizionamento di Yaroslavskiy per ordinare una collezione di tipi primitivi, dividendola in tre sottoinsiemi definiti da P1 e P2 dopo ogni passo di partizionamento, (i quali raggiungono la loro posizione definitiva) nel seguente modo:

Elementi semplici $< P1$	$P1$	$P1 \leq$ Elementi semplici $\leq P2$	$P2$	Elementi semplici $> P2$

Figura 2.: Schema organizzazione chiavi dopo il partizionamento

La scelta del perno può essere effettuata prendendo *chiavi[*`left`*]* come P1 e *chiavi[*`right`*]* come P2, che risulta più semplice da implementare ma anche meno efficiente. Una strategia più raffinata è quella di considerare i perni come chiavi in posizioni *chiavi[*`left` $+$ `third`*]* come P1 e *chiavi[*`right` $-$ `third`*]* come P2 , dove `third` ha come valore un terzo del numero di elementi dell'istanza iniziale e poi viene aggiornato durante l'esecuzione dell'algoritmo [15]. Come detto in precedenza per Quicksort le sotto istanze del problema originate dal processo di partizionamento (anche nel caso di Dual Pivot Quicksort) se hanno una cardinalità che rientra nella soglia delle 27 chiavi possono essere ordinate con Insertion

Sort se integrato nell'algoritmo. Dopo queste considerazioni preliminari ma fondamentali andiamo a descrivere il processo di partizionamento di Yaroslavskiy che è il cuore dell'algoritmo Dual Pivot Quicksort.

SCHEMA DEI QUATTRO PASSI D'ESECUZIONE DI DUAL PIVOT QUICKSORT:

1. Controllo della dimensione dell'istanza corrente da ordinare, se risulta avere una cardinalità minore di 27 allora viene richiamato l'algoritmo Insertion Sort (se integrato);

2. Vengono selezionati gli elementi perni al passo attuale, della corrente istanza da ordinare. P1 viene confrontato con P2, se $P1 < P2$ i perni vengono scambiati altrimenti vengono lasciati nella rispettive posizioni (se P1 e P2 sono scelti come `left` e `right`);

3. Passo di partizionamento con il quale si ottengono i sottoinsiemi schematizzati visti in figura 1;

4. Collocamento nella loro posizione finale dei due perni utilizzati nel partizionamento appena finito;

Ma più nello specifico possiamo analizzare il sottoinsieme centrale, andando a rendere più dettagliata la nostra figura:

$P1$	$< P1$	$P1 \leq and \leq P2$	?	$> P2$	$P2$
$left$		$\overset{l}{\rightarrow}$	$\overset{k}{\rightarrow}$ $\overset{g}{\leftarrow}$		$right$

Figura 3.: Organizzazione dettagliata delle chiavi durante il partizionamento

Assumiano che:

- Il sottoinsieme dell'istanza che racchiude le chiavi strettamente minori di P1 sia chiamato *Parte1* e va dalla posizione $\texttt{left} + 1$ alla $\texttt{l} - 1$.

- Il sottoinsieme dell'istanza che racchiude le chiavi maggiori o uguali di P1 e minori o uguali di P2 sia chiamato *Parte2*.

- Il sottoinsieme dell'istanza che racchiude le chiavi maggiori a P2 sia chiamato *Parte3*.

- Il sottoinsieme dell'istanza che racchiude le chiavi che ancora non sono state confrontate con i perni e quindi non collocate in uno dei tre sottoinsiemi citati in precedenza, sia chiamata *Parte4*.

Il sottoinsieme centrale è caratterizzato dal raggruppare due tipi di elementi semplici, ovvero quelli contenuti in *Parte2* e *Parte4*, in particolare poniamo l'attenzione sulle chiavi in *Parte4*; esse al passo corrente risultano sconosciute al modello adottato in quanto durante l'esecuzione dell'algoritmo k può non essere arrivato ad incrociarsi con g a causa del suo incremento né del decremento stesso di g, questo significa che le chiavi appartenenti a *Parte4* non sono state confrontate con i perni. La struttura appena descritta caratterizza il partizionamento di Yaroslavskiy durante la sua esecuzione, quindi ad ogni iterazione la chiave in posizione k viene confrontata con P1, se è minore dovrà essere posizionata in *Parte1* effettuando uno scambio tra *chiavi[k]* e *chiavi[l]*, incrementando l'indice l (che risulta minore o uguale a k ed aggiorna dinamicamente *Parte1* nella partizione corrente). Altrimenti sarà confrontata con P2: se *chiavi[k]*$>$ P2 allora *chiavi[k]* dovrà essere posizionata in *Parte3* attraverso uno scambio con *chiavi[g]*. Quindi fino a che *chiavi[g]*$>$ P2 e $k < g$ l'indice g verrà decrementato, infatti tale indice serve idealmente a definire l'inizio di *Parte3*, successivamente avverrà un scambio tra *chiavi[k]* e *chiavi[g]*. Poi avverrà un confronto tra *chiavi[k]* e P1, ovvero se *chiavi[k]*$<$ P1 allora avverrà lo scambio tra *chiavi[k]* e *chiavi[l]* e l'indice l sarà incrementato. Questo significa che la chiave dovrà essere collocata in Parte1 essendo minore di P1. Successivamente l'algoritmo può procedere al suo passo successivo incrementando l'indice k (operazione svolta sempre alla fine di ogni iterazione del ciclo while), iterando nuovamente il procedimento nel caso in cui $k \leqslant g$, ovvero esaminerà altre chiavi appartenenti alla *Parte4*. Il processo di partizionamento continua fino a che $k \leqslant g$, ovvero fino a che k e g non si incrociano, dopodiché vengono restituiti gli indici $l-1$ e $g+1$ da usare nella successiva invocazione dell'algoritmo di partizione di Yaroslavskiy, dopo un opportuno decremento di l ed un opportuno incremento di g. Gli elementi a sinistra di k sono minori o uguali di P2, gli elementi a destra di g sono maggiori. Inoltre l precede k aggiornando dinamicamente *Parte1* e separa le chiavi più piccole di P1 da quelle che si trovano tra P1 e P2. L'indice g, aggiornando dinamicamente Parte3, separa le chiavi maggiori di P2 da quelle che si trovano tra P1 e P2. Quando il processo di partizionamento finisce, k e g si incrociano e quindi l e g dividono l'istanza corrente in tre sottoinsiemi come mostrato nella figura 4. Alla fine di ogni partizionamento P1 viene scambiato con

chiavi[l] raggiungendo la sua posizione definitiva nella collezione. Stessa cosa avviene per P2 che viene invece scambiato con *chiavi[g]*.

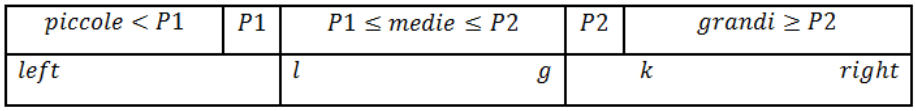

Figura 4.: Organizzazione dettagliata delle chiavi dopo il partizionamento

Andiamo a definire gli elementi semplici dei tre sottoinsiemi appena ottenuti:

- CHIAVE PICCOLA : Chiave appartenente a *Parte1*;
- CHIAVE MEDIA : Chiave appartenente a *Parte2*;
- CHIAVE GRANDE : Chiave appartenente a *Parte3*;

PSEUDOCODICE DUAL PIVOT QUICKSORT

Possiamo schematizzare la spiegazione della sezione precedente attraverso uno pseudocodice:

```
P1 = chiavi[left]
P2 = chiavi[right]
l = left + 1
k = l
g = right - 1
while k <= g
  if chiavi[k] < P1
    scambio chiavi[k] con chiavi[l]
    l = l + 1
  else
    if chiavi[k] > P2
      while chiavi[g] > P2 and k < g
        g = g - 1
      end while
      scambio chiavi[k] con chiavi[g]
      g = g - 1
      if chiavi[k] < P1
        scambio chiavi[k] con chiavi[l]
        l = l + 1
```

```
      end if
    end if
  end if
  k = k + 1
end while
l = l - 1
g = g + 1
scambio chiavi[left] con chiavi[l]
scambio chiavi[right] con chiavi[g]
QuicksortDualPivot(chiavi, left, l - 1)
QuicksortDualPivot(chiavi, l + 1; g - 1)
QuicksortDualPivot(chiavi, g + 1, right)
```

Andiamo a vedere il funzionamento dell'algoritmo con un esempio. Supponiamo di voler ordinare la seguente collezione di chiavi:

10 12 57 58 9 1 46 4 9 100

Prendiamo come elementi perno *chiavi[*left*]* e *chiavi[*right*]*, dato che $10 < 100$ i due perni non vengono scambiati fra di loro. Inizia il partizionamento, l'indice l rimane fermo fino alla terza iterazione dato che *chiavi[*1*]*, *chiavi[*2*]* e *chiavi[*3*]* risultano maggiori di P1, mentre l'indice k viene incrementato ad ogni iterazione indipendentemente da tutto. Quando avviene il confronto *chiavi[*4*]*$<$ P1 allora viene effettuato il primo scambio tra *chiavi[*k*]* e *chiavi[*l*]*, ovvero tra 9 e 12, dopo questa operazione l viene quindi incrementato per la prima volta. Stessa situazione si verifica quando l'indice k arriva su *chiavi[*5*]* ovvero 1, quest'ultima chiave viene scambiata con 57 attraverso un ragionamento analogo col primo scambio effettuato. Prima che k diventi strettamente maggiore di g vengono scambiate allo stesso modo anche la chiave 58 e la chiave 4, vengono inoltre scambiate le chiavi 9 e 12, dopodiché il primo partizionamento può considerarsi terminato. A questo punto viene decrementato l ed incrementato g, *chiavi[*left*]* (ovvero P1) è scambiato con *chiavi[*l*]* e *chiavi[*right*]* è scambiato con *chiavi[*g*]* cosicché i perni raggiungano la loro posizione finale rispetto al problema d'ordinamento della collezione iniziale. In questo caso 10 e 100 sono quindi scambiati con 9 e con 100 stesso. L'insieme delle chiavi dopo la prima partizione è dunque così organizzato:

9 9 1 4 10 57 46 58 12 100

Possiamo osservare come gli elementi che precedono P1, siano tutti strettamente minori di esso andando quindi a formare *Parte1*. Le chiavi

che sono comprese tra P1 e P2 (ovvero 100) sono maggiori non strette di P1 e minori non strette di P2, tale intervallo costituisce *Parte2*. Notiamo inoltre come la *Parte3* sia vuota dato che P2 è la chiave maggiore della collezione. Un'altra osservazione molto importante da fare è che le chiavi coinvolte nelle chiamate ricorsive e quindi soggette a partizionamento sono quello appartenenti a *Parte1*, *Parte2* e *Parte3*. Infatti gli estremi entro i quali viene richiamato l'algoritmo sono dati proprio dagli indici l e g rispettivamente per chiamate ricorsive sinistre e destre, quindi le chiavi apparteneti a *Parte4* diventeranno soggette a partizionamento solo dopo essere state esaminate. L'algoritmo viene quindi richiamato sulla sotto istanza sinistra generata dal partizionamento, ovvero:

9 9 1 4

Il processo di partizionamento ricomincia considerando come perni 9 e 4 con l e k posizionati inizialmente in posizione 1, e g in posizione 2. La prima operazione che viene fatta è quella di scambiare i due perni fra di loro, in quanto *chiavi[*left*]* è maggiore di *chiavi[*right*]*, successivamente k viene incrementato perché *chiavi[*1*]* non è nè maggiore stretto di P2, né minore stretto di P1, all'iterazione successiva la chiave puntata da k sarà 1 che risulterà minore stretta di 4. Analogamente con quanto visto prima avviene uno scambio tra *chiavi[*k*]* e *chiavi[*l*]* con successivo incremento di l e di k, all'iterazione successiva k sarà maggiore stretto di g e quindi il passo di partizionamento terminerà. l viene poi decrementato e *chiavi[*l*]* scambiato con *chiavi[*left*]*, g viene incrementato e *chiavi[*g*]* viene scambiato con *chiavi[*right*]*. 4 e 9 che sono stati i perni in questo passo di partizionamento quindi risultano ordinati, otteniamo così la sequente sotto istanza:

1 4 9 9

In rosso sono evidenziati i perni durante il partizionamento appena concluso. L'algoritmo viene richiamato quindi su 1 e su 9 per la quale si otterrà un ordinamento con un solo passo visto che sono elementi singoli. Andiamo a vedere invece come si comporta l'algoritmo con la sotto istanza destra originata subito dopo il primo passo di partizionamento, ovvero con:

57 46 58 12

57 e 12 vengono subito scambiati tra di loro dato che essendo i perni risultano disordinati. Per capire meglio l'algoritmo, possiamo osservare come 12 sia la chiave più piccola della sotto istanza, l'obiettivo di Dual Pivot Quicksort è quello di realizzare questo fatto più velocemente possibile, allora in questo caso k è incrementato insieme alle iterazioni fino a che non diventa maggiore di g. l che non è mai stato modificato in questo partizionamento viene adesso decrementato insieme all'incremento di g. Avviene quindi uno scambio tra *chiavi[*l*]* e *chiavi[*left*]* che sono la stessa chiave, a conferma del fatto che 12 fosse già nella sua posizione finale. Mentre g è stato decrementato quando la condizione *chiavi[2]* $>$ P2 è stata rispettata. Finito il partizionamento tale indice è decrementato e provoca quindi lo scambio del 57 col 58. Le chiavi 12 e 58 sono dunque state ordinate in questo passo di partizionamento. L'algoritmo viene infine richiamato su 46 e 57 che dopo un'iterazione e due scambi risulteranno ordinati. Ricombinando le sotto istanze si ottiene la soluzione al problema di ordinamento iniziale.

PROPRIETÀ DEL PARTIZIONAMENTO DI YAROSLAVSKIY

1. Le chiavi a sinistra sono prima confrontate con P1. Se e solo se *chiavi[*k*]* non è piccola allora è confrontata con P2.
2. Le chiavi a destra prima sono confrontate con P2. Se non sono più grandi di P2 esse sono confrontate con P1.
3. Ogni chiave piccola eventualmente provoca uno scambio per essere collocata dietro l.
4. Le chiavi grandi in *Parte2* e quelle non grandi in *Parte3* sono sempre scambiate a coppie.

CARATTERISTICHE GENERALI

- L'algoritmo Dual Pivot Quicksort non è stabile, andiamo a verificarlo. Supponiamo di voler di applicare il partizionamento di Yaroslavskiy su questo insieme di chiavi:

  3 3 3 3 4

  Come al solito, P1 sarà scelto con *chiavi[*left*]* con left che durante il primo partizionamento è uguale a 0 e P2 come *chiavi[*right*]*

con g che durante il primo partizionamento è uguale a 4. Nel partizionamento k viene incrementato fino ad essere maggiore stretto di g, il primo passo termina con l che non è stato modificato, quindi verrà scambiato *chiavi[left]* con *chiavi[l]* . Entrambi i valori delle chiavi sono uguali a 3 ma come detto lo scambio avviene ugualmente, concludiamo dicendo che Dual Pivot Quicksort non è stabile;

- L'algoritmo Dual Pivot Quicksort esegue mediamente circa $0.6nln(n)$ scambi per ordinare una collezione di n chiavi [6]. Tale numero di scambi è maggiore rispetto a quello di Quicksort ma Dual Pivot Quicksort risulta lo stesso generalmente più veloce di Quicksort Classico. Nel capitolo relativo alla sperimentazione degli algoritmi è presente anche il conteggio degli scambi.
- L'algoritmo Dual Pivot Quicksort è più veloce di Quicksort Classico, questa affermazione verrà dimostrata sperimentalmente nel corso della tesi;
- L'algoritmo Dual Pivot Quicksort sfrutta il principio del Divide et Impera per ordinare una collezione di elementi primitivi, questo è dovuto al fatto che deriva dal Quicksort Classico il quale come abbiamo detto in precedenza sfrutta tale principio a sua volta;
- L'algoritmo Dual Pivot Quicksort funziona bene con memorie cache, questo fatto viene dimostrato nel corso della tesi.
- Esegue in media circa $1.9nln(n)$ confronti per ordinare una collezione di n chiavi [6].
- Usa due perni invece che uno, questo è determinante per la sua velocità di esecuzione, nel capitolo successivo ne viene spiegata la motivazione.
- È un algoritmo più adattivo di Quicksort Classico in quanto una grande quantità di istruzioni può non essere eseguita se *chiavi[k]* e *chiavi[g]* sono rispettivamente maggiore uguale di P1 e minore o uguale di P2.

DUAL PIVOT QUICKSORT IN JDK7 E JDK8

Dual Pivot Quicksort ha esordito nelle librerie di Java a partire da JDK7: la scelta di assumere questo algoritmo come standard è arrivata dopo

una lunghissima serie di test. Oracle ha implementato l'algoritmo con una scelta dei due elementi perni tra cinque chiavi. Sia n il numero totali della chiavi da ordinare, le candidate si trovano fra loro a distanza circa $\frac{1}{7}n$ [11]. La prima e la quinta distano dalle chiavi esterne di circa $\frac{3}{14}n$, tra queste chiavi vengono selezionati i terzili, ovvero le chiavi che suddividono l'istanza corrente in tre parti della stessa numerosità. In Java 8, Dual Pivot Quicksort ha continuato ad essere lo standard per risolvere i problemi di ordinamento, nella documentazione ufficiale possiamo trovare la seguente descrizione del metodo *sort()*, richiamabile per ordinare collezioni di elementi primitivi:

> *The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers* $O(n \log n)$ *performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations [27].*

4

EFFICIENZA DI DUAL PIVOT QUICKSORT

Dato che Dual Pivot Quicksort esegue più scambi di Quicksort Classico e un numero di confronti abbastanza vicino a quello di quest'ultimo per ordinare una collezione, andiamo a ricercare altrove i motivi della sua efficienza. Ovvero nell'accesso ai dati...

PRINCIPI SU CUI SI BASA LA MEMORIA CACHE

Una memoria è detta cache se rispetta i principi di località spaziale e temporale [23].

PRINCIPIO DI LOCALITÀ TEMPORALE Un dato a cui è stato fatto riferimento da poco tempo è probabile che verrà nuovamente richiesto.

PRINCIPIO DI LOCALITÀ SPAZIALE Una dato vicino ad un altro a cui è stato fatto riferimento da poco tempo è probabile che verrà nuovamente richiesto.

IMPORTANZA DELLA MEMORIA CACHE

Quindi basandosi su questi principi una memoria cache è veloce, piccola e costosa. È utilizzata ovunque in dispositivi elettronici e informatici, a partire dai personal computer fino ai proxy Server, ai DNS Server locali ecc... L'utilizzo della memoria cache è fondamentale nelle architetture moderne visto che ogni anno le prestazioni della memoria aumentano di circa il 10% mentre quelle del processore del 60% [6]. La differenza di prestazioni quindi cresce rapidamente ed è necessario introdurre sempre più livelli di cache per colmare il divario nell'accesso ai dati fra processore e memoria principale. Questo sta a significare che il ruolo delle memorie cache sarà sempre più importante nelle prestazioni dei

programmi. Utilizzando quindi le memorie cache si pone il problema di capire se un dato che è stato richiesto si trovi nella cache stessa oppure no. Potrebbe dunque essere necessario richiederlo in memoria centrale maturando ritardo nella consegna del dato a chi lo ha richiesto. Dividendo idealmente la memoria in una parte superiore (più vicina al processore e quindi di più veloce accesso) ed una inferiore (più lontana dal processore e quindi di più lento accesso) introduciamo le seguenti definizioni:

SUCCESSO NELL'ACCESSO Il dato richiesto si trova nella parte superiore della memoria, ovvero nella memoria cache, quindi è possibile diminuire il tempo di attesa alla richiesta di un dato [23].

FALLIMENTO NELL'ACCESSO Il dato richiesto non si trova nella parte superiore della memoria ma in quella inferiore che rappresenta la memoria centrale, quindi occorrerà portare tale oggetto nella parte superiore della memoria dove potrà essere inviato al richiedente ed essere copiato in cache per richieste future. Nei processori moderni questa situazione comporta centinaia di istruzioni in più svolte dal processore stesso, quindi ai fini dell'esecuzione di un algoritmo è un evento da evitare il più possibile [23].

BLOCCO Minima unità di dati [23].

CACHE L1 (CACHE DI PRIMO LIVELLO) cache integrata direttamente nel processore [23].

CACHE L2 (CACHE DI SECONDO LIVELLO) cache non integrata direttamente nel processore ma utilizza un bus per accedervi [23].

RUOLO DEL NUMERO DEI PIVOT NELL'EFFICIENZA DELL'ALGORITMO

Studi relativi al multipivoting in Quicksort (come Multi-Pivot Quicksort: Theory and Experiments [6]) hanno dimostrato che all'aumentare del numero dei pivot, l'algoritmo cresce di efficienza per quanto riguarda il suo uso in memoria cache. A cosa è dovuta tale efficienza? Il numero medio di accessi in memoria si riduce all'aumentare del numero di perni all'interno della partizione e quindi si riducono i fallimenti nell'accesso generati. Dopo la pubblicazione di Dual Pivot Quicksort da parte di Vladimir Yaroslavskiy è stata posta molta attenzione al comportamento Quicksort in memoria cache, infatti alcuni studiosi della University of

Waterloo in un recente articolo datato per la precisione 7 Novembre 2013 (Multi-Pivot Quicksort: Theory and Experiments [6]) hanno dimostrato che una versione dell'algoritmo Quicksort a tre perni è molto efficiente nello sfruttare la memoria cache, anche se esegue più scambi di Quicksort Classico. Studi su tale algoritmo implementato in C (quindi evitando per sicurezza influenze sulla rilevazione dei tempi dovuti alla JVM nel caso fosse stato usato Java), hanno dimostrato sperimentalmente che risulta più efficiente di Dual Pivot Quicksort, impiegando mediamente $7-8\%$ in meno del tempo per ordinare una collezione di chiavi rispetto a quest'ultimo. Ma perché aumentando il numero di perni l'algoritmo Quicksort diventa sempre più efficiente nella memoria cache? Per il fatto che più perni vengono usati, più sotto istanze di piccole dimensioni vengono create durante il processo di partizionamento, questo implica una maggiore probabilità che le chiavi relative alla sotto istanza potranno essere caricate al costo di un numero piú basso di fallimenti nell'accesso rispetto ad istanze di maggiori dimensioni create dal partizionamento di Quicksort Classico. Infatti le istanze generate dal partizionamento di Quicksort Classico essendo di cardinalità maggiore richiedono solitamente un maggior numero di accessi alla memoria cache per essere caricati all'interno di essa, dato che la dimensione del blocco potrebbe non essere sufficiente a permettere un caricamento con pochi accessi [23]. D'altra parte il partizionamento diventa più complicato e ci sarà un numero maggiore di chiavi fuori posto, questo va ad aumentare il numero di scambi necessari per ordinare la collezione. In virtù di queste osservazioni possiamo concludere che la velocità di accesso ai dati risulta più determinante del numero di confronti e scambi per quanto riguarda la velocità di Quicksort. Studi condotti da Shrinu Kushagra, Alejandro López-Ortiz, J.Ian Munro e Aurick Qiao della University of Waterloo hanno prodotto i seguenti risultati: sia D la dimensione della cache, B la dimensione di ogni singolo blocco ed n il numero di chiavi da ordinare [6], le seguenti formule stabiliscono i fallimenti nell'accesso nella memoria cache:

- Quicksort: $2(\frac{n+1}{B})\ln(\frac{n+1}{D+2})$
- Dual Pivot Quicksort: $\frac{8}{5}(\frac{n+1}{B})\ln(\frac{n+1}{D+2})$
- Three Pivot Quicksort: $\frac{18}{13}(\frac{n+1}{B})\ln(\frac{n+1}{D+2})$

Dopo aver mandato in esecuzione i tre algoritmi su molti test allo scopo di ordinare 10.000.000 di chiavi, sono stati ottenuti i seguenti risultati:

- Quicksort: 3.700.000 fallimenti nell'accesso medi rilevati.
- Dual Pivot Quicksort: 3.100.000 fallimenti nell'accesso medi rilevati.
- Three Pivot Quicksort: 2.700.000 fallimenti nell'accesso medi rilevati.

Quanto detto in precedenza diventa maggiormente significativo in caso di cache L1 ed L2 dato che le memorie di questo tipo sono quelle più vicine possibile al processore. L'aumento del numero dei perni per offrire efficienza a Quicksort deve essere supportato da un'implementazione che sfrutti le potenzialità dell'algoritmo a livello di memoria cache. Ad oggi implementazioni Quicksort con numero di perni maggiore di 5 risultano più lente rispetto a Dual Pivot Quicksort. Le prestazioni dell'algoritmo quindi dipendono dall'architettura sottostante e dalla memoria cache. In futuro il divario di prestazioni tra Quicksort e Dual Pivot Quicksort è probabile che aumenti, perché più le memorie cache saranno migliorate tanto più implementazioni Multi-Pivoting di Quicksort riusciranno a fare la differenza in termini di velocità d'esecuzione rispetto a Quicksort Classico.

IPOTESI SULLA PIPELINE

Dato che l'algoritmo Dual Pivot Quicksort non è più efficiente da un punto di vista della quantità di scambi e per quanto riguardo il numero di confronti delle chiavi risulta non avere un vantaggio significativo rispetto a Quicksort Classico, i motivi della sua efficienza sono da ricercare altrove. Abbiamo appena visto che almeno uno dei motivi che giustificano la velocità d'esecuzione di Dual Pivot Quicksort è dovuto alla sua efficienza in memoria cache. Ma adesso ci chiediamo se tale efficienza sia dovuta tutta ai motivi detti nella sezione precedente oppure no. Quindi si potrebbe ipotizzare che un altro motivo dell'efficienza dell'algoritmo sia dovuto al fatto che riesca a sfruttare meglio della sua versione classica l'esecuzione in Pipeline delle istruzioni. La Pipeline è una tecnica utilizzata dai processori moderni per parallelizzare l'esecuzione di più istruzioni allo scopo di aumentare il throughput (ovvero la quantità di istruzioni eseguite per unità di tempo). L'esecuzione di un'istruzione in un processore è caratterizzata da cinque fasi [23]:

- Instruction Fetch (IF): Viene letta l'istruzione dalla memoria.
- Decode (ID): Viene individuato il tipo di istruzione perché in base ad esso verranno letti un certo numero di registri. Per registri si

intende delle piccole unità di memoria implementate nel processore che permettono un accesso molto veloce alle informazioni in esse contenuti. Infatti vengono utilizzati per massimizzare il throughput.

- Execute (EX): Viene eseguita l'istruzione.
- Memory (MEM): Viene chiamata in causa la memoria per i tipi di istruzioni che lo richiedono.
- Write Back (WB): Viene scritto il risultato dell'operazione svolta dall'istruzione in un determinato registro.

In riferimento a queste cinque fasi, l'idea che sta alla base della Pipeline è quindi quella di far iniziare la fase di IF dell'istruzione $i+1$ con $i = 1, 2, ..., n$ (e n che è il numero dell'ultima istruzione) direttamente quando l'istruzione i è in fase di ID. Come mostrato nella seguente immagine:

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Figura 5.: Esempio di utilizzo della Pipeline

Il un processore senza Pipeline l'istruzione $i-esima$ inizierebbe soltanto dopo la WB dell'istruzione $i-1$.

Un fattore che influenza la velocità d'esecuzione dei processori moderni è dovuto all'implementazione della predizione dinamica per quanto riguarda l'istruzione branch.

Per spiegare questo concetto è opportuno ricordare alcuni concetti sulle Pipeline:

- Stallo: È lo stato in cui si trova il processore quando le istruzioni sono bloccate, questo può capitare nel caso in cui si hanno delle criticità.

- Criticità: Momento in cui un'istruzione non può essere eseguita nel ciclo di clock immediatamente seguente. Ne esistono di vari tipi ma a noi interesserà soltanto quella sul controllo.
- Criticità sul controllo: Criticità che si verifica quando un'istruzione di salto deve conoscere il risultato dell'esecuzione della precedente istruzione per sapere se il salto deve essere effettuato oppure no. Questa criticità può verificarsi perché una combinazione di istruzioni potrebbe far sì che dei dati utili all'istruzione di salto non siano disponibili al momento del confronto.
- Istruzione branch: Tipo di istruzione per il controllo di flusso, la cui destinazione è specificata con un valore di spiazzamento che in caso di salto deve essere sommato all'indirizzo dell'istruzione corrente.

Infatti allo scopo di prevenire uno stallo nella Pipeline viene eseguita una previsione dinamica sull'esito del salto. Questa tecnica porta benefici in caso di successo perché la Pipeline è già predisposta a lavorare con quel risultato, contrariamente invece crea ritardi importanti perché occorre ripulire i registri di Pipeline (registri di cui la Pipeline è provvista allo scopo di avere a disposizione il flusso di dati necessario durante l'esecuzione delle istruzioni) dalle informazioni che sono state portate avanti a causa della predizione sbagliata. Nelle implementazioni moderne di Pipeline la predizione dinamica indovina circa al 90% l'esito del salto ma nel 10% dei casi no. La penalità di predizioni perse ovvero il numero di cicli di clock persi a causa delle previsioni sbagliate sull'esito del salto, va a crescere. Inoltre bisogna considerare che questo aspetto sarà sempre più importante nelle prestazioni dei processori moderni dato che il numero di stadi nelle Pipeline odierne è in continuo aumento, ciò vuol dire che più stadi ha una Pipeline più dati contenuti nei registri di Pipeline dovranno essere eliminati in caso di predizione errata. Due studiosi di nome Kaligosi e Sanders [5] hanno sperimentato la supposizione che Dual Pivot Quicksort sfrutti meglio di Quicksort l'esecuzione in Pipeline delle istruzioni su un Pentium 4 Prescott, con una Pipeline appunto nella quale le previsioni sull'esito del salto della branch fossero il fattore determinante per il tempo d'esecuzione. Negli esperimenti da loro svolti è stata utilizzata sia la tecnica del mediano fra tre chiavi candidate per la scelta del perno nel Quicksort che la selezione del perno random. È stato osservato che il mediano fra tre non porta benefici dal punto di vista dei fallimenti di predizioni perse, anzi peggiora la situazione. Il numero

di fallimenti nelle previsioni degli esiti delle branch è risultato avere un valore simile per Quicksort Classico rispetto a Dual Pivot Quicksort. Si conclude quindi che la differenza di velocità tra i due algoritmi non è dovuta ad un ipotetico miglior uso della Pipeline da parte di Dual Pivot Quicksort rispetto a Quicksort Classico. Per maggiori dettagli sull'esperimento appena descritto è consigliata la visione dell'articolo 'How Branch Mispredictions Affect Quicksort' [5] che racchiude l'esperienza fatta da Kaligosi e Sanders, il cui riferimento è presente nella bibliografia.

5

VERIFICA SPERIMENTALE

INTRODUZIONE AI TEST

In questa sezione vengono descritti i test effettuati al fine di verificare che il comportamento dell'algoritmo Dual Pivot Quicksort è quello descritto nei capitoli precedenti specialmente in fatto di velocità d'esecuzione. Gli algoritmi sono stati testati senza l'integrazione di Insertion Sort in modo da avere risultati più provanti per gli algoritmi stessi, comunque è stato realizzato un modulo che implementa Insertion Sort che è facilmente utilizzabile per testare gli algoritmi con tale miglioria. È stato scelto di scrivere il codice per la sperimentazione in linguaggio C, in modo da evitare alterazioni sulla rilevazione dei tempi da parte della JVM (nel caso si fosse programmato in Java). Inoltre è stato scelto un ambiente Linux come Ubuntu 16.04 LTS in modo da poter gestire meglio i processi che occupano il processore, dato che tale numero è molto limitato in confronto a un sistema operativo come Windows. Quindi al fine di rilevare più correttamente possibile i tempi degli algoritmi, l'uso del processore è stato ridotto al minimo lasciando attivi solo quei proccessi essenziali per il funzionamento del sistema. Di seguito sono riportate le caratteristiche relative al computer sulla quale sono stati svolti i test:

- Processore: Intel(R) Core(TM)2 Duo T6600 2.20 Ghz
- Cache L1: 32 KiB
- Cache L2: 2 MiB

Dove le unità di misura delle memorie cache sono espresse in prefissi binari, 1 KiB e 1 MiB corrispondono rispettivamente a 1024 e 1.000.000 (circa) byte. È importante riportare le caratteristiche del sistema utilizzato per effettuare i test, soprattutto in una situazione in cui l'efficienza dell'algoritmo dipende proprio dall'architettura sottostante.

I tipi di test effettuati sono:

1. Test delle permutazioni;
2. Test random;
3. Test con chiavi già ordinate;
4. Test con chiavi ordinate nell'ordine inverso;
5. Test che conta i confronti e gli scambi effettuati;

DESCRIZIONE DEI TIPI DI TEST

1. Il primo tipo di test consiste nella popolazione del vettore *chiavi* di un numero di chiavi scelte, con valore della chiave stessa uguale a quella dell'indice. Successivamente *chiavi* è stato permutato sfruttando l'algoritmo di Fischer-Yates [18]:

```
Sia n il numero di chiavi da ordinare
Da n - 1 fino ad 1 iterando con l'indice i:
     j = intero casuale tale che: 0 <= j <= i
     scambio(chiavi, j, i)
```

2. Il secondo tipo di test consiste nella popolazione del vettore *chiavi* in maniera casuale sfruttando la funzione *rand()* di libreria. L'intervallo di valori assumibili dalle chiavi è comunque maggiore o uguale a 0 e minore uguale rispetto alla grandezza del vettore. A differenza del test precedente la presenza di chiavi ripetute è possibile.
3. Il terzo tipo di test si pone come obiettivo quello di dimostrare il comportamento migliore di Dual Pivot Quicksort rispetto a Quicksort Classico in uno dei due casi peggiori del secondo di essi, ovvero quando le chiavi sono già ordinate. In questo caso Quicksort Classico degenera nel suo caso peggiore impiegando un tempo quadratico rispetto al numero di chiavi da ordinare. Viene popolato il vettore *chiavi* con valori corrispondenti all'indice nel quale la chiave si trova (in ordine crescente).
4. Il quarto tipo di test è speculare al terzo, con la sola differenza che le chiavi sono generate in ordine decrescente.

5. Il quinto tipo di test si basa sull'uso di un contatore allo scopo di mostrare che l'algoritmo Dual Pivot Quicksort esegue più scambi per ordinare una collezione rispetto a Quicksort Classico.

Alla fine dei primi quattro tipi di test i dati rilevati come per esempio i tempi di esecuzione di ogni algoritmo vengono salvati in file. I tempi d'esecuzione degli algoritmi sono stati rilevati in microsecondi (10^{-6} secondi) al fine di ottenere un'ottima accuratezza dei risultati. Successivamente sono stati implementati moduli in Python per mostrare i risultati ottenuti, allo scopo di far osservare con più chiarezza possibile l'andamento degli algoritmi. La scelta di passare a Python per la realizzazione dei grafici è dovuta al fatto che in C si sarebbe ottenuto un risultato esteticamente meno soddisfacente.

TEST SULLE PERMUTAZIONI

Il primo test è stato condotto tenendo fisso il numero delle chiavi contenute nel vettore *chiavi*, ovvero 1000. I test hanno seguito un numero crescente di prove eseguite rilevando per ogni esecuzione del programma (costituito dall'esecuzione dei tre algoritmi) le medie di Quicksort, Quicksort con scelta del perno come mediano fra tre chiavi e Dual Pivot Quicksort. Sull'asse x sono presenti i numeri crescenti della quantità di prove eseguite ed il punto corrispondente nel grafico è il tempo medio ottenuto da quell'esecuzione. Il numero di prove eseguite è stato aumentato di 200 ad ogni esecuzione partendo da 100 fino ad arrivare a 1500.

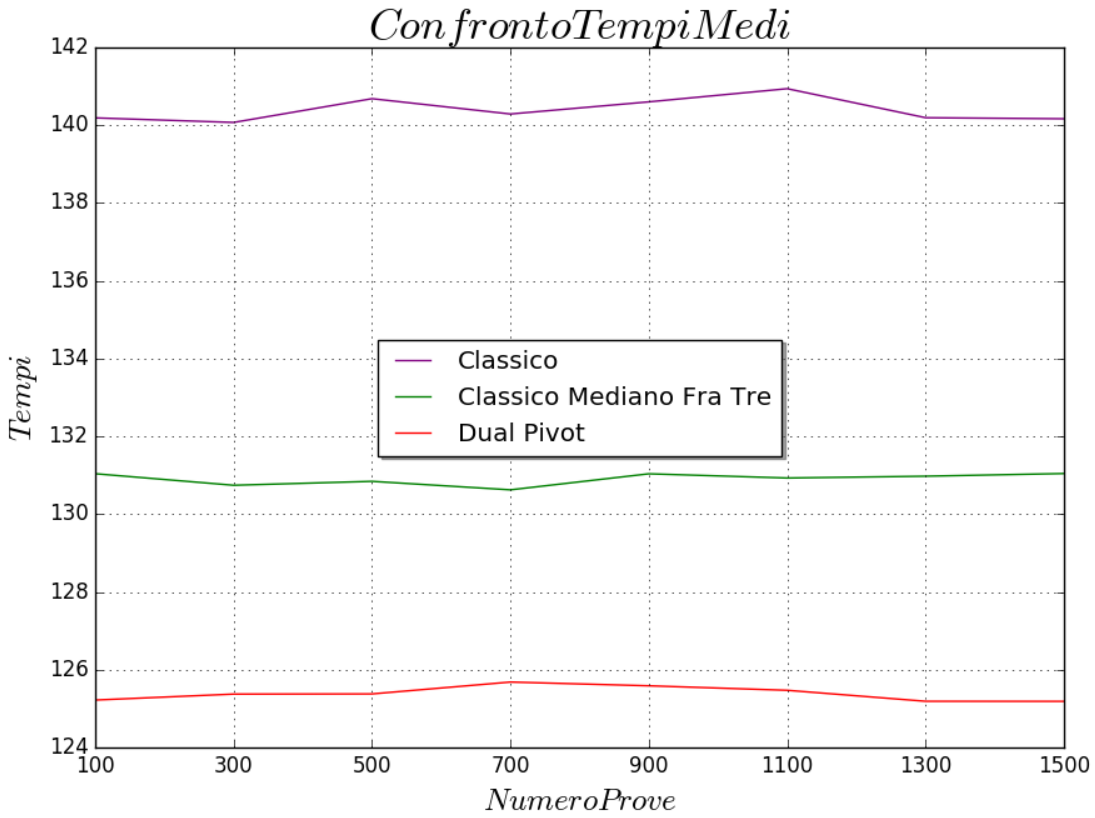

Figura 6.: Test di primo tipo con aumento progressivo del numero delle prove

I dati ottenuti sono conformi con quanto ci aspettavamo da risultati presi in considerazione in altre sperimentazioni, infatti in *'Multi-Pivot Quicksort: Theory and Experiments'* [6] di studiosi dell'University of Waterloo è stato concluso che Dual Pivot Quicksort è circa del 10% più veloce di Quicksort Classico. Successivamente è stato sperimentato l'andamento dei due algoritmi fissando il numero di prove ma aumentando progressivamente il numero delle chiavi. Partendo da 100 chiavi da ordinare si è arrivati a 800 con un incremento di 100 ad ogni esecuzione del programma.

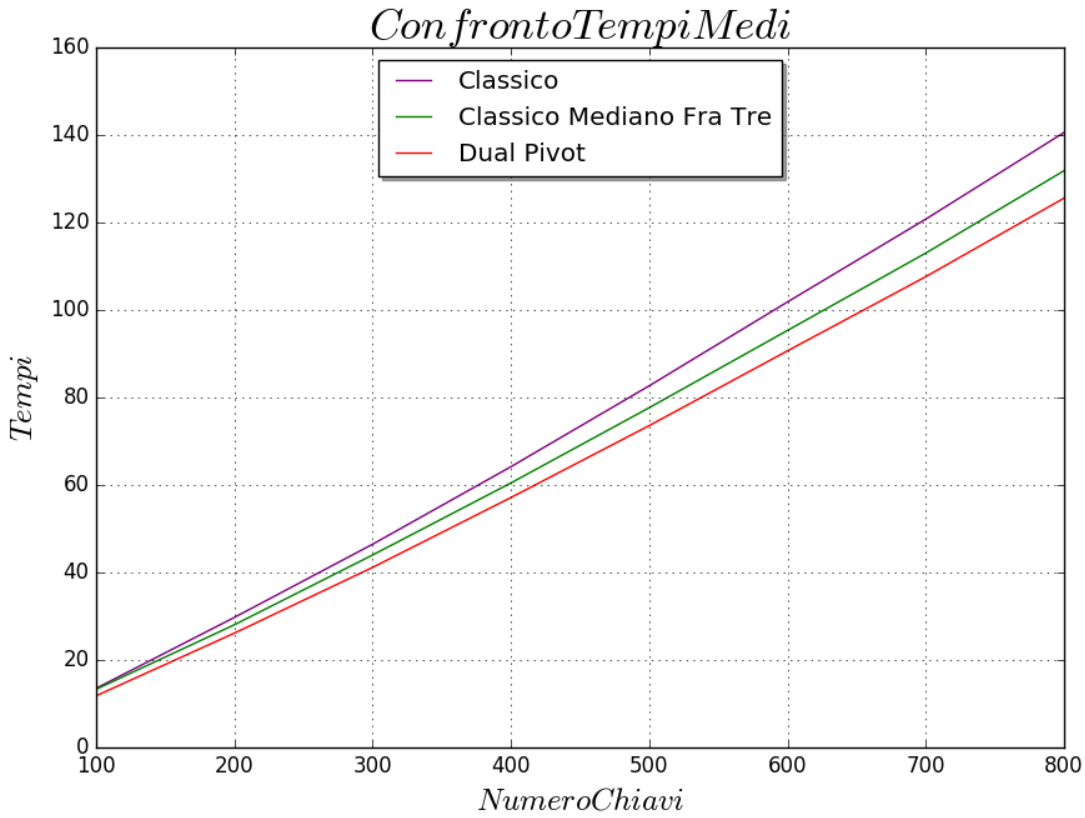

Figura 7.: Test di primo tipo con numero delle chiavi da ordinare aumentato progressivamente

Il test di primo tipo evidenzia che all'aumentare del numero delle chiavi da ordinare, la differenza di velocità fra Quicksort e Dual Pivot Quicksort si fa sempre più evidente in proporzione al numero di chiavi. Possiamo osservare in particolare i tempi rilevati quando gli algoritmi sono mandati in esecuzione con l'obiettivo di ordinare 600 chiavi. Quicksort fa rilevare un tempo di circa 100 microsecondi, Quicksort con scelta del perno come mediano fra tre chiavi circa 95 microsecondi e Dual Pivot Quicksort circa 90 microsecondi. Quanto appena osservato rappresenta una perfetta fotografia che mostra come Dual Pivot Quicksort sia circa 5% più veloce di Quicksort con scelta del perno come mediano fra tre chiavi e 10% più veloce di Quicksort Classico.

TEST RANDOM

Anche per il tipo di test random la prima serie è stata effettuata fissando il numero di chiavi da ordinare ed aumentando progressivamente il numero delle prove (allo stesso modo dei test di tipo 1), i risultati sono conformi con quanto ci aspettavamo. In particolare i tempi medi dell'ordinamento delle chiavi da parte dei tre algoritmi risultano con una differenza ben delineata fin da subito.

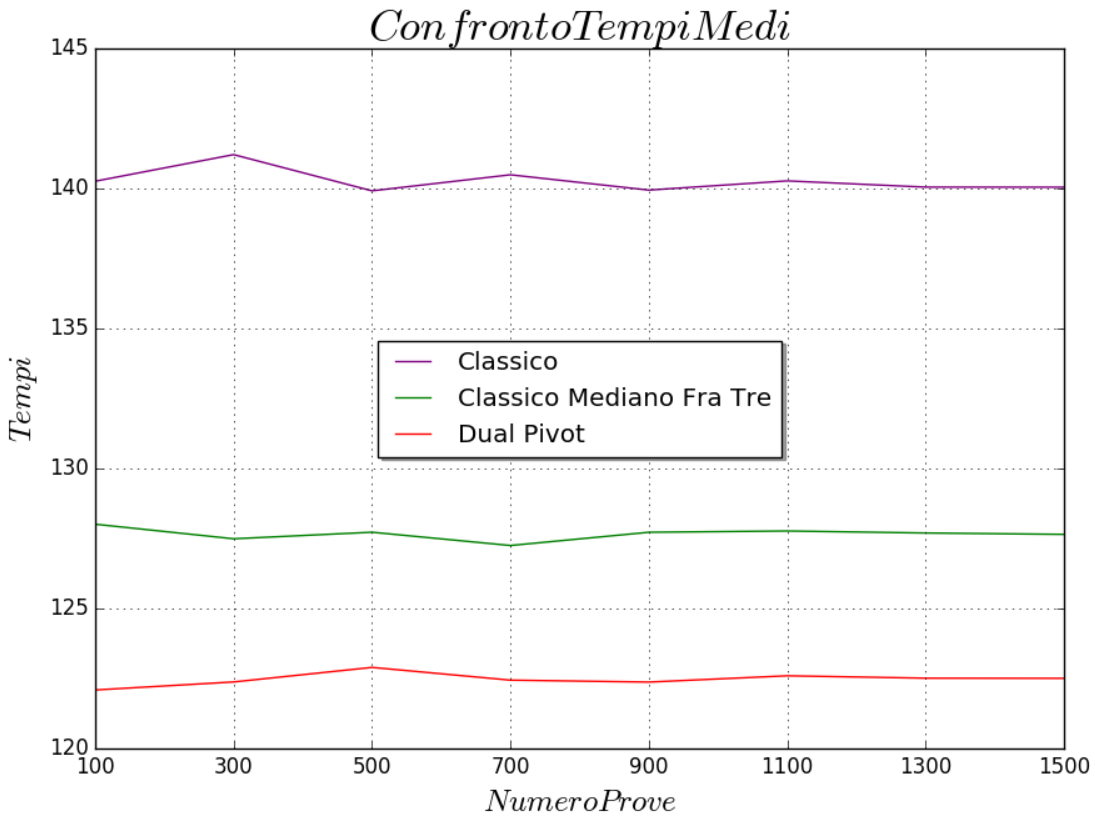

Figura 8.: Test di secondo tipo con numero delle prove eseguite aumentato progressivamente

Successivamente è stato sperimentato l'andamento dei tre algoritmi fissando il numero di prove ma aumentando progressivamente il numero delle chiavi, allo stesso modo dei test di tipo 1. Ovvero partendo da 100

chiavi da ordinare si è arrivati a 800 con un incremento di 100 ad ogni esecuzione del programma.

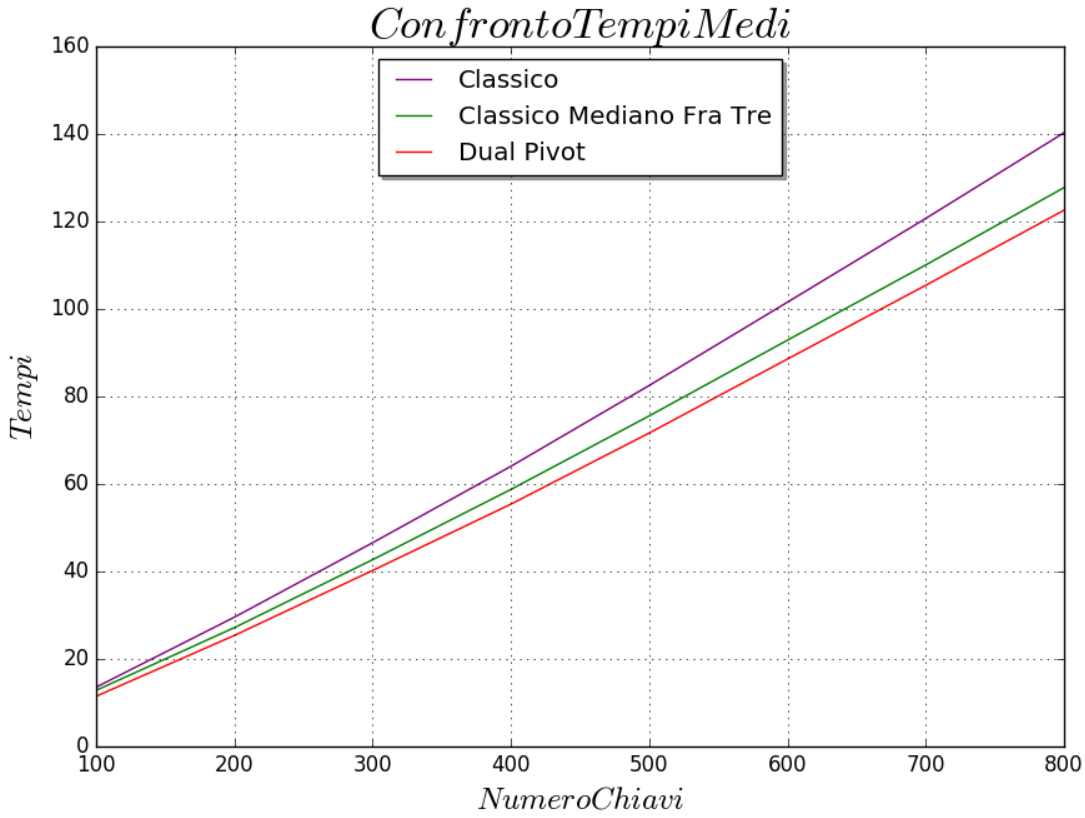

Figura 9.: Test di secondo tipo con numero delle chiavi aumentato progressivamente

Il test di secondo tipo evidenzia che all'aumentare del numero delle chiavi da ordinare, la differenza di velocità tra Quicksort Classico, Quicksort con scelta del perno come mediano fra tre chiavi e Dual Pivot Quicksort si fa sempre più evidente in proporzione al numero di chiavi. Pertanto valgono le considerazioni fatte per il tipo di test precedente.

TEST TIPO 3 E TIPO 4

Osserviamo come si comporta Dual Pivot Quicksort al presentarsi dei casi peggiori di Quicksort Classico, nei grafici seguenti viene evidenziato il comportamento dei tre algoritmi al presentarsi di una collezione di chiavi già ordinata e ordinata nell'ordine inverso a quello desiderato. Il numero delle prove eseguite è stato progressivamente aumentato di 200 da un valore iniziale di 100 ad uno finale di 500, ordinando 500 chiavi.

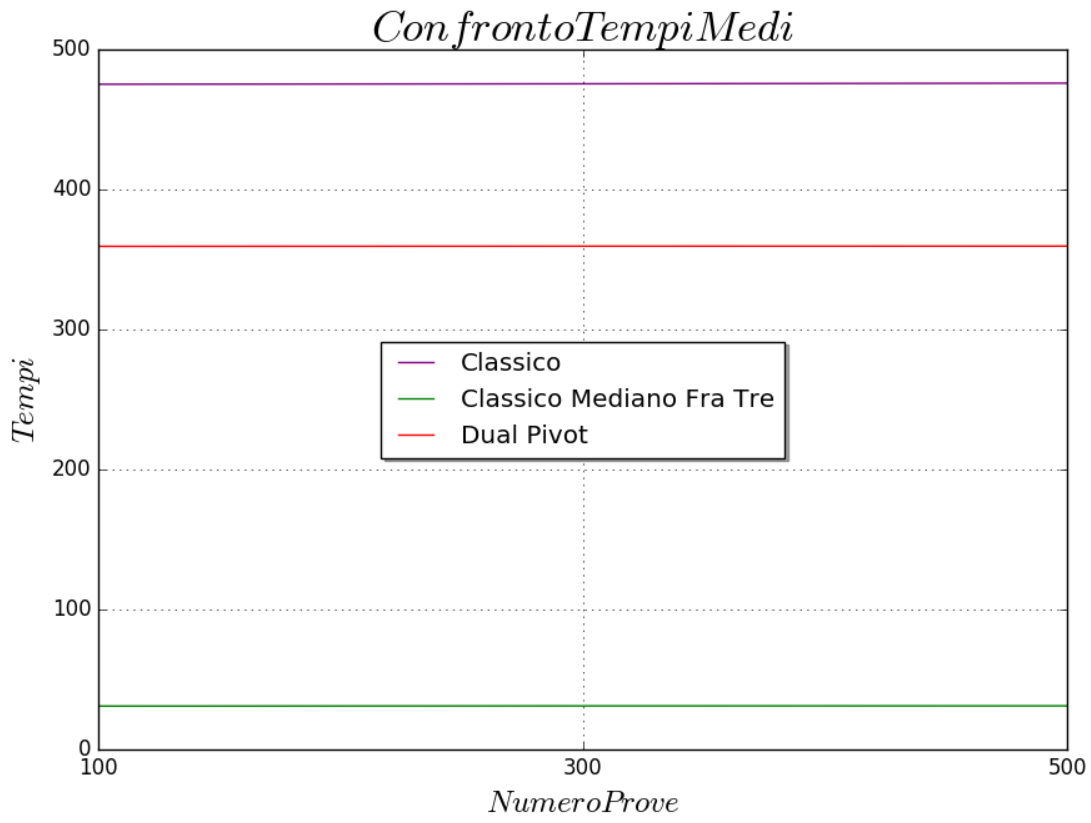

Figura 10.: Test di terzo tipo con numero delle prove eseguite aumentato progressivamente

Osserviamo dal grafico che Dual Pivot Quicksort si comporta meglio rispetto a Quicksort Classico. Inoltre possiamo osservare che Quicksort con scelta del perno come mediano fra tre chiavi si comporta meglio anche di Dual Pivot Quicksort, questo fatto è dovuto alla scelta degli elementi perno come chiavi che si trovano agli estremi della collezione. Questa

scelta di più facile implementazione di fatto crea partizioni sbilanciate e fa degenerare Dual Pivot Quicksort al suo caso peggiore che coincide con quello di Quicksort Classico. Tuttavia può essere implementata una scelta dei perni più accurata, ovvero scegliendo come perni due chiavi che si trovano all'interno dell'istanza da ordinare. Tale modifica permette di superare l'efficienza di Quicksort con scelta del mediano fra tre chiavi anche in questo caso. A tal proposito si veda il documento scritto da Vlamidir Yaroslavskiy nel 2009 presente nella bibliografia [15].

TEST DI TIPO 5

In quest'ultima sezione di test saranno evidenziati dei risultati nei quali non è stato tenuto conto del tempo di esecuzione degli algoritmi ma bensì del numero di scambi eseguiti da Quicksort Classico e Dual Pivot Quicksort per ordinare n chiavi. Attraverso l'utilizzo di un contatore è stato misutato quanti scambi vengono effettuati da Quicksort e Dual Pivot Quicksort, allo scopo di dimostrare sperimentalmente che:

- Quicksort effettua mediamente circa $0.33n\ln(n)$ scambi per ordinare una collezione di n chiavi.
- Dual Pivot Quicksort effettua mediamente circa $0.6n\ln(n)$ scambi per ordinare una collezione di n chiavi.

Allo scopo di verificare al meglio queste formule i due algoritmi sono stati utilizzati senza la miglioria offerta da Insertion Sort per vettori di piccole dimensioni, che ne avrebbe falsato il numero di scambi effettuati. Dopo 1000 prove con un numero di chiavi fissato a 500 (eseguendo un test di tipo 1) le medie ottenute dai due algoritmi sono state rispettivamente

- Quicksort: 1063 scambi, con risutato teorico previsto di circa 1025.
- Dual Pivot Quicksort: 1880 scambi, con risultato teorico previsto di circa 1864.

I risultati ottenuti confermano l'applicabilità delle formule per sapere a priori (conoscendo la cardinalità della collezione) l'ordine di grandezza del numero di scambi che i due algoritmi impiegheranno per risolvere il problema dell'ordinamento.

OPERAZIONI ESEGUITE VS FALLIMENTI NELL'ACCESSO

Per quanto detto in precedenza l'accesso ai dati risulta il fattore dominante per la velocità di Dual Pivot Quicksort e l'architettura sottostante risulta significativa per le prestazioni dell'algortimo. Queste non sono solo considerazioni teoriche ma fattori importanti che sono stati affrontati durante la sperimentazione di Dual Pivot Quicksort. All'inizio del capitolo sono state evidenziate le caratteristiche della macchina sulla quale sono stati svolti i test, tale computer risale ormai a quasi una decina di anni fa. Infatti le memorie cache L1 ed L2 proprie della macchina non permettono all'algoritmo di sfruttare a pieno la sua velocità in caso di un numero di chiavi alto. Facendo girare l'algoritmo su una quantità di chiavi maggiore o uguale a 1500, Dual Pivot Quicksort dà i primi segni di cedimento cominciando a perdere terreno nei confronti di Quicksort con scelta del perno come mediano fra tre chiavi. Quindi all'aumento del numero di chiavi da ordinare le operazioni che l'algoritmo deve svolgere diventano sempre di più, e se l'architettura sottostante non sostiene adeguatamente Dual Pivot Quicksort tali costi tendono a diventare sempre più dominanti nell'esecuzione dell'algoritmo. In particolare è stato testato che da circa 10000 chiavi Dual Pivot Quicksort risulta addirittura più lento di Quicksort con scelta del mediano fra tre chiavi. Il grafico sottostante mostra i risultati dei test eseguiti su grandi quantità di chiavi sulla prima macchina, i test sono stati svolti aumentando progressivamente un numero di chiavi a partire da 1000 fino a 12000 con un passo di 1000 ad ogni esecuzione del programma di test. Come evidenziato nel grafico inizialmente la linea verde che rappresenta Quicksort con scelta del mediano fra tre chiavi candidate è molto vicina a Dual Pivot Quicksort (linea rossa) ma senza superarlo. Già con questa quantità di chiavi da ordinare comunque non cè il 5% di vantaggio di Dual Pivot Quicksort rispetto a Quicksort con scelta del perno come mediano fra tre chiavi candidate. All'aumentare delle chiavi la linea verde sovrappone quella rossa fino a portarsi sotto di essa da circa 10000 chiavi in poi.

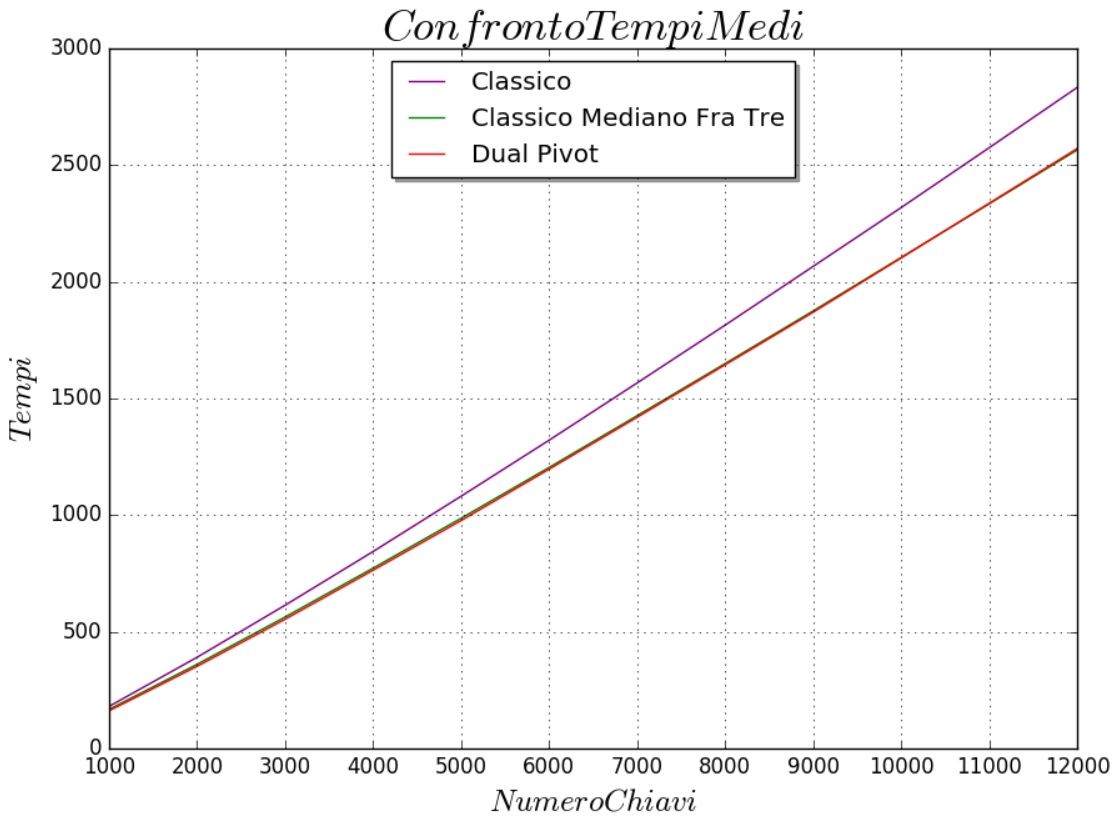

Figura 11.: Test di primo tipo con un alto numero di chiavi sulla prima macchina

Dato che abbiamo detto più volte che le prestazioni di Dual Pivot Quicksort sono dipendenti dall'architettura sottostante, abbiamo ipotiz-

zato che l'algoritmo non si stesse esprimendo al massimo sulla macchina utilizzata fino a quel momento.

- Ipotesi: Con una macchina più moderna, dotata di cache a più livelli e più capienti i risultati ritornino ad essere quelli attesi;
- Tesi: Ottenere risultati coerenti con le altre sperimentazioni presenti in bibliografia;

Il passo successivo allora è stato quello di procurarsi una macchina più al passo coi tempi, con il solito sistema operativo della precedente. Una volta trovata sono stati eseguiti alcuni test nelle stesse condizioni di quelli eseguiti in precedenza sull'altra macchina. La seconda macchina (avuta a disposizione per poco tempo) su cui sono stati testati gli algoritmi ha le seguenti caratterisitche:

- Processore Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 4096K

Come possiamo osservare tale macchina ha due cache di primo livello, mentre nella precedente ne era presente solo una, ed ha anche un livello in più. Facendo girare gli algoritmi con un test di tipo 1 sulla seconda macchina con l'obiettivo provante di ordinare 12000 chiavi (per prova con 1500 esecuzioni degli algoritmi per ognuna) i tempi ottenuti sono tornati ad essere conformi con quanto ci aspettavamo. Nella seguente tabella sono riportate le medie in microsecondi relativi alle prove appena descritte sulla seconda macchina:

Prova	Quicksort	Quicksort Mediano Fra Tre	Dual Pivot Quicksort
1	1585	1490	1442
2	1602	1546	1457
3	1614	1518	1471
4	1604	1574	1461
5	1615	1511	1468
6	1588	1485	1443
7	1597	1536	1450
8	1623	1594	1477

Concludiamo quindi affermando che le prestazioni di Dual Pivot Quicksort sono dipendenti da quelle offerte dalle memorie cache nelle architetture delle macchine di test.

6

CONCLUSIONI

Dual Pivot Quicksort indica una nuova strada da percorrere per studiare Quicksort, i cui miglioramenti sono difficili da realizzare perché è un già ottimo (e delicato) algoritmo. Lo studio del Multipivoting [6] [24] applicato a Quicksort sarà probabilmente di notevole importanza nella realizzazione di algoritmi ancor più efficienti. Nel corso della tesi è stato citato Three Pivot Quicksort [7] che risulta più veloce di Dual Pivot Quicksort per motivi relativi agli effetti dell'avere più perni che suddividano maggiormente l'istanza corrente in più sottoistanze. Ma un algoritmo come Three Pivot Quicksort è caratterizzato da un'implementazione estremamente complessa e siamo solo a tre pivot. Come detto nel corso della tesi esistono implementazioni che utilizzano anche più di tre perni ma che comunque sono più lente di Dual Pivot Quicksort, quindi la sfida per migliorare ulteriormente Quicksort è sicuramente in salita da un punto di vista di idee algoritmiche. Nella bibliografia è possibile trovare alcuni documenti che per grandi quantità di dati da ordinare parlano di implementazioni di Quicksort a 127 perni [6]. Chiaramente una possibile versione di Quicksort con un numero alto di perni deve essere supportata da un'implementazione che non solo funzioni in ogni caso ma che sfrutti l'architettura sottostante della macchina sulla quale è eseguito. In futuro è probabile che miglioramenti di Quicksort saranno dovuti all'avanzamento da un punto di vista hardware, ovvero a memorie cache sempre più capienti e veloci allo stesso tempo e ad architetture più avanzate sfruttabili maggiormente rispetto ad adesso. Infatti se venissero sviluppati Quicksort ancora più efficienti che sfruttino caratteristiche di architetture future, i risultati ottenuti oggi su macchine al passo con i tempi potrebbero non valere più.

A

CODICE SVILUPPATO

Il programma implementato per confrontare gli algoritmi è stato realizzato in linguaggio C per i motivi spiegati nel capitolo 5. Sono stati implementati vari moduli con l'idea della massima estendibilità possibile, in modo da aggiungere nuovi algoritmi da testare con facilità e/o anche nuovi test. Infatti ogni modulo ha un suo compito specifico ed è isolato più possibile dagli altri. Ogni modulo prima di essere integrato con il resto del programma corrente è stato testato a parte. Lo sviluppo del codice è stato caratterizzato da una scelta delle variabili con denominazioni più chiare possibili, in modo da permettere ad altri eventuali programmatori che volessero estendere il programma di capire bene il contesto e velocemente. Sempre a scopi di maggior chiarezza sono state separate più possibile la dichiarazione di variabili e dei prototipi delle funzioni dal resto del codice, infatti sono stati creati dei file.h (headers) con queste informazioni e file.c con il cuore delle funzionalità specifiche del modulo. Per la rilevazione dei tempi è stata usata la struttura *timeval* così caratterizzata:

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

Figura 12.: Contenuto della struttura timeval

In particolare per mostrare bene la differenza dei tempi d'esecuzione dei due algoritmi è stata utile suseconds_t tv_usec che permette la rilevazione dei tempi in microsecondi (10^{-6}).

INIZIALIZZAZIONE VETTORE CHIAVI PER I TEST

Il compito di inizializzare *chiavi* è delegato a ModuloPrincipale.c che fornisce quindi le funzioni per popolare il vettore in base al tipo di test scelto, e la basilare funzione di scambio per due chiavi i cui indici sono passati come argomento della funzione. I prototipi delle funzioni sono dichiarate in ModuloPrincipale.h insieme al vettore chiavi e alle strutture per la rilevazione dei tempi. L'implementazione delle funzioni è delegata a ModuloPrincipale.c (che include ModuloPrincipale.h).

ModuloPrincipale.h

```
# ifndef MODULO_PRINCIPALE
# define MODULO_PRINCIPALE

# include <stdio.h>
# include <sys/time.h>
# include <stdbool.h>
# include <stdlib.h>
# include <time.h>

# define NUMERO_CHIAVI // Da specificare prima del lancio del
    programma

int chiavi [ NUMERO_CHIAVI ];
struct timeval inizioAlgoritmo;
struct timeval fineAlgoritmo;

void popolaArrayChiavi(int []);
void inizializzazioneChiavi(int []);
void inizializzazioneChiaviRandom(int []);
void inizializzazioneChiaviOrdineInverso(int []);
void stampaArrayChiavi();
void permuta(int []);
void scambio(int [], int, int);

# endif
```

ModuloPrincipale.c

```
# include "ModuloPrincipale.h"

void inizializzazioneChiavi ( int chiavi [] ) {
  int i = 0;
  for ( i = 0; i < NUMERO_CHIAVI; i++ ) {
    chiavi [ i ] = i;
  }
}

void inizializzazioneChiaviRandom ( int chiavi [] ) {
  int i = 0;
  for ( i = 0; i < NUMERO_CHIAVI; i++ ) {
    chiavi [ i ] = rand () % NUMERO_CHIAVI;
  }
}

void inizializzazioneChiaviOrdineInverso ( int chiavi [] ) {
  int i = 0;
  for ( i = 0; i < NUMERO_CHIAVI; i++ ) {
    chiavi [ i ] = NUMERO_CHIAVI - 1 - i;
  }
}

void popolaArrayChiavi ( int chiavi [] ) {
  inizializzazioneChiavi( chiavi );
  permuta ( chiavi );
}

void permuta ( int chiavi [] ) {
   int i = 0;
   for ( i = NUMERO_CHIAVI - 1; i > 0; i-- ) {
      int j = rand () % ( i + 1 );
      scambio ( chiavi, i, j );
   }
}

void stampaArrayChiavi ( int chiavi [] ) {
  int i = 0;
  printf ( "Array delle chiavi\n" );
  for ( i = 0; i < NUMERO_CHIAVI; i++) {
    printf("Elemento %d --> %d\n ", i, chiavi [ i ]);
  }
```

```
}

void scambio ( int chiavi [], int i, int j ) {
   int app = chiavi [ i ];
   chiavi [ i ] = chiavi [ j ];
   chiavi [ j ] = app;
}
```

QUICKSORT

L'implementazione per Quicksort classico è stata realizzata scegliendo come perno la chiave più a sinistra della collezione. In questo caso occore una sentinella che arresti l'incremento dell'indice i.

QuickSortClassico.h

```
# ifndef QUICKSORT_CLASSICO
# define QUICKSORT_CLASSICO

void quicksort ( int chiavi [], int l, int r );

# endif
```

QuickSortClassico.c

```
# include "QuickSortClassico.h"

void quicksort ( int chiavi [], int l, int r ) {
   if (r <= l) {
     return;
   }
   int pivot = l;
   int i = l;
   int j = r;
      while ( i < j ) {
          while ( chiavi [ i ] <= chiavi [ pivot ] && i < r ) {
            i++;
          }
          while ( chiavi [ j ] > chiavi [ pivot ] ) {
            j--;
```

```
        }
        if ( i < j ) {
           scambio ( chiavi, i, j);
        }
     }
      scambio(chiavi, pivot, j);
      quicksort(chiavi, l, j - 1);
      quicksort(chiavi, j + 1, r);
  }
```

QUICKSORT DUAL PIVOT

Quicksort Dual Pivot con scelta dei perni come chiavi agli estremi della collezione, l'implementazione segue fedelmente lo pseudocodice mostrato nel capitolo 3.

QuickSortDualPivot.h

```
# ifndef QUICKSORT_DUAL_PIVOT
# define QUICKSORT_DUAL_PIVOT

void dualPivotQuickSort (int chiavi[], int left, int right);

# endif
```

QuicksortDualPivot.c

```
# include "QuickSortDualPivot.h"

void dualPivotQuickSort (int chiavi[], int left, int right) {
  if (right <= left) {
    return;
  }
  if (chiavi[left] > chiavi[right]) {
    scambio(chiavi, left, right);
  }
  int P1 = chiavi [left];
  int P2 = chiavi [right];
  int l = left + 1;
```

```
  int k = l;
  int g = right - 1;
  while ( k <= g ) {
    if (chiavi[k] < P1) {
      scambio(chiavi, k, l);
      l++;
    } else {
      if (chiavi[k] > P2) {
        while (chiavi[g] > P2 && k < g) {
          g = g - 1;
        }
        scambio(chiavi, k, g);
        g = g - 1;
        if (chiavi[k] < P1) {
          scambio(chiavi, k, l);
          l = l + 1;
        }
      }
    }
    k++;
  }
  l = l - 1;
  g = g + 1;
  scambio(chiavi, left, l);
  scambio(chiavi, right, g);
  dualPivotQuickSort (chiavi, left, l - 1);
  dualPivotQuickSort (chiavi, l + 1, g - 1);
  dualPivotQuickSort (chiavi, g + 1, right);
}
```

QUICKSORT CON SCELTA DEL PERNO FRA TRE CHIAVI

Quicksort con scelta del perno fra tre chiavi è stato implementato utilizzando tre funzioni: una per la scelta del perno, una per il partizionamento ed una che le richiama e si preoccupa di capire se l'istanza corrente sia caratterizzata da una sola chiave o meno.

QuickSortMedianoFraTre.h

```
# ifndef QUICKSORT_MEDIANOFRATRE
# define QUICKSORT_MEDIANOFRATRE
```

```
int partizionamento (int chiavi[], int l, int r);
void medianoFraTre (int chiavi[], int l, int r);
void sceltaPerno (int chiavi[], int l, int right);

# endif
```

QuickSortMedianoFraTre.c

```
# include "QuickSortMedianoFraTre.h"

void sceltaPerno (int chiavi[], int l, int r) {
  scambio(chiavi, (l + r)/2, r - 1);
  if ( chiavi [ r - 1 ] < chiavi [ l ] ) {
    scambio (chiavi, l, r - 1);
  }
  if ( chiavi [ r ] < chiavi [ l ] ) {
    scambio (chiavi, l, r);
  }
  if (chiavi [ r ] < chiavi [ r - 1 ] ) {
    scambio (chiavi, r - 1, r);
  }
}

int partizionamento ( int chiavi[], int l, int r) {
  int i = l;
  int j = r - 1;
  while (true) {
    while (chiavi[i] < chiavi[r]) {
      i++;
    }
    while (chiavi[r] < chiavi[j] && i < j) {
      j--;
    }
    if (i >= j) {
      break;
    }
    scambio(chiavi, i, j);
    i++;
    j--;
```

```
  }
    scambio(chiavi, i, r);
    return i;
}

void medianoFraTre (int chiavi[], int l, int r) {
 int i = 0;
 if (r <= l) {
  return;
 }
 sceltaPerno (chiavi, l, r);
 i = partizionamento(chiavi, l + 1, r - 1);
 medianoFraTre(chiavi, l, i - 1);
 medianoFraTre(chiavi, i + 1, r);
}
```

SVOLGIMENTO DEI TEST

Il compito di svolgere i test veri e propri è delegato a Test.c, il quale è dotato di un *int main()*, questo modulo include l'implementazione del Quicksort Classico e di Quicksort Dual Pivot fornite rispettivamente da Quicksort.c e QuicksortDualPivot.c (che include con la direttiva #include). Questo modulo include anche una funzione che effettua una copia di *chiavi* in modo che la successione di elementi generati non vada persa dopo il primo ordinamento da parte di Quicksort. La copia è dunque utilizzata per effettuare lo stesso test delle chiavi anche con Quicksort con scelta del perno fra tre chiavi e poi riutilizzando la stessa funzione viene resa disponibile la collezione di chiavi da ordinare per Quicksort Dual Pivot. Inoltre questo modulo ne richiama un altro per raccogliere i dati prodotti dalle esecuzioni degli algoritmi e che le va a scrivere su dei file. Oltre che per raccogliere i dati, i file sono utilizzati per delegare a Python la responsabilità di fare i grafici, infatti tali dati sono letti da moduli scritti in Python che mostrano i risultati ottenuti. I test sono ripetibili un numero arbitrario di volte (indicato dall'utente al lancio dell'applicazione da terminale) e i dati prodotti sono memorizzati in appositi vettori (durante l'esecuzione degli algoritmi) prima di essere scritti nei file. È stato scelto di scartare i primi 50 test di ogni esecuzione del programma perché al momento del lancio dell'eseguibile da Linux (impostato a riga di comando), iniziava l'esecuzione di un processo che durava alcuni microsecondi che falsava i tempi rilevati per i tre algoritmi.

Test.h

```
# ifndef TEST
# define TEST

# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <string.h>
# include <unistd.h>
# include "ModuloPrincipale.c"
# include "AnalisiStatistiche.c"
# include "GestoreDati.c"
# include "InsertionSort.c"
# include "QuickSortClassico.c"
# include "QuickSortMedianoFraTre.c"
# include "QuickSortDualPivot.c"

void conservaArrayChiavi (int [], int []);
void messaggioErroreLancioApplicazione ();

# endif
```

Test.c

```
# include "Test.h"

void messaggioErroreLancioApplicazione () {
  printf("Errore sul lancio dell'applicazione, sintassi: <nomeEseguibile>
      <numeroTipoDiTest> <numeroTest>\n");
  exit(EXIT_FAILURE);
}

void conservaArrayChiavi(int chiavi [], int chiavi2 []) {
   int i = 0;
   for ( i = 0; i < NUMERO_CHIAVI; i++ ) {
     chiavi2 [ i ] = chiavi [ i ];
   }
}

int main ( int argc, char* argv[] ) {
```

```
if (argc >= 4 || argc == 1 || argc == 2) {
 messaggioErroreLancioApplicazione();
}

int numeroTest = atoi(argv[2]);
int tempoEsecuzione = 0;
int numeroTipoDiTest = atoi(argv[1]);
int chiavi2 [ NUMERO_CHIAVI ];
int tempiTestClassico [ numeroTest - 50];
int tempiTestDualPivot [ numeroTest - 50];
int tempiTestMedianoFraTre [ numeroTest - 50];
int j = 0;
int k = 0;
int h = 0;

if (numeroTipoDiTest < 1 || numeroTipoDiTest > 4 ) {
 messaggioErroreLancioApplicazione();
}

int i = 0;

for ( i = 0; i < numeroTest; i++ ) {

switch (numeroTipoDiTest) {

case 1:
      popolaArrayChiavi(chiavi);
      break;
case 2:
      inizializzazioneChiaviRandom(chiavi);
      break;
case 3:
      inizializzazioneChiavi(chiavi);
      break;
case 4:
      inizializzazioneChiaviOrdineInverso(chiavi);
      break;
}

printf("TEST NUMERO: %d\n", i);
```

```
conservaArrayChiavi ( chiavi, chiavi2 );

gettimeofday(&inizioAlgoritmo, NULL);

quicksort(chiavi, 0, NUMERO_CHIAVI - 1);

gettimeofday(&fineAlgoritmo, NULL);

tempoEsecuzione = ((fineAlgoritmo.tv_sec - inizioAlgoritmo.tv_sec)
    * 1000000L + fineAlgoritmo.tv_usec) - inizioAlgoritmo.tv_usec;

if ( i > 49 ) {
 tempiTestClassico [ j ] = tempoEsecuzione;
 j++;
}

printf("Tempo in microsecondi ( QuickSort Classico ): %d
    microsecondi\n", tempoEsecuzione);

conservaArrayChiavi ( chiavi2, chiavi );

gettimeofday(&inizioAlgoritmo, NULL);

medianoFraTre(chiavi2, 0, NUMERO_CHIAVI - 1);

gettimeofday(&fineAlgoritmo, NULL);

tempoEsecuzione = ((fineAlgoritmo.tv_sec - inizioAlgoritmo.tv_sec)
    * 1000000L + fineAlgoritmo.tv_usec) - inizioAlgoritmo.tv_usec;

if ( i > 49 ) {
 tempiTestMedianoFraTre [ k ] = tempoEsecuzione;
 k++;
}

printf ( "Tempo in microsecondi ( QuickSort Classico M3 ): %d
    microsecondi\n", tempoEsecuzione);

gettimeofday(&inizioAlgoritmo, NULL);

dualPivotQuickSort(chiavi, 0, NUMERO_CHIAVI - 1);
```

```
gettimeofday(&fineAlgoritmo, NULL);

tempoEsecuzione = ((fineAlgoritmo.tv_sec - inizioAlgoritmo.tv_sec)
    * 1000000L + fineAlgoritmo.tv_usec) - inizioAlgoritmo.tv_usec;

if ( i > 49) {
  tempiTestDualPivot [ h ] = tempoEsecuzione;
  h++;
}
printf ( "Tempo in microsecondi ( QuickSort Dual - Pivot ): %d
    microsecondi\n", tempoEsecuzione);
}
printf("\n");

// Statistiche Quicksort Classico
float mediaClassico = calcoloMedia(tempiTestClassico, numeroTest);

printf("mediaClassico: %f\n", mediaClassico);
printf("\n");

// Statistiche Quicksort Classico M3
float mediaClassicoM3 = calcoloMedia(tempiTestMedianoFraTre,
    numeroTest);

printf("mediaClassicoM3: %f\n", mediaClassicoM3);
printf("\n");

// Statistiche Quicksort Dual Pivot
float mediaDualPivot = calcoloMedia(tempiTestDualPivot,
    numeroTest);

printf("mediaDualPivot: %f\n", mediaDualPivot);

// Registrazione dati
gestioneDatiMultipli("tempiTestClassico.txt", tempiTestClassico,
    numeroTest);
gestioneDatiMultipli("tempiTestMedianoFraTre.txt",
    tempiTestMedianoFraTre, numeroTest);
gestioneDatiMultipli("tempiTestDualPivot.txt", tempiTestDualPivot,
    numeroTest);

gestioneDatiSingoli("medieClassico.txt", mediaClassico, true);
```

```
  gestioneDatiSingoli("medieMedianoFraTre.txt", mediaClassicoM3,
      true);
  gestioneDatiSingoli("medieDualPivot.txt", mediaDualPivot, true);
  gestioneDatiSingoli("numProve.txt", numeroTest, false);
  exit(EXIT_SUCCESS);
}
```

INSERTION SORT

Tipica implementazione di Insertion Sort.

InsertionSort.h

```
# ifndef INSERTION_SORT
# define INSERTION_SORT

void InsertionSort (int [], int l, int r);

# endif
```

InsertionSort.c

```
# include "InsertionSort.h"

void InsertionSort (int chiavi [], int l, int r) {
  int i = 0;
  int j = 0;
  for ( i = l + 1; i <= r; i ++ ) {
    for ( j = i; j > l && chiavi [ j ] < chiavi [ j - 1 ]; j-- ) {
      scambio ( chiavi, j, j - 1 );
     }
  }
  return;
}
```

GESTIONE DEI DATI

Allo scopo di separare la responsabilità di testare gli algoritmi e quella di consevare i dati raccolti durante le sperimentazioni, sono stati implementati un file.h ed un file.c che assumessero tale compito. In particolare sono state implementate due funzioni, una che gestisce la scrittura di dati singoli come per esempio le medie. Ed un'altra che gestisce la scrittura di dati multipli, come i tempi rilevati dalla sperimentazione degli algoritmi.

GestoreDati.h

```
# ifndef GESTORE_DATI
# define GESTORE_DATI

void gestioneDatiSingoli (char* nomeFile, float dato, bool flag);
void gestioneDatiMultipli (char* nomeFile, int tempiTest [], int
    numeroTest);

# endif
```

GestoreDati.c

```
# include "GestoreDati.h"

void gestioneDatiSingoli (char* nomeFile, float dato, bool flag) {
  FILE *p = NULL;
  if (flag == true) {
    p = fopen(nomeFile, "a");
  } else {
    p = fopen(nomeFile, "w");
  }
  fprintf(p, "%f ", dato);
  fclose(p);
}

void gestioneDatiMultipli (char* nomeFile, int tempiTest [], int
    numeroTest) {
  FILE *p = NULL;
  p = fopen(nomeFile, "w");
  int i = 0;
```

```
  for ( i = 0; i < numeroTest - 50; i++) {
    fprintf(p, "%d ", tempiTest[i]);
  }
  fclose(p);
}
```

MEDIE E STATISTICHE

Le medie relative ai tempi degli algoritmi sono state delegate ad un modulo specifico che si occupa di fare solo quello, inoltre è stata implementata una funzione che calcola la varianza della distribuzione dei tempi ottenuta. Quest'ultima non è stata utilizzata nella sperimentazione attuale, bensì potrà servire a condurre statistiche in futuro, in caso di ulteriori sperimentazioni.

AnalisiStatistiche.h

```
# ifndef ANALISI_STATISTICHE
# define ANALISI_STATISTICHE

# include <stdio.h>
# include <stdlib.h>
# include <stdbool.h>

float calcoloMedia (int tempi[], int numeroTest);
float calcoloVarianza (float media, int tempi[], int numeroTest,
    bool campionaria);

#endif
```

AnalisiStatistiche.c

```
float calcoloMedia (int tempi[], int numeroTest) {
  int i = 0;
  float somma = 0;
  for ( i = 0; i < numeroTest - 50; i++ ) {
    somma = tempi [ i ] + somma;

  }
```

```
  float media = somma / (numeroTest - 50);
  return media;
}

float calcoloVarianza ( float media, int tempi[], int numeroTest,
    bool campionaria) {
  int i = 0;
  float dVarianza = 0;

  for ( i = 0; i < numeroTest - 50; i++ ) {
    dVarianza = ( tempi [ i ] - media ) * ( tempi [ i ] - media ) +
        dVarianza;
  }
  float varianza = 0;
  if (campionaria == true) {
    varianza = dVarianza / (numeroTest - 50 - 1);
  } else {
    varianza = dVarianza / numeroTest - 50;
  }
  return varianza;
}
```

Il codice presentato in questa appendice non è la totalità del codice sviluppato durante lo studio di Dual Pivot Quicksort, e nella tesi non sono stati inseriti i tempi salvati nei file durante l'esecuzione degli algoritmi per motivi di spazio. Per visualizzare tutto il materiale prodotto in questa esperienza (compreso il codice Python per la realizzazione dei grafici) è consigliato visitare [29].

BIBLIOGRAFIA

[1] Martin Aumuller, Martin Dietzfelbinger e Pascal Klaue - *How Good is Multi-Pivot Quicksort?* - http://eiche.theoinf.tu-ilmenau.de/quicksort-experiments/uploads/multi-pivot-paper-draft-2015-10-12.pdf

[2] GitBook - *Dual Pivot Quicksort* - https://thebillkidy.gitbooks.io/ugent-algoritmen-1/content/17_dual_pivot_quicksort.html

[3] Stijn Eyerman, James E.Smith, Lieven Eeckhout - *Characterizing the Branch Miss Prediction Penalty* - http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1620789

[4] Vasileios Iliopoulos and David B.Penman -*Dual Pivot Quicksort* - http://repository.essex.ac.uk/13268/1/dual%20pivot%20Quicksort-postprint.pdf

[5] Kanela Kaligosi, Peter Sanders - *How Branch Mispredictions Affect Quicksort* - http://algo2.iti.kit.edu/sanders/papers/KalSan06.pdf (Cited on pages 34 and 35.)

[6] Shrinu Kushagra, Alejandro López-Ortiz, J.Ian Munro, Aurick Qiao - *Multi-Pivot Quicksort: Theory and Experiments* - http://epubs.siam.org/doi/pdf/10.1137/1.9781611973198.6 (Cited on pages 14, 26, 29, 30, 31, 40, and 51.)

[7] Somshubra Majumdar, Ishaan Jain e Aruna Gawade - *Parallel Quick Sort using Thread Pool Pattern* - http://www.ijcaonline.org/research/volume136/number7/majumdar-2016-ijca-908495.pdf (Cited on page 51.)

[8] rerun - *Quicksorting 3-Way and Dual Pivot* - http://rerun.me/2013/06/13/quicksorting-3-way-and-dual-pivot/

[9] Jonas Schiffl - *Dual Pivot Quicksort: Verification and Proof using KeY* - http://i12www.ira.uka.de/~key/keysymposium16/slides/Jonas_Schiffl-Dual_Pivot_Quicksort.pdf

[10] Sebastian Wild, Markus E. Nebel, Ralph Neininger - *Average Case and Distributional Analysis of Dual-Pivot Quicksort* - https://arxiv.org/pdf/1304.0988.pdf

[11] Sebastian Wild, Markus E. Nebel, Conrado Martinez - *Analysis of Pivot Sampling in Dual-Pivot Quicksort* - https://arxiv.org/pdf/1412.0193.pdf (Cited on page 27.)

[12] Sebastian Wild, Markus E. Nebel e Hosam Mahmoud - *Analysis of Quickselect under Yaroslavskiy's Dual-Pivoting Algorithm* - http://wwwagak.cs.uni-kl.de/downloads/papers/Analysis_of_Quickselect_under_Yaroslavskiys_Dual-Pivoting_Algorithm.pdf

[13] Sebastian Wild, Markus E. Nebel - *Average Case Analysis of Java 7's Dual Pivot Quicksort* - https://ai2-s2-pdfs.s3.amazonaws.com/b76b/b328076dbc9340e57360eff63627343c87a9.pdf

[14] Sebastian Wild, Markus Nebel, Raphael Reitzig e Ulrich Laube - *Engineering Java 7's Dual Pivot Quicksort Using MaLiJAn* - https://ai2-s2-pdfs.s3.amazonaws.com/11e6/3739f033fdf97dd832dbc210ac7a21139647.pdf

[15] Vladimir Yaroslavskiy - *Dual Pivot Quicksort algorithm* - http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf (Cited on pages 7, 15, 19, and 45.)

[16] Wikipedia, l'enciclopedia libera - *Algoritmo di ordinamento* - https://it.wikipedia.org/wiki/Algoritmo_di_ordinamento (Cited on pages 7, 8, and 14.)

[17] Wikipedia, l'enciclopedia libera - *Divide et Impera* - https://it.wikipedia.org/wiki/Divide_et_impera (Cited on page 11.)

[18] Fisher-Yates - *Wikipedia, the free Encyclopedia* - https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle (Cited on page 38.)

[19] Wikipedia, l'enciclopedia libera - *Insertion sort* - https://it.wikipedia.org/wiki/Insertion_sort (Cited on page 15.)

[20] Wikipedia, l'enciclopedia libera - *Tony Hoare* - https://it.wikipedia.org/wiki/Tony_Hoare (Cited on page 7.)

[21] Wikipedia, the Free Encyclopedia - *Quicksort* - *https://en.wikipedia.org/wiki/Quicksort* (Cited on page 7.)

[22] Wikipedia, l'enciclopedia libera - *Quicksort* - *https://it.wikipedia.org/wiki/Quicksort* (Cited on pages 7 and 8.)

[23] David A. Patterson, John L.Hennessy - *Computer Organization and Design* (Cited on pages 29, 30, 31, and 32.)

[24] M. Cecilia Verri - *L'Ordinamento di Dati* - *https://e-l.unifi.it/pluginfile.php/28544/mod_resource/content/2/4%20Ordinamento.pdf* (Cited on pages 8 and 51.)

[25] M. Cecilia Verri - *Quicksort, Mergesort e Heapsort* - *https://e-l.unifi.it/pluginfile.php/28545/mod_resource/content/3/5%20QuickMergeHeapSort.pdf* (Cited on page 16.)

[26] Oracle Java 7 API - *Class Arrays* - *https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html*

[27] Oracle Java 8 API - *Class Arrays* - *https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html* (Cited on pages 9 and 27.)

[28] pkqs - *On the Analysis of Two Fundamental Randomized Algorithms* - *http://pkqs.net/~tre/aumueller_dissertation.pdf* (Cited on page 7.)

[29] GitHub - *TesiTriennale* - *https://github.com/nobleVine/TesiTriennale* (Cited on page 68.)

RINGRAZIAMENTI

Ci tengo a ringraziare la mia relatrice, Maria Cecilia Verri per la disponibilità e l'indispensabile collaborazione per la stesura della tesi. Ringrazio il mio compagno Michele De Vita per aver messo a disposizione il suo computer per i test degli algoritmi. Inoltre, un particolare ringraziamento va a: Lorenzo Marino, Giulio Grimani e Manuel Ronca per la loro amicizia fraterna e per il loro contributo al mio percorso di studi. Infine, per ultimo ma primo in ordine d'importanza ci tengo molto a ringraziare i miei genitori per il supporto psicologico ed economico che mi hanno dato durante tutto il percorso di studi.