Università Ca' Foscari - Venezia
Corso di Laurea in Informatica (Specialistica)

Building in Parallel a Term-Partitioned Global Inverted File Index

Documentazione del progetto di Laboratorio di Calcolo Parallelo
AA 2005/2006

Possamai Lino, Tonin Mirco

Indice

Capitolo 1

Introduzione

Sempre più spesso, i motori di ricerca sono considerati gli hub di Internet, gli aggregatori. Assieme alla loro importanza, ultimamente, sta crescendo l'attenzione che viene dedicata all'ottimizzazione ed alla creazione di motori performanti. Negli ultimi anni, il basso costo che si registra delle singole unità di elaborazione e la loro larga diffusione, ha fatto si che molti problemi siano affrontati utilizzando una tecnica distribuita, aumentando in questo modo la velocità di elaborazione, ma soprattutto la capacità di elaborare un numero crescente di dati, cioè un aumento del throughput.

Oltre a numerosi problemi di tipo scientifico Grand Challenge, anche i problemi dell'indicizzazione dei documenti ipertestuali possono essere definiti in termini di elaborazione parallela. Data la crescita esponenziale che si è avuta negli ultimi anni del numero di pagine pubblicate nel web, probabilmente si potrebbe affermare con certezza, che questa soluzione sia l'unica adottabile, se tempo e costi sono parametri rilevanti.

Con questo lavoro, si affronterà in dettaglio una possibile implementazione, del back-end di un motore di ricerca e precisamente la costruzione in parallelo di un indice globale partizionato in base ai termini (term partitioning).

Il resto della relazione è organizzato come segue. Nel capitolo 2, verrà presentata una panoramica più approffondita del problema in questione. Nel capitolo 3 saranno classfificati e descritti gli algoritmi che sono stati utilizzati per l'ordinamento, esterno ed interno, degli elementi. L'architettura e le scelte implementative, comprese le tecniche di comunicazione adottate, che sono state fatte si possono trovare nel capitolo 4. I risultati che sono stati ottenuti dopo aver fatto i test delle prestazioni sono elencati dettagliatamente nel capitolo 5. L'appendice A contiene una descrizione degli script che sono stati utilizzati durante tutta la fase di testing. L'appendice B raccoglie tutti i grafici che sono stati prodotti.

Capitolo 2

Descrizione del progetto

Ciò che sta alla base di un motore di ricerca, è l'indice. L'indice è quella struttura dati che permette ad un web search engine di restituire i risultati di una query il più velocemente possibile. Per formalizzare questo concetto, se supponiamo di avere M documenti (d'ora in poi identificati da $D_j \in \mathcal{D}$), che possono contenere N termini distinti (li denoteremo con $T_i \in \mathcal{T}$), allora possiamo vedere un documento come una funzione $f : \mathcal{D} \to \mathcal{T}$ definita da $D_j \mapsto T_i$ che mappa elementi dell'insieme dei documenti in quello dei termini.

L'indice, rappresenterà di conseguenza, la funzione $g : \mathcal{T} \to \mathcal{D}$ definita da $T_i \mapsto D_j$.

Per concretizzare ciò che viene fatto per costruire una tale funzione g, sono necessari i seguenti passi:

- *Parsing* di ogni documento $D_j \in \mathcal{D}$. Il parsing consiste nella costruzione di liste di coppie (T_i, D_j). Questo approccio rappresenta una versione semplificata, rispetto alla realtà, nel qualcaso vengono considerati altri parametri come per esempio il numero di occorrenze e l'offset, del termine, dalla testa del documento.

- *Sorting* di ogni coppia in modo tale che questo tipo di ordinamento sia definito: $(T_i, D_j) \leq (T_{i+1}, D_{j+1}) \Rightarrow T_i \leq T_{i+1}$ e $D_j \leq D_{j+1}$ per ogni i e j.

- Creazione, date le coppie precedentemente create e ordinate, di una lista di coppie $(T_i, \mathcal{P}_j)$ con $\mathcal{P}_j \subset \mathcal{D}$. $\mathcal{P}_j$ saranno chiamate *posting lists*.

Il discorso fatto fin'ora rappresenta la **procedura sequenziale** nel caso in cui non avessimo a disposizione un'architettura parallela, come per esempio un cluster di workstations (COW). A questo proposito, se vogliamo effettuare un processo di parallelizzazione, potrebbe essere utile il modello di Foster[10], anche se in forma ridotta, dato che non tutti i punti proposti possono essere applicati.

La prima parte considera una possibile **partizione** del problema. Nel nostro caso, partizionare vuol dire assegnare ad ogni nodo a disposizione un sottoinsieme di documenti $\mathcal{P}_j$, in modo tale che $\bigcup_j \mathcal{P}_j = \mathcal{D}$. Questo tipo di partizionamento è detto *verticale* e, sebbene più semplice nell'implementazione rispetto ad altri partizionamenti, al momento della produzione dei risultati di una query, necessita la consultazione di più nodi. Viceversa, come per i documenti, anche per i termini è possibile effettuare un partizionamento, detto *orizzontale*, in modo tale che ogni nodo abbia la gestione di un sottoinsieme di termini. In tutti e due i casi, in ogni nodo si dovranno effettuare le operazioni di *parsing* e *inversion*.

Per entrare più nello specifico del progetto in questione, è stato scelto un partizionamento orizzontale, che si può vedere come un partizionamento rispetto ai dati in output. Data questa scelta, è possibile assegnare del lavoro ai nodi in termini di sottoinsiemi di $\mathcal{T}$. D'ora in poi, il sottoinsieme di termini che viene assegnato ad ogni nodo sarà $\mathcal{T}^1, \ldots, \mathcal{T}^p$ (con p il numero di nodi disponibile) tale che $\bigcup_i \mathcal{T}^i = \mathcal{T}$.

Per poter creare più agilmente le posting lists, abbiamo bisogno di ordinare le coppie presenti in ogni nodo, secondo l'ordinamento che è stato definito in precedenza.

Dati gli ordini di grandezza della quantità di dati da elaborare, le soluzioni che utilizzano solo ed esclusivamente la memoria (*ordinamento interno*), benchè apprezzabili per la loro velocità di accesso ai dati, non sono adatte perché per il loro funzionamento sono necessarie grandi quantità di memoria. Nella nostra situazione, abbiamo dati memorizzati in grandi file e un tipico metodo di *ordinamento esterno* è il *sort-merge*[7], utilizzato molto spesso nel campo dei database, che cerca di minimizzare il numero di accessi ai file.

Il sort-merge ordering si basa nei seguenti step:

1. Prende un file temporaneo in cui ci sono le coppie disordinate;
2. Utilizza una zona di memoria primaria di dimensione k in cui memorizzare temporaneamente delle coppie;
3. Il primo step consiste nel prendere dal file temporaneo k coppie, portarle in memoria, ordinarle con l'algoritmo di ordinamento interno quicksort e riportare il blocco ordinato nel file;
4. Alla fine della procedura precedente, ci troveremo con un file in cui ci sono blocchi di coppie che sono ordinate;
5. Il passo successivo consiste nel prendere **sempre** coppie di blocchi ma ogni volta di dimensione doppia e, avendo a disposizione due puntatori che riferiscono all'inizio dei blocchi, effettuare un confronto elemento per elemento e scrivere in un nuovo file temporaneo il risultato. Questo step deve essere ripetuto $log(n)$ volte con $n = \lceil \mathcal{C}/k \rceil$ e $\mathcal{C}$ numero di coppie in ogni nodo. Nell'ultimo step, prima di ottenere un file completamente ordinato, dovremo ordinare due blocchi che contengono rispettivamente $\mathcal{C}/2$ elementi. Da questo si può dedurre che ad ogni step, viene creato un nuovo file temporaneo della stessa dimensione del file temporaneo originario.

Le osservazioni sul tipo di partizionamento, induce a pensare a quale possa essere la fase successiva del modello di Foster, e precisamente la **comunicazione**. Ogni nodo dovrà farsi carico di spedire la parte di dati che non deve gestire, al nodo di competenza. Questo è il punto in cui saranno utilizzare diverse tecniche di comunicazione al fine di analizzare le prestazioni che da ognuna ne deriva.

Ora, ogni nodo conterrà delle liste ordinate di coppie che sono state spedite dai nodi del cluster, nonchè la lista di coppie che sono state generate localmente. L'obiettivo è quello di avere in ogni nodo le proprie posting lists. Quest'ultima operazione viene fatta grazie all'algoritmo *multiway-merge*. Questo algoritmo semplicemente sfrutta una struttura a coda con priorità, di dimensione p. La coda viene inizialmente riempita con gli elementi che stanno in testa alle liste di coppie. Quando la coda è piena, viene tolto l'elemento più piccolo (qui rappresentato dalla coppia con T_i inferiore). Il nuovo candidato da inserire nella coda dovrà essere cercato negli elementi che stanno in testa alle liste e sarà quello che avrà il T_i più piccolo.

2.1 Assunzioni

Dopo una presentazione generale del problema, è necessario presentare le scelte implementative nonchè le semplificazioni adottate nel progetto in questione.

- Si presuppone che ogni nodo abbia già a disposizione un insieme di coppie (T_i, D_j), create secondo una distribuzione uniforme. Ogni nodo potrà quindi generare dei termini che non deve gestire direttamente. L'insieme dei termini che creerà sarà indicato come $\mathcal{G}^1, \ldots, \mathcal{G}^p$.
- I T_i presenti in ogni nodo sono globali al cluster, ovvero termini con identificativo uguale in nodi diversi, denotano termini uguali.
- I D_j sono locali ad ogni nodo (e questo implica che una qualche forma di comunicazione è necessaria per far si che i D_j diventino globali).
- Le posting lists conterranno solamente sottoinsiemi di $\mathcal{D}$ e vengono create dopo la fase di merging (multiway).
- La cardinalità di $\mathcal{T}$, $\mathcal{D}$ e $\mathcal{C}$ (numero di coppie da generare in ogni nodo) è fissata staticamente all'inizio dell'esecuzione del programma ed è pari a MAX_TERM, MAX_DOC e MAX_CP rispettivamente.

Capitolo 3

Ordinamento

Nel progetto sono stati implementati i più utilizzati algoritmi di ordinamento interno (quicksort, mergesort e heapsort) ed esterno (sort-based)[9],[12],[1]. Essi sono classificati anche in base all'utilizzo o meno di memoria ausiliaria e alla loro stabilità[5],[2]. Effettueremo una piccola descrizione delle caratteristiche di ogni algoritmo di ordinamento interno, e ci soffermeremo maggiormente sull'algoritmo sort-based. Vedremo in dettaglio, come avviene la fusione di due runs del file e le due strategie di implementazione della fase di merging. Con scopo puramente didattico, emuleremo in memoria il funzionamento dell'algoritmo esterno *sort-based* attraverso il sort-based simulato.

In maniera del tutto informale, possiamo definire l'ordinamento, quell'algoritmo che viene utilizzato per ordinare (in modo crescente o decrescente) gli elementi di un dato insieme.

Definizione formale: È un algoritmo che preso in:

- **input** una sequenza di n numeri $< a_1, a_2, a_3, \ldots, a_{n-1}, a_n >$,
- restituisce in **output** una permutazione (riordinamento) $< a'_1, a'_2, a'_3, \ldots, a'_{n-1}, a'_n >$ della sequenza in input tale che: $a'_1 \leq a'_2 \leq \ldots \leq a'_{n-1} \leq a'_n$

3.1 Classificazione degli algoritmi

Gli algoritmi di ordinamento sono classificati a seconda:

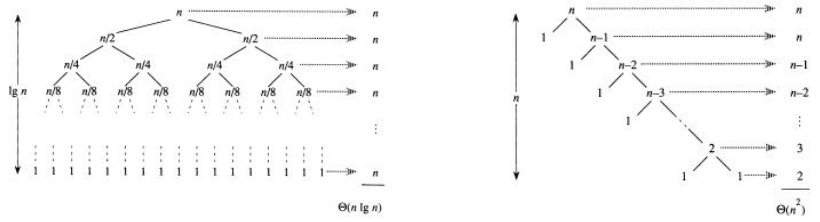
- metodo interno/esterno: se l'insieme degli elementi da ordinare può essere totalmente contenuto in memoria, il metodo viene detto *interno*, altrimenti *esterno*.
  - ordinamento interno: è possibile accedere direttamente ad un qualsiasi record (vedi array). Ne fanno parte gli algoritmi quicksort, mergesort e heapsort.
  - ordinamento esterno: i record devono essere acceduti in modo sequenziale. Esempi di algoritmi che appartengono a questa categoria sono il *sort-based inversion* ed il *sort-based inversion simulato*.
- richiesta di memoria ausiliaria: alcuni algoritmi fanno uso di ulteriore memoria rispetto a quella usata per contenere gli elementi da ordinare: - sort-based 'inversion' - 'sort-based 'inversion' simulato' - mergesort

- stabilità. Un metodo di ordinamento si dice stabile se preserva l'ordine relativo ai dati con chiavi uguali all'interno del file da ordinare. Traducendo il tutto con un esempio: se si ordina per anno di corso una lista di studenti già ordinata alfabeticamente, un metodo stabile produce una lista in cui gli alunni dello stesso anno sono ancora in ordine alfabetico, mentre un ordinamento instabile probabilmente produrrà una lista senza più alcuna traccia del precedentemente ordinamento. Gli ordinamenti da noi implementati non sono stabili: coppie uguali possono scambiarsi di posto.

3.2 Analisi degli algoritmi

3.2.1 Ordinamento interno: Quicksort

È un algoritmo di ordinamento interno non stabile che non necessita di memoria ausiliaria. Dato n il numero di elementi da ordinare, la complessità asintotica sarà pari, nel caso peggiore, a $O(n^2)$ (ottenuto quando l'insieme dei dati in input è già ordinato), mentre nel caso migliore e medio è pari a $O(nlog(n))$ (vedi figura 3.1).



(a) (b)

Figura 3.1: (a) Caso migliore: la procedura *partition* bilancia sempre i due lati della partizione in modo uguale. (b) Caso peggiore: albero di ricorrenza in cui la procedura *partition* mette sempre e soltanto un elemento su un lato della partizione. Si ottiene ciò ogni qual volta l'insieme è già ordinato.

Anche se la complessità asintotica nel caso medio è identica all'heapsort, cioè $O(nlog(n))$, l'ordinamento è più veloce [4]. Grazie a ciò, è molto popolare per ordinare grandi array di input. Il quicksort, come il merge sort, è basato sul paradigma **divide-et-impera**.

Divide. L'array $A[p \dots r]$ è ripartito in due sottoarray non vuoti $A[p \dots q]$ e $A[q+1 \dots r]$ in modo tale che ogni elemento di $A[p \dots q]$ sia minore o uguale a qualunque elemento di $A[q+1 \dots r]$.

Impera. I due sottoarray $A[p \dots q]$ e $A[q+1 \dots r]$ sono ordinati con chiamate ricorsive a quicksort.

Ricombina. Poichè i due sottoarray sono ordinati in loco, non è richiesto alcuno sforzo per ricombinarli, l'intero array $A[p \dots r]$ è già ordinato.

3.2.2 Ordinamento interno: Mergesort

Algoritmo di ordinamento interno non stabile, che necessita di memoria ausiliaria pari a $O(n)$. In virtù del fatto che, ad ogni passo l'insieme degli array viene equipartizionato, la complessità

asintotica sarà sempre pari a $O(nlog(n))$ sia nel caso peggiore, sia nel caso medio che nel caso migliore [3],[6].

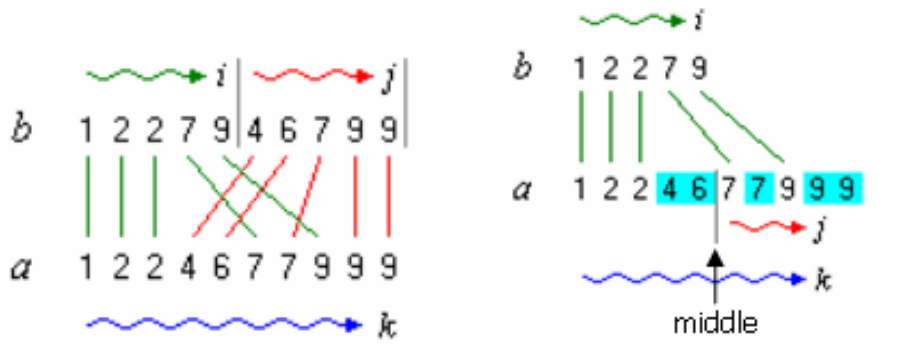
Esegue in modo stretto il paradigma **divede-et-impera**:

Divide: divide gli n elementi della sequenza da ordinare in due sottosequenze di $n/2$ elementi ciascuna.

Impera: ordina le due sottosequenze usando ricorsivamente il merge sort.

Combina: fonde le due sottosequenze per produrre in output la sequenza ordinata.

Esistono differenti implementazioni (più o meno furbe) della fase di conquer. Una delle quali si basa su una implementazione classica e intuitiva (v 1.0, vedi figura 3.2 (a)): utilizzo di una struttura ausiliaria b di grandezza pari all'array a contenente gli elementi da ordinare. Alternativamente si può far meglio sia in termini di performance che di spazio di memoria utilizato. Il progetto implementa la fase di conquer del mergesort (v 2.0) come è presentato in figura 3.2 (b).



(a) (b)

Figura 3.2: (a) Setup: copia di a in b. Le due metà di b sono scandite rispettivamente dai puntatori i e j; il più piccolo elemento puntato, viene ricopiato in a. (b) Si copi in b solo la prima metà di a mentre i restanti elementi rimangono in a. I principali vantaggi di queso approccio sono lo spazio di memoria ausiliario ridotto a $O(n/2)$ e uno speedup nell'operazione di copia pari a 2.

Perché funziona? Se copio un elemento di j prima di $middle$, allora la sua posizione originale dovrà essere riempita con un elemento di i in quanto il numero totale di elementi è costante.

L'algoritmo 1 rappresenta lo pseudo codice del merge.

Quali sono le condizioni di terminazione?

- $k = j$ a partire da j incluso, sono tutti elementi maggiori di quelli già ordinati; poichè ogni partizione al suo interno è già ordinata, allora abbiamo concluso.

- $j > hi$ gli elementi a partire da $(middle+1)$ a j escluso sono stati copiati nella prima metà di a. Le posizioni originali, dovranno essere occupate da elementi appartenenti alla prima partizione: ultimo while (ricordiamoci che ogni parzione è già al suo interno ordinata).

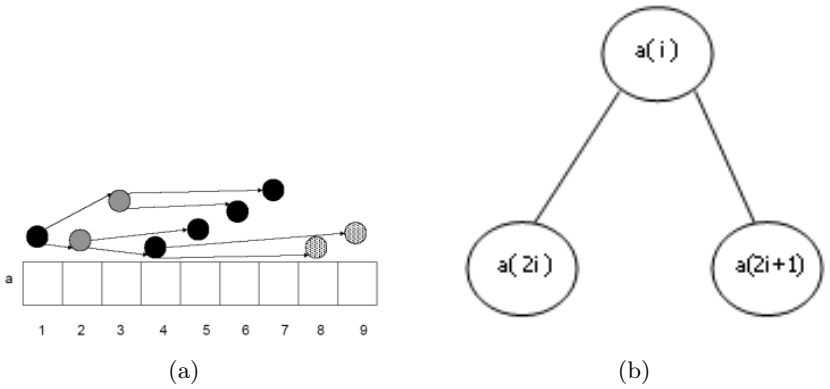
Algorithm 1 Merge(**int** lo, **int** m, **int** hi)

```
 1: int i,j,k;
 2: i=0,j=lo;
 3: while j ≤ m do
 4:    b[i++]=a[j++];
 5: end while
 6:
 7: i=0; k=lo;
 8: while k < j and j ≤ hi do
 9:    if b[i] ≤ a[j] then
10:       a[k++]=b[i++];
11:    else
12:       a[k++]=a[j++];
13:    end if
14: end while
15:
16: while k < j do
17:    a[k++]=b[j++];
18: end while
```

3.2.3 Ordinamento interno: Heapsort

L'heapsort introduce un'altra tecnica di progetto degli algoritmi: l'uso di una struttura (virtuale, nel senso che esiste solamente a livello concettuale e pertanto non occupa risorse di sistema) di dati, denominata *heap*, per gestire le informazioni durante l'esecuzione dell'algoritmo; utile non solo per l'heapsort, ma anche per realizzare efficientemente (con complessità asintotica $O(log(n))$ code a priorità (è quanto si è fatto per realizzare il multiway-merge).

La struttura di dati *heap binario* è un array che può essere visto come un albero binario quasi completo: ogni nodo dell'albero corrisponde a un elemento dell'array che contiene il valore del nodo (vedi figura 3.3 (a)). Nell'heapsort il valore chiave di ogni nodo è maggiore o uguale



(a) (b)

Figura 3.3: (a) visualizzazione di un array come un albero binario (heap) (b) sono valide: (a(i).chiave ≥ a(2i).chiave) e (a(i).chiave ≥ a(2i+1).chiave)

nome algoritmo	complessità caso peggiore	complessità caso medio	complessità caso migliore	quantità memoria ausiliaria
quicksort	$O(n^2)$	$O(nlog(n))$	$O(nlog(n))$	0
mergesort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	O(n/2)
heapsort	$O(nlog(n))$	$O(nlog(n))$ più lento di quicksort	$O(nlog(n))$	0

Tabella 3.1: Tabella comparativa degli algoritmi utilizzati

a quello dei suoi figli (nella versione per code a priorità è il contrario: il valore della chiave di ogni nodo è sempre minore o uguale a quella dei figli, pertanto nella root dello heap cioè nel primo elemento dell'array troviamo l'elemento più piccolo che corrisponde a quello con priorità più eleveta).

Ulteriori caratteristiche, elencate anche nella tabella riepilogativa 3.1, sono le seguenti:

- non necessita di memoria ausiliaria
- ordinamento non stabile
- complessità caso peggiore = caso migliore: $O(nlog(n))$
- complessità caso medio è più lento del quicksort: $O(nlog(n))$

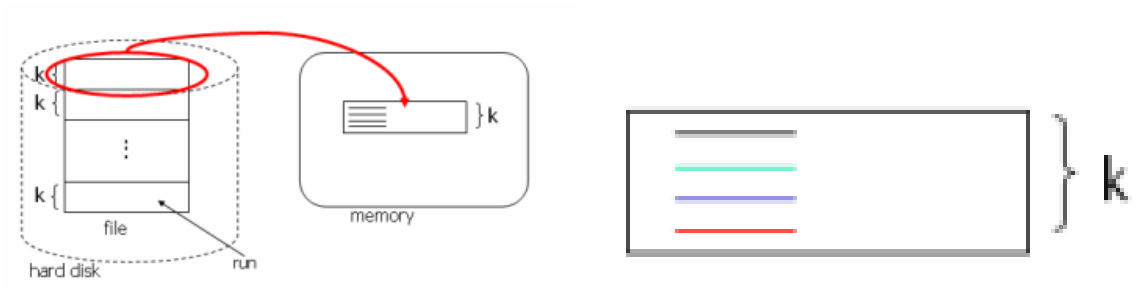
3.2.4 Ordinamento esterno

Solitamente nell'effettuare l'ordinamento dei record di un file, capita che risultino essere in un numero troppo grande per poter essere completamente contenuti nella memoria (a meno di un uso massiccio dello swapping). A casa di ciò, le performance degli algoritmi visti fin'ora (ordinamento interno) sono inacettabili; la soluzione a tale tipo di problemi è rappresentata dagli algoritmi di ordinamento esterno: *sort based*. Il criterio per la valutazione delle performance è il numero di letture e scritture di record, ovvero vengono conteggiate le operazioni di I/O su disco.

3.2.5 Ordinamento esterno: sort based inversion

Il sort-based inversion è composto da due fasi da eseguirsi in cascata. La prima consiste nella *costruzione delle sorted runs*. Per questo progetto si assume che il numero di coppie che possono essere tenute in memoria sia k (ovvero la dimensione di un run). Nel progetto tale valore è rappresentato dalla costante simbolica RUN_DIM_BUFFER, che di default vale 5. I passi da compiere sono:

- lettura di un run (k record) dal file, e scrittura in locazioni contigue di memoria (vedi figura 3.4 (a));
- ordinamento non decrescente dei record in memoria mediante quicksort (vedi figura 3.4);
- riscrittura dei record nel file (vedi figura 3.5 (b));
- ritorno al primo punto finchè ci sono runs nel file;



(a) (b)

Figura 3.4: (a) Lettura di un run (k record) dal file e scrittura in locazioni contigue di memoria. (b) Ordinamento dei record in memoria mediante quicksort: ordinamento non decrescente sui termId ed a parità, ordinamento non decrescente sui docId

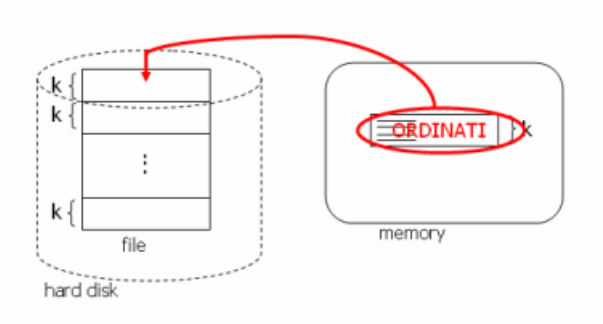


Figura 3.5: Riscrittura dei record ordinati sul file: riscrittura dei vecchi k records

La seconda fase consiste nel fare la fusione e l'ordinamento di coppie di runs. Alla fine, ci troveremo con un file contentente tutte le coppie ordinate (vedi figura 3.6).

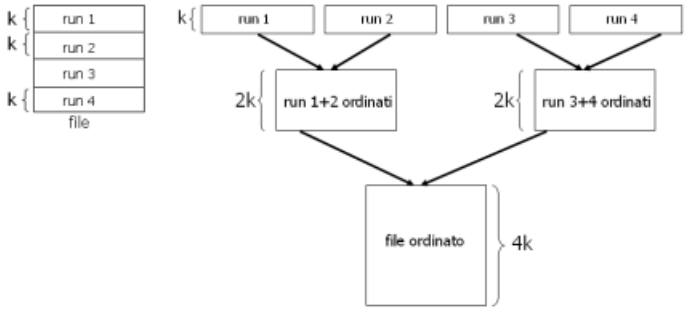


Figura 3.6: Esempio di fusione a coppie di run

Visualizzazione della fusione tra due runs (vedi figura 3.7):

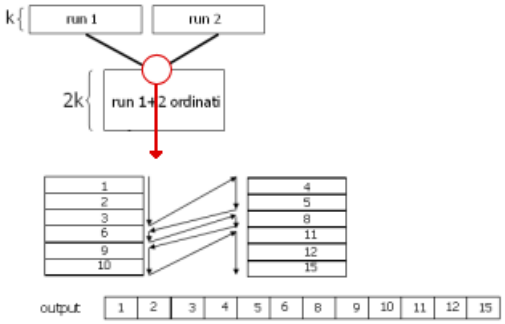


Figura 3.7: Esempio di fusione con ordinamento tra due runs

Il merging può essere implementato in due modi aventi differenti performance. La prima (che d'ora in avanti denoteremo con versione 1.0) consiste nel prendere coppie di runs, trasferire il loro contenuto in due file temporanei differenti e, a partire da questi, effettuare la fusione con l'ordinamento. Il risultato dell'operazione deve essere riscritto nel file originario (vedi figura 3.8). Per quanto riguarda le performace di questo algoritmo, possiamo stimare in $4k$ operazioni di lettura/scrittura, più altrettante operazioni per il merging. In totale il costo sarà dell'ordine di $8k$ operazioni di I/O.

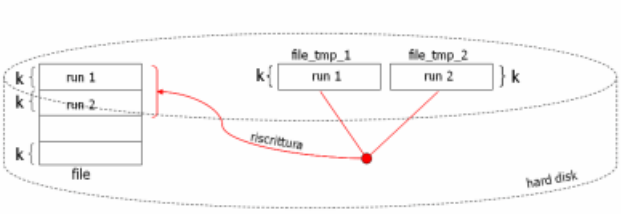


Figura 3.8: Esempio di merging, versione 1.0

La seconda versione (che per comodità la denoteremo con versione 2.0) si differenzia dalla precedente perché i dati non vengono riscritti nel file originario, bensì in un nuovo file. Come è facile immaginare, alla fine dello step i, si può cancellare il file creato allo step $i-1$. Le performance in questo caso sono migliori rispetto alla versione 1.0 (ed è per questo che abbiamo deciso di utilizzarlo nel progetto). Si può stimare il costo di I/O come dell'ordine di $4k$ (rispettivamente $2k$ per la lettura dei dati, $2k$ per la fusione).

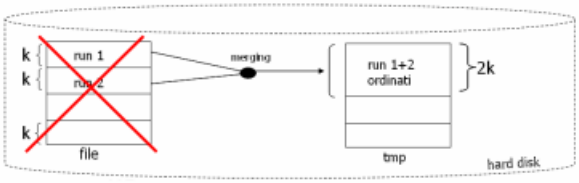


Figura 3.9: Esempio di merging, versione 2.0

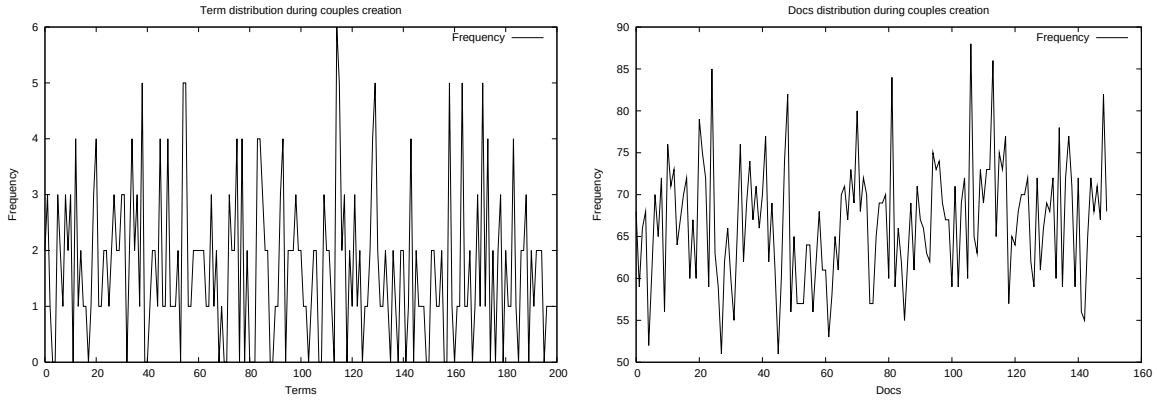
Sort-based inversion simulato. Con scopo puramente didattico, nel progetto è stato riprodotto l'algoritmo esterno *sort-based* attraverso un'emulazione in memoria. L'intento è stato quello di enfatizzare il diverso utilizzo della memoria - disco.

Capitolo 4

L'architettura

4.1 Creazione delle coppie

Questa sezione descrive più in dettaglio come vengono generate le coppie in ogni nodo (vedi anche sezione 2.1). L'algoritmo che è stato utilizzato, utilizza la funzione standard `rand` della libreria `stdlib.h`, grazie alla quale si possono generare numeri casuali all'interno dell'intervallo $[0, RAND\_MAX]$. Dato che le esigenze del progetto richiedono un intervallo pari a $[0, MAX\_DOC]$ che può essere diverso da quello di default, è stato scelto di normalizzare i numeri casuali creati e rappresentarli all'interno di $[0, 1)$. Il risultato ottenuto lo si moltiplicherà per MAX_DOC. Per avere un'idea della forma del grafico delle frequenze che può assumere T_i e D_j, si veda la figura 4.1. È stato rilevato che dopo aver generato circa 2/3 dei $RAND\_MAX$ numeri, la sequenza dei numeri che verranno creati perderà la proprietà di casualità.



(a) (b)

Figura 4.1: Grafico della frequenza dei T_i (a) e D_j (b) per un intervallo di circa 200 e 150 elementi

4.2 Globalizzazione dell'identificatore dei documenti

Considerando la versione parallela della nostra applicazione, abbiamo che ogni nodo ha i suoi documenti ognuno dei quali ha un identificatore id che è unico all'interno del nodo stesso,

ma non nell'insieme di tutta l'applicazione, pertanto è necessario che tali id diventino globali. Vedremo due metodi, il primo è quello che ci verrebbe più spontaneo adottare, il secondo utilizza l'operazione collettiva Prefix Sums. Confronteremo le performance di tali due metodi nelle tre tipiche topologie della rete: array lineare (ring), two dimensional mesh e ipercubo.

4.2.1 Algoritmo versione 1

Prima di passare alla presentazione dell'algoritmo, è necessario aggiungere rispetto alla notazione precedentemente presentata, la definizione di $numberOfDocId[P_m] = |\mathcal{P}_m|$ come il numero di documenti distinti nel generico nodo P_m. Ora, la soluzione al problema più naturale che viene in mente è quella in cui ogni nodo P_m comunica al nodo con rank P_{m+1}, il proprio $numberOfDocId[P_m]$. I passi da effettuare sono i seguenti:

- si ordina in modo crescente i processi in base al loro rank
- la comunicazione avviene solo e soltanto tra due processi consecutivi
- il processo 0 (in short P_0) manda il suo numero di distinti docId ($numberOfDocID$) al processo successivo ovvero P_1
- P_1 rienumera i propri local docId a partire dal valore di $numberOfDocID[P_0] + 1$. Manda a P_2 il massimo docId ottenuto dalla rienumerazione cioè $numberOfDocID[P_0]+$ $numberOfDocID[P_1]$
- ...e così via ...

In generale, il processo P_i:

- riceve $numberOfDocID[P_{i-1}]$
- rienumera i propri documenti a partire da $numberOfDocID[P_{i-1}] + 1$
- spedisce a P_{i+1} il massimo docID ottenuto dal precedente punto (in maniera equivalente: $numberOfDocID[P_{i-1}] + numberOfDocID[P_i]$)

Esprimiamo il tutto con un esempio. Supponiamo di avere un'architettura DM-MIMD, quattro loosely coupled machines collegate da una rete avente topologia d'array lineare e per ogni processore i seguenti numeri di distinti docId: $numberOfDocID[P_0] = 5$, $numberOfDocID[P_1] = 8$, $numberOfDocID[P_2] = 3$, $numberOfDocID[P_3] = 6$. Il pattern di comunicazione è riassunto nella figura 4.2.

Analizzando le performance, avendo 4 processi il numero totale di step necessari alla globalizzazione dei docId è 4. In generale, dato p il numero di processi, il numero totale di step è $p - 1$.

Tale pattern di comunicazione può essere realizzato in maniera più efficiente, mediante **prefix sums**.

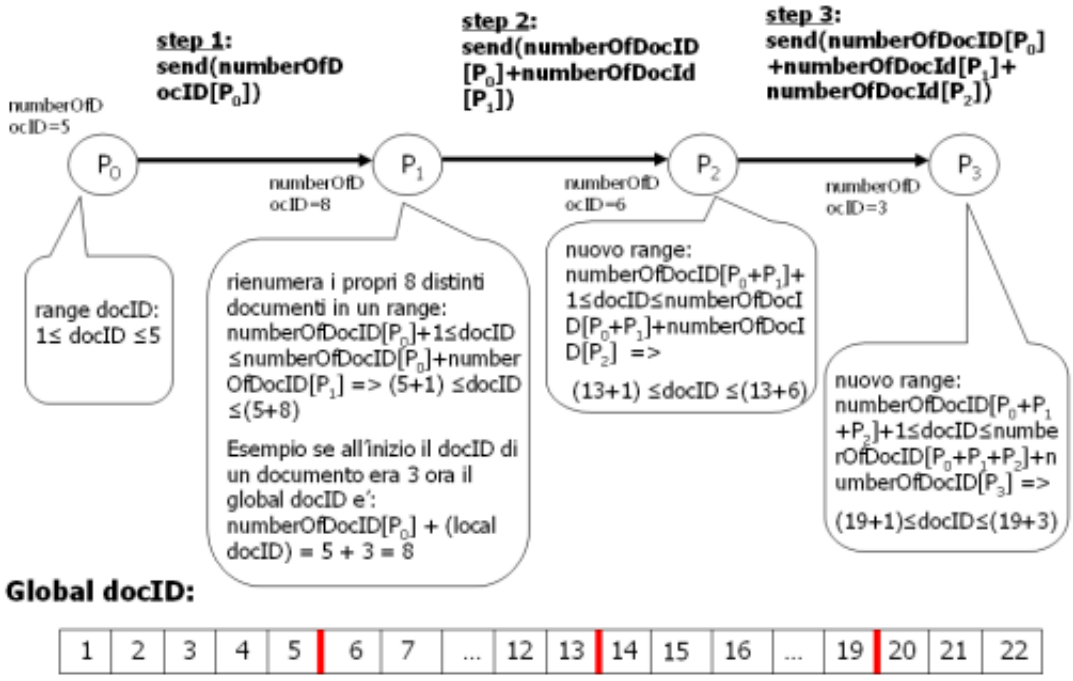


Figura 4.2: Pattern di comunicazione in un array lineare tra 4 processi per la globalizzazione dei docID tramite algoritmo versione 1.0

4.2.2 Prefix sums

Il Prefix Sums, implementato in MPI tramite l'operazione collettiva `MPI_Scan`, è il seguente: dato p numeri $n_0, n_1, n_2, \ldots, n_{p-2}, n_{p-1}$ (uno per ogni processo), per ogni nodo k con $k < p$ viene calcolato:

$$S_k = \sum_{i=0}^{k} n_i$$

Traducendo il tutto nel nostro esempio: n_0 è $numberOfDocID[P_0]$, n_1 è $numberOfDocID[P_1]$, n_2 è $numberOfDocID[P_2]$ e n_3 è $numberOfDocID[P_3]$; per ogni processo viene calcolato:

$$S_k = \sum_{i=0}^{k} numberOfDocId[P_i]$$

Al nodo i-esimo viene calcolata la somma dei $numberOfDocID$ dei precedenti $i-1$ processi più il proprio $numberOfDocID[i]$. Al fine di conoscere il docId di partenza per l'enumerazione, basta sottrarre da S_k il local $numberOfDocID$ e aggiungere 1.

Il pattern di comunicazione del Prefix Sums è molto simile al all-to-all broadcast, infatti lo si può vedere nella figura 4.3.

Anche con l'utilizzo del Prefix Sums, applicato ad un array lineare, necessitano ancora di $p-1$ step (dove p è il numero di processi), tuttavia si ottengono ulteriori benefici: l'utilizzo di una sola e identica funzione per tutti i processi (operazione collettiva), metodo già implementato in MPI e pertanto meno lavoro per il programmatore, garanzie di performance indipendentemente dalla topologia fisica della rete (process-processor mapping) e infine il programmatore

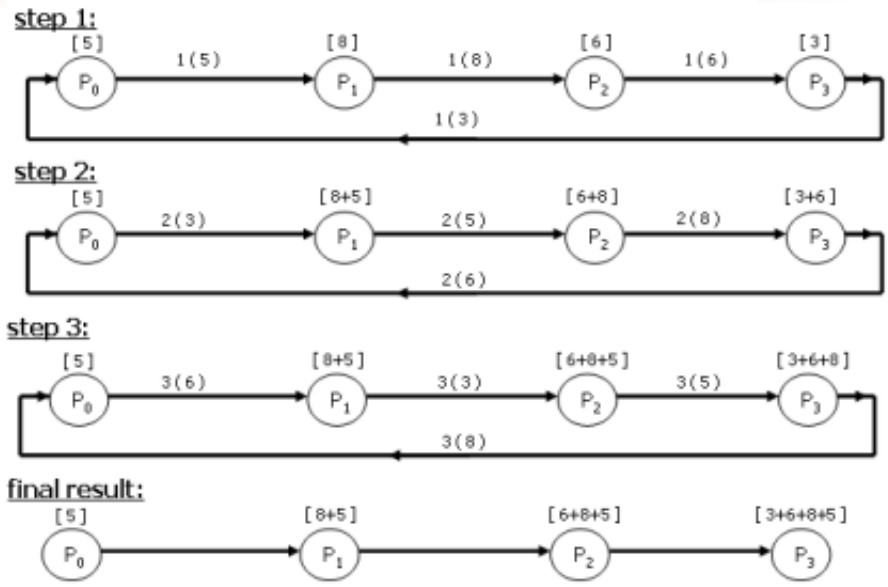


Figura 4.3: Pattern di comunicazione in un array lineare tra 4 processi per la globalizzazione dei docID tramite prefix sum

non deve rimettere mani al proprio codice quando cambia la topologia della rete (portabilità dell'applicazione).

4.2.3 Algoritmo versione 1.0 Vs Prefix Sums in un 2D mesh

Confrontiamo le performance dell'algoritmo v1.0 con il Prefix Sums cosiderando un'altra topologia dell'interconnection network: two dimensional mesh. Supponiamo di avere $p = 9$ processi organizzati in un two dimensional mesh detto anche *3-ary 2 cube*.

Analisi performance in un 2D mesh:

L'algoritmo v1.0 effettua la globalizzazione dei docId in un numero totale di passi pari a 8 cioè: dato p processi il numero di step da compiere è $p - 1$ (vedi figura 4.4(a) e (b)). Il Prefix Sums esegue il medesimo compito in 4 passi, cioè:

- prima fase: in parallelo $\sqrt{p} - 1$ passi (vedi figura 4.4 (c)).
- seconda fase: in parallelo $\sqrt{p} - 1$ passi (vedi figura 4.4 (d)).
- totale: $2 * (\sqrt{p} - 1)$ passi.

L'algoritmo migliore anche nel two dimensional mesh è senza ombra di dubbio il Prefix Sums. Entrambi i pattern di comunicazione sono riassunti in figura 4.4.

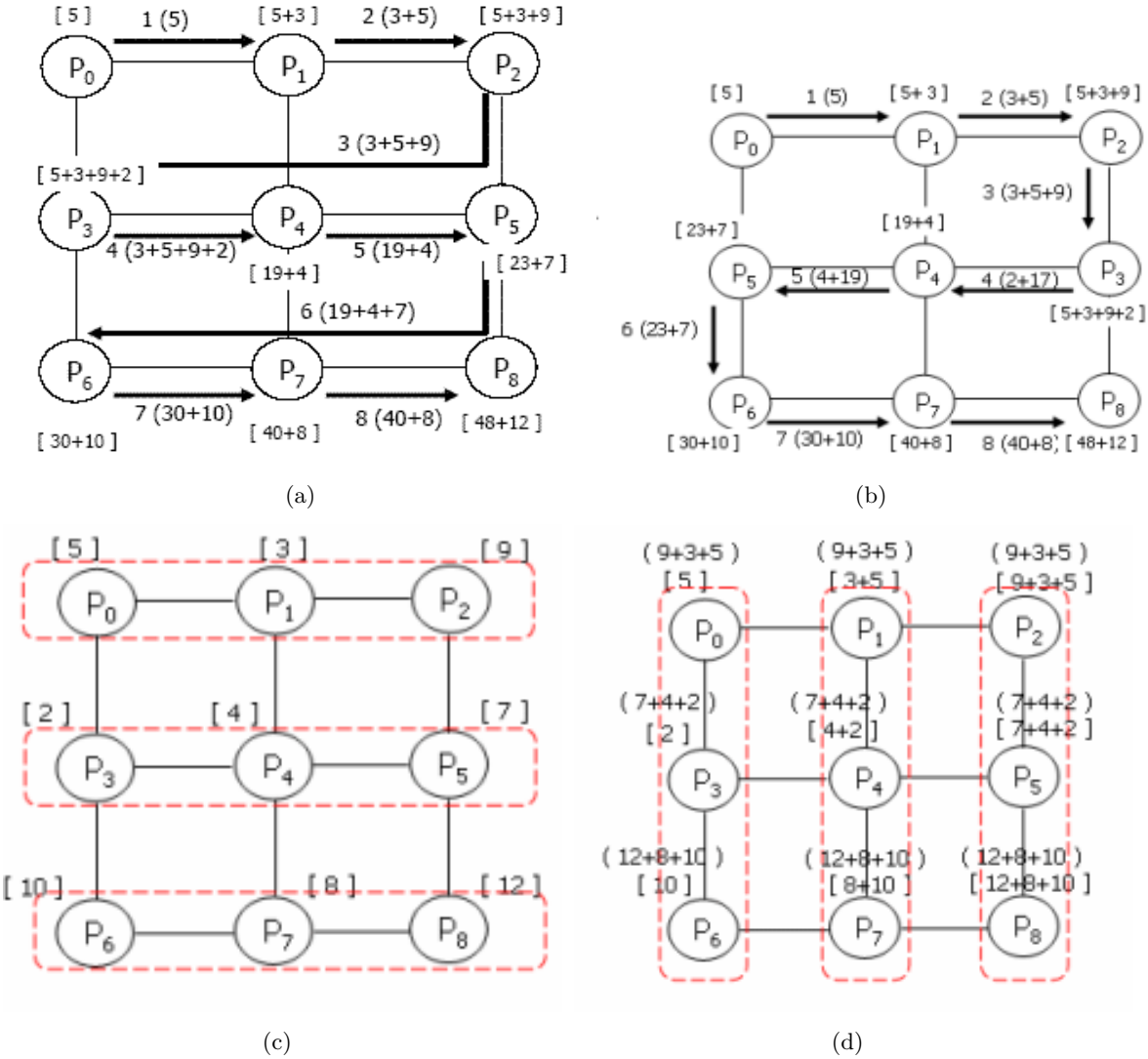
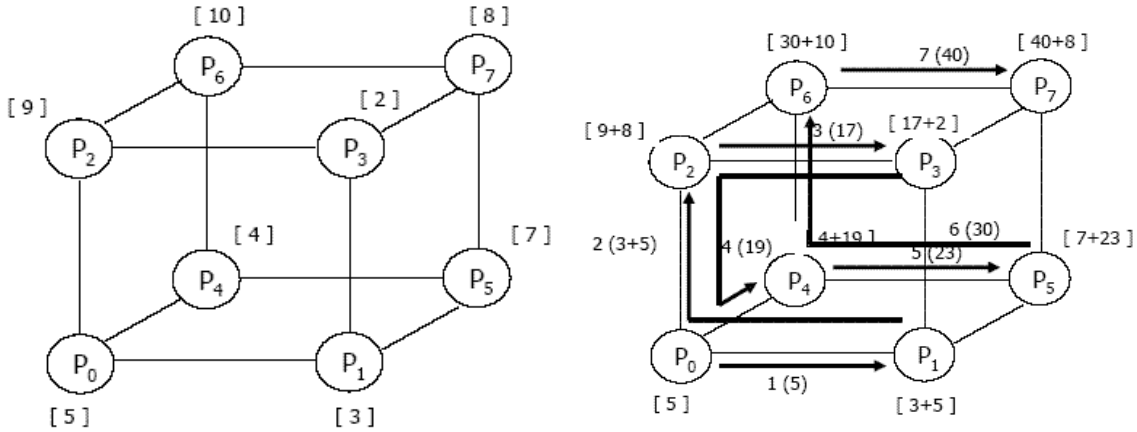


Figura 4.4: (a) Pattern di comunicazione in un 2D mesh utilizzando l'algoritmo versione 1.0. (b) Pattern di comunicazione ottenuto mediante un migliore process-processor mapping e utilizzando algoritmo versione 1.0. (c) Prima fase del pattern di comunicazione del prefix sum. (d) Seconda fase del pattern di comunicazione del prefix sum.



(a) (b)

Figura 4.5: (a) Ipercubo a 3 dimensioni. (b) Pattern di comunicazione in un ipercubo a 3 dimensioni utilizzando algoritmo versione 1.0.

4.2.4 Algoritmo versione 1.0 Vs Prefix Sums in un ipercubo a 3 dimensioni

Raffrontiamo le performance dell'algoritmo v1.0 e il Prefix Sums utilizzando come interconnection network l'ipercubo a 3 dimensioni (*2-ary 3-cube*): assumiamo di avere $p = 8$ processi (infatti $N = 2^d = 8$) .

Analisi performance in un ipercubo a 3 dimensioni:

L'algoritmo v1.0 effettua la globalizzazione dei docId in 7 step cioè: dato p processi il numero di passi da eseguire è pari a $p - 1$ (vedi figura 4.5).

Il Prefix Sums esegue il medesimo compito in 3 passi: in generale, dato 2^d processi, dove d è la dimensione dell'ipercubo, il numero totale di step è $log_2 2^d = d$ step (vedi figura 4.6).

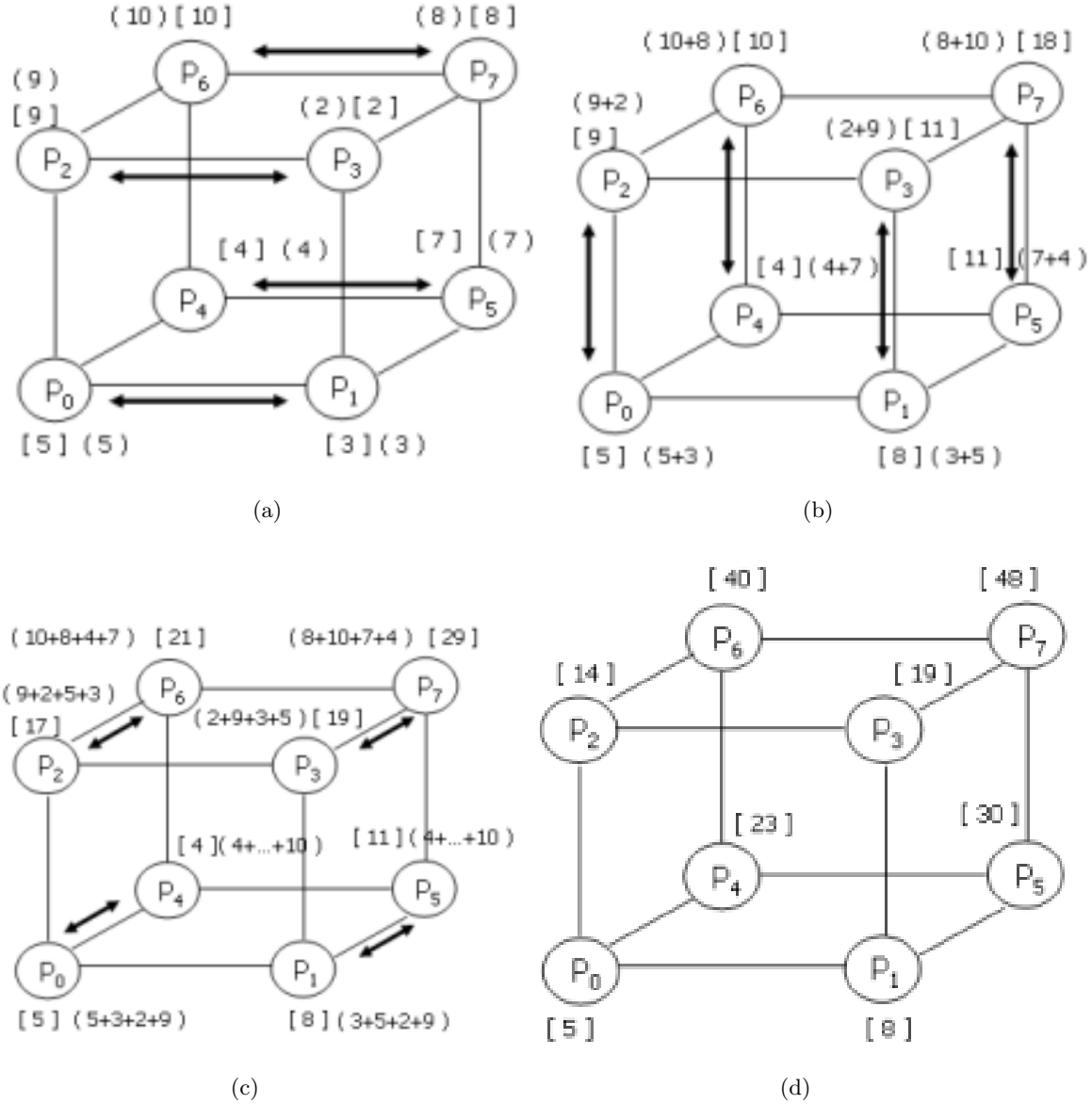
Confronto: $(p - 1)$ Vs $log_2 p$ (dove $p = 2^d$) Netta supremazia nelle performance da parte del Prefix Sums.

4.3 Versione sequenziale

In questa versione del progetto, il nodo root (identificato dal rank 0) svolge tutto il lavoro (generazione coppie, ordinamento e creazione posting lists) e successivamente distribuisce partizioni delle posting lists agli altri nodi facenti parte dell'applicazione. Pertanto, ciò che otteniamo è il medesimo risultato della versione parallela del progetto, con la differenza che qui un solo nodo effettua la computazione, mentre gli altri attendono il risultato: ecco il motivo della denominazione sequenziale. Tale versione del progetto è utile per calcolare lo Speedup.

Il noto root (rank 0) esegue in modo sequenziale i seguenti passi.

Lettura dei parametri dal file identificato dalla costante simbolica FILE_PARAMETERS. Tale file contiene le informazioni necessarie alla creazione delle coppie, come il numero di termini distinti (MAX_TERM), il numero di documenti distinti (MAX_DOC), il numero di coppie da generare (MAX_CP), il termId di partenza ed il termId di fine (*stopTerm*) [rappresentano il range del termID: $startTerm = termID < stopTerm$]



(a) (b)

(c) (d)

Figura 4.6: (a) prima dello step 1. (b) prima dello step 2. (c) prima dello step 3. (d) risultato finale

Generazione coppie: creazione delle coppie rispettando quanto inserito nel file dei parametri (vedi sezione 4.1).

Ordinamento coppie: sorting delle coppie utilizzando uno dei seguenti algoritmi: ordinamento esterno (sort-based), ordinamento interno (quicksort, mergesort, heapsort e sort-based simulato). La scelta dell'algoritmo di ordinamento avviene a run time da parte dell'utente il quale riceve suggerimento dall'applicazione: se la dimensione del file delle coppie è inferiore o uguale al limite espresso (in MB) dalla costante simbolica DIM_LIMIT_IN_MB, allora l'applicazione consiglia l'utilizzo di un algoritmo di ordinamento interno altrimenti utilizza il sort based. **Creazione posting lists**: generazione delle posting lists relative a tutti i termId. **Partizionamento posting lists**: equipartizione delle posting lists in base al numero di nodi dell'applicazione (incluso il nodo root). Esempio: dato 10 distinti termId, e 4 nodi, il nodo 0 (root) tratterrà 3 posting lists, il nodo 1 ne riceverà 3, il nodo 2 ne riceverà 2 e il nodo 3 ne riceverà 2. **Distribuzione delle posting lists** ai rispettivi nodi: ad ogni nodo non verrà mandato un singolo messaggio pari alla dimensione delle posting lists da spedirgli, ma tanti frames la cui parte dati è della dimensione specificata dalla costante simbolica NUMBER_OF_CHAR TO_READ, ovvero 200 bytes (o esprimibile come 200 caratteri visto che MPI_CHAR occupa 1 byte).

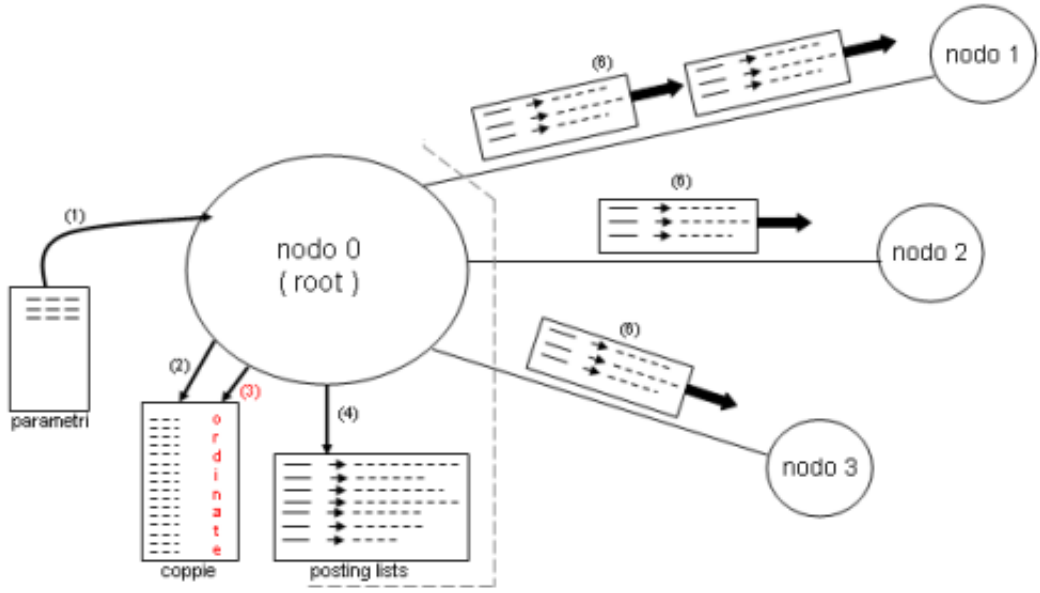


Figura 4.7: (1) caricamento del file dei parametri; (2) generazione delle coppie; (3) ordinamento delle coppie; (4) creazione delle posting lists; (5-6) equipartizione e distribuzione delle posting list;

Quali criteri guida si devono seguire per modificare la lunghezza del messaggio intesa come quantità di dati da spedire al destinatario? Abbiamo considerato come interconnection network Ethernet (802.3)[14], la quale è presente nel laboratorio 14 del dipartimento di informatica dell'università Ca' Foscari di Venezia, nel quale sono stati effettuati i test. Lo standard 802.3 fissa come limite minimo, alla grandezza del dato in un frame, di almeno 46 bytes; qualora il dato risultasse minore, viene aggiunto del *padding* per raggiungere la dimensione minima del frame. Attenendoci alle specifiche ANSI C in particolar modo all'header `LIMITS.H`, il numero di bits

in un char è di 8, cioè 1 bytes (in ugual modo MPI_CHAR), pertanto per evitare l'utilizzo del *padding* si dovrà spedire almento 46 caratteri. Al fine di mascherare il degrado delle performance dell'applicazione dovuto alla comunicazione, è necessario che la quantità di characters da mandare sia superiore. Lo standard 802.3 dice in più che il limite massimo di dati che possono essere contenuti in un frame è 1500 bytes.

La figura 4.7 rappresenta i passi che vengono compiuti dal nodo 0.

4.4 Ring

Un'architettura per il trasferimento di dati è necessaria dal momento che, come riferito nella sezione 2, ogni nodo deve cedere la propria parte di dati che non gestisce direttamente al nodo di competenza. La prima architettura che è stata testata in questo progetto è quella a ring. Questa architettura necessita di $p-1$ step, se p è il numero di nodi a disposizione. Ad ogni step, il nodo i-esimo trasferisce dati al nodo $i+1$-esimo. La quantità di dati non è sempre uguale. Infatti, durante il primo step, ogni nodo k trasferisce al successivo tutte quelle coppie (T_i, D_j) in cui $T_i \notin \mathcal{T}^k$ per ogni i, j. Più precisamente, tutte quelle coppie (T_i, D_j) con $T_i \in (\mathcal{G}^k - \mathcal{T}^k)$.

Dal secondo step al $(p-1)$-esimo, ogni nodo dovrà spedire al nodo successivo i dati ricevuti dal predecessore, togliendo da questi le coppie che devono essere gestite dal nodo corrente. Questo algoritmo non è altro che quello che in [11] viene definito come *all-to-all personalized communication*.

4.4.1 Threaded version

Passando al lato implementativo, il modulo adibito alla gestione del ring, crea in ogni nodo tre thread. Uno addetto alla preparazione dei dati che devono essere spediti (*spedizione*) al nodo successore, l'altro addetto alla preparazione dei dati che devono essere ricevuti (*ricezione*) dal nodo predecessore e l'ultimo che gestisce tutte le chiamate a funzioni MPI (vedi figura 4.8). Nonostante fosse stato più naturale aver utilizzato due thread che concorrentemente richiamavano funzioni MPI, ci siamo dovuti adeguare alla versione di `mpich` (1.2.5 e 1.2.7) presente nei laboratori, la quale non è thread safe. Questo implica che, se thread differenti, effettuano chiamate MPI, il programma potrebbe non terminare oppure terminare inaspettatamente, come riferito in [8].

Ci sono diversi modelli di programmazione a thread in letteratura. Nel nostro caso, si è adottato un modello ibrido, nel senso che nella prima fase il modello è **peer**, ogni thread prosegue la propria esecuzione indipendentemente; nella seconda parte, invece, è del tipo **produttore-consumatore**. Questo è una diretta conseguenza di come sono stati implementati i thread e che vedremo successivamente.

Prima della creazione e dell'esecuzione in parallelo dei thread, ogni nodo si fa carico di accodare al proprio file delle coppie (che contiene i dati da spedire) una coppia fittizia che fungerà da terminatore. Questa scelta è stata fatta perché così non è necessario spedire alcun messaggio MPI che indica la fine dei dati da spedire. Ogni volta che uno dei due thread riceve la coppia fittizia, posso incrementare il numero dello step corrente. Da ciò ne deriva che esistono due contatori differenti degli step, rispettivamente per il primo e per il secondo thread e che comunque in tutti e due i casi, possono raggiungere un valore al massimo pari a $p-1$ (dato che nodo deve ricevere dati da tutti gli altri).

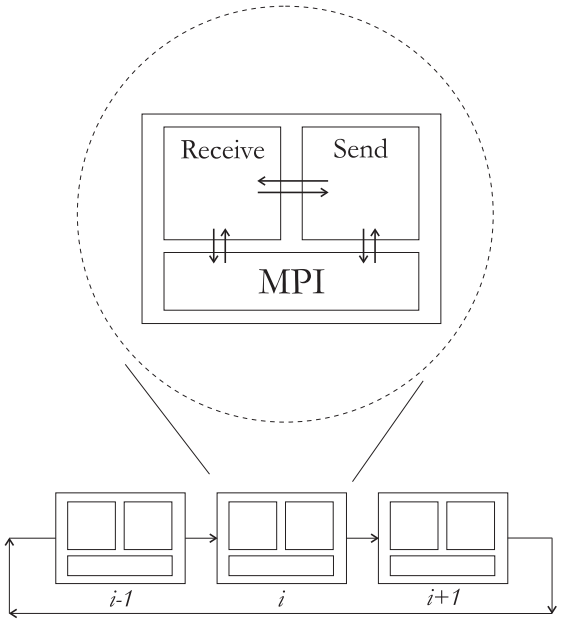


Figura 4.8: Architettura per il trasferimento dei dati dell'i-esimo nodo

I peer thread che contengono la logica di spedizione e ricezione dei dati, utilizzano rispettivamente un buffer per la ricezione ed uno per la spedizione dei dati, rispettivamente *buffSend*, *buffRecv*. L'applicazione accetterà un parametro da riga di comando grazie al quale è possibile variare la dimensione di questi buffer in modo tale da osservare come la modifica di questa variabile influisce sulla velocità di esecuzione del progetto (vedi Risultati, cap. 5). Questa è l'unica zona di memoria primaria utilizzata, che si può quantificare in $|buffSend| + |buffRecv|$. In tutti gli altri casi si utilizza memoria secondaria.

Thread di spedizione. Quando il thread parte, la prima cosa che viene fatta prima di iniziare la spedizione dei dati, è di aprire il file delle coppie generate localmente (e ordinate) e fare un filtro. Infatti, questo file può contenere delle coppie (T_i, D_j) tali che $T_i \in \mathcal{T}^k$ (assumento di essere nel nodo k) e per cui escluse dalla spedizione (vedi figura 4.10 (d)). Queste coppie saranno scritte nel file $runheapway_p$. Il primo step finisce nel momento in cui viene letta la coppia fittizia che termina la lista. Da questo momento in poi, e per tutti gli altri step, il file da cui leggere le coppie sarà creato dal thread che riceve i dati (in seguito lo chiameremo $forwardfile_b$ con $b \in [1, p-1]$). Se il thread raggiunge la fine di questo file e non ha ancora letto la coppia fittizia, allora andrà in wait (`pthread_cond_wait`) aspettando che il thread di ricezione lo risvegli non appena ha ricevuto (e quindi accodato) dati al file. Questo è momento di switch tra i due modelli di programmazione a thread: si passa da peer a produttore-consumatore.

Thread di ricezione. A differenza del thread precedentemente descritto, il thread di ricezione deve essere eseguito dopo aver ricevuto i dati nel buffRecv. Se così non fosse, il thread va in wait e viene risvegliato non appena il buffer contiene dati da elaborare. Dopo aver verificato questa condizione, e per ogni step in cui il thread si trova, bisogna effettuare il filtro dei dati. Infatti, come per lo step 1 del thread di spezione, possiamo aver ricevuto coppie gestite direttamente dal nodo oppure no. Nel primo caso si spostano le coppie dal buffer ad un file (uno per ogni step) denominato $runheapway_s$ con $s \in [1, p-1]$ e nel secondo caso nel file

$forwardfile_b$ con $b \in [1, p-1]$.

Per una rappresentazione grafica di come funziona questa architettura, si veda la figura 4.10 con la relativa spiegazione.

Thread che gestisce le chiamate a funzioni MPI. Questo thread è stato creato in modo tale per cui quando viene creato, va subito in wait aspettando che il primo thread che ha bisogno di ricevere o spedire dati, lo risvegli. Dopo un segnale di risveglio, viene lanciata la funzione MPI richiesta. Non appena il lavoro è terminato, il thread ritorna in wait. In figura 4.9 si è cercato di rappresentare il grafo degli stati assunto dal thread che gestisce le chiamate a funzioni MPI.

Il trasferimento dei dati avviene attraverso chiamate a funzioni MPI *non bloccanti* con l'aggiunta di cicli while per il testing di riempimento del buffer utente (sia per le send che per le receive). La scelta di funzioni non bloccanti non preclude la normale esecuzione del thread [13]. Viceversa, l'uso di send o receive bloccanti implica il blocco del thread che ha invocato la chiamata e questo può portare, per come è strutturata questa architettura, a deadlock o comunque a terminazioni inaspettate del sistema.

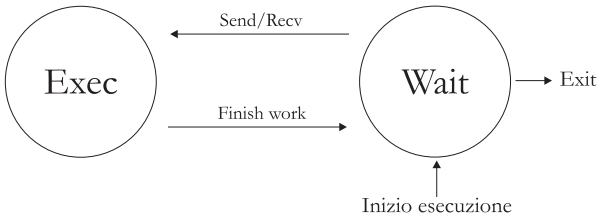


Figura 4.9: **Grafo degli stati** del thread MPI. Quando il thread viene creato ed eseguito va subito in wait aspettando che uno degli altri due thread lo risvegli con un segnale. Alla fine del lavoro, il thread ritorna in wait. La terminazione, e la conseguente distruzione del thread, sarà effettuata a terminazione dei thread di ricezione e spedizione avvenuta.

Per spedire le coppie, inizialmente si spedivano 2 array di tipo `MPI_INT` (uno per l'identificatore dei termini e l'altro per l'identificatore dei documenti). In un secondo momento, alcune prove sono state fatte utilizzando una struttura dati MPI del tipo:

```
MPI_Datatype Couple;
MPI_Datatype type[2]={MPI_INT, MPI_INT};
int blocklen[2]={1,1};
MPI_Aint displacement[2]={0,sizeof(int)};
MPI_Type_struct(2,blocklen,displacement,type,&Couple);
MPI_Type_commit(&Couple);
```

4.4.2 Not-Threaded version

La sezione precedente analizza in dettaglio l'implementazione dell'architettura a ring nel caso di programmazione a thread. Per avere una visione completa delle possibili soluzioni implementative del problema, e per vedere quale fosse la versione migliore in funzione dei parametri di esecuzione, è stato scelto di realizzare una versione not-threaded. Al fine di riutilizzare il maggior codice possibile, è stata ripresa la struttura base della versione a thread. Infatti, le

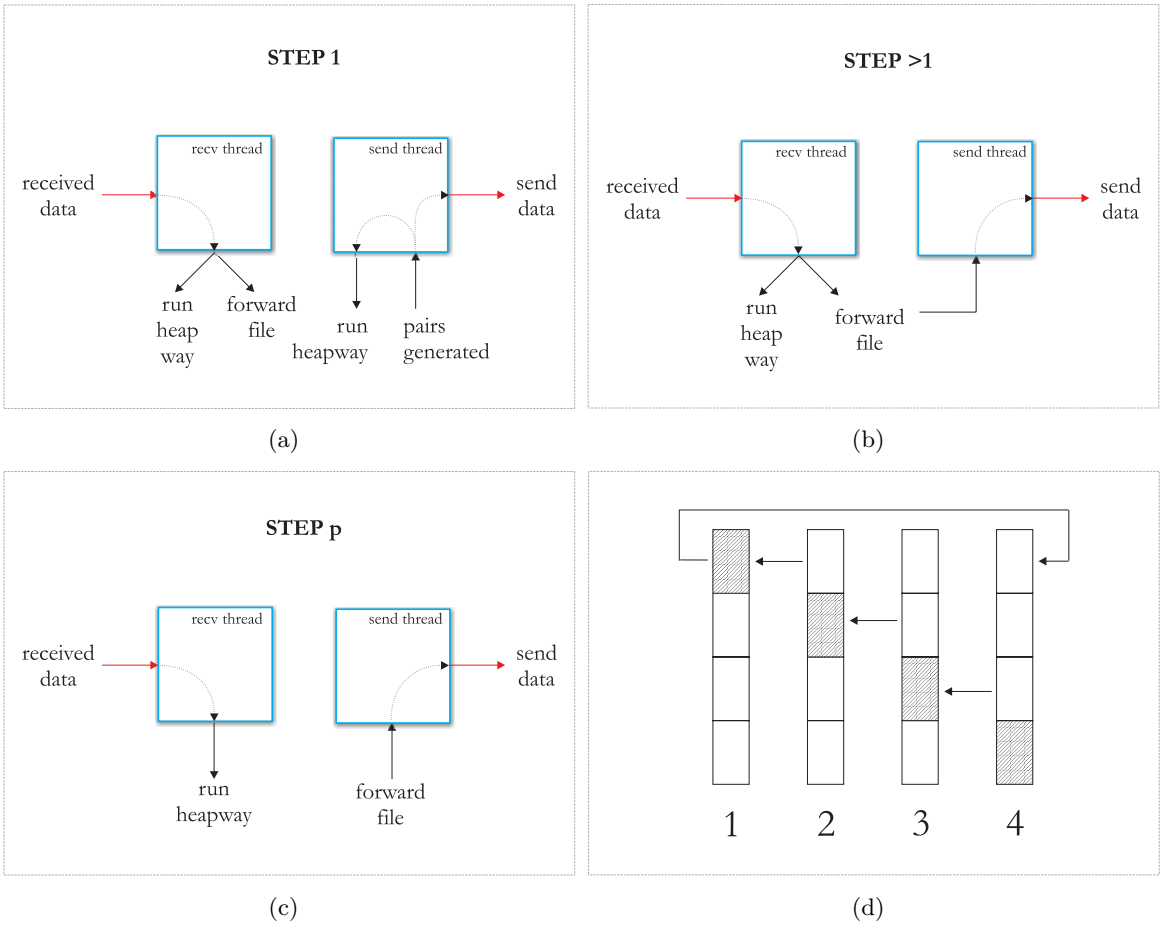


Figura 4.10: Configurazioni che si presentano durante una normale esecuzione del programma con architettura a thread. (a) rappresenta il primo step: solitamente, se il thread send viene eseguito per primo, spedisce le coppie che non vengono gestite dal nodo e crea un file che sarà usato durante la fase di merging dei run ricevuti. (b) Durante gli altri step, ultimo escluso, i due thread lavorano in parallelo e accedono in maniera concorrente al forward file in cui il thread che riceve scrive e quello che spedisce, legge. (c) L'ultimo step differisce dai precedenti perché i dati che il thread riceve, vengono immediatamente scritti nei file delle corse. Come si può notare, anche osservando (d), l'ultimo step è come se corrispondesse alla ricezione, in direzione contraria, dal nodo successore, di termini che ogni nodo gestisce direttamente. I numeri rappresentano i rank dei nodi. Le zone evidenziate corrispondono agli insiemi $\mathcal{T}^k$, $k \in [1, \ldots, 4]$.

funzioni che prima erano associate i thread di spedizione e ricezione avranno inglobate le chiamate MPI. Quello che per la versione a thread era una wait, ora è un sistema che salva lo stato della funzione (cioè il numero di coppie elaborate e quindi inserite nel buffer) in modo tale che al successivo richiamo, si prosegua con il lavoro precedentemente iniziato. Il codice che creava i thread e faceva il join è stato rimpiazzato da un ciclo che alternativamente richiama le due funzioni.

Capitolo 5

Risultati

Sia per la versione sequenziale che per quella parallela, sono state utilizzate le risorse offerte dal dipartimento di informatica dell'Università Ca' Foscari. I computer in ogni laboratorio hanno tutti la medesima configurazione hardware e software. Fujitsu Siemens, Scenic T con processore Genuine Intel 1.80 Ghz, BIOS Phoenix Technologies Ltd 4.06 Rev 1.06.1381, 07/05/02, memoria principale pari a 256 MB, scheda grafica Nvidia GeForce2 MX/MX 400 (PCI) con 64MB di Ram, disco Maxtor 4D040H2 da 40GB di cui 14GB destinati al sistema operativo linux, scheda di rete Intel PRO/100 VE Network Connection, sistema operativo Linux Ubuntu, kernel 2.6.10-5-686 e con versione MPI mpich 1.2.7, con compilatore C: gcc 4.0.2 20050901.

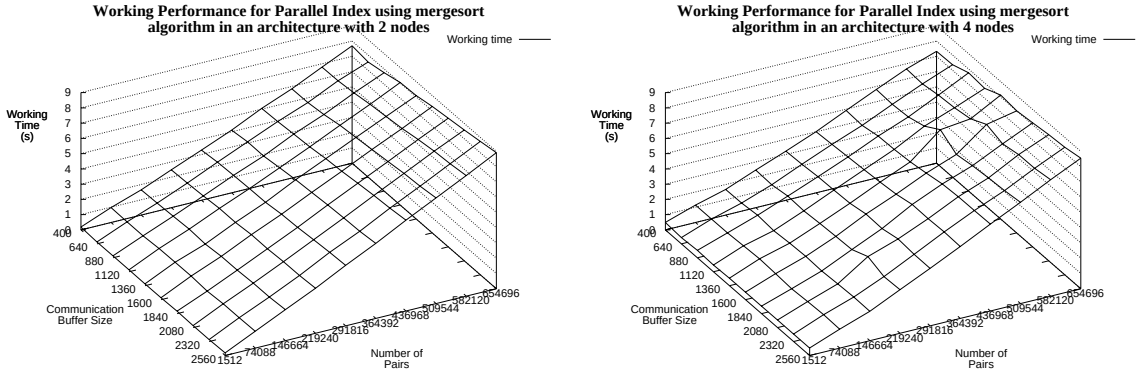
I test sono stati eseguiti in due laboratori con configurazione di rete differente. La maggior parte di questi sono stati fatti utilizzando computer connessi tramite una rete Ethernet a 100Mb/s. Alcuni test sono stati fatti nel laboratorio con una rete di prestazioni inferiori rispetto alla precedente (vedi test versione parallela) e questo ci ha permesso di rilevare un degrado delle prestazioni. In tutti e due i casi, i tempi di esecuzione che sono stati rilevati variano in base alla dimensione del buffer dei runs (Run Buffer Size), alla dimensione del buffer dei messaggi (Communication Buffer Size), al numero delle coppie (Number of Pairs) ed al numero delle macchine utilizzate (in tutti i casi sono state utilizzate 2 e 4 macchine). Tutte le prove che sono state fatte, suppongono di avere $\mathcal{T} = 6000$ e $\mathcal{D} = 1200$ (quest'ultimo valore distribuito equamente nelle varie macchine che formano la processors farm).

Per quanto concerne i possibili valori che le variabili possono assumere, si è definito:

- Run Buffer Size $\in [5, 81851]$ con incrementi di 9094 unità per un totale di 10 step.
- Communication Buffer Size $\in [400, 2560]$ (bytes) con incrementi di 240 bytes rispettivamente per la versione sequenziale, $[100, 640]$ con incrementi di 60 bytes per la versione parallela. La scelta di imporre un range diverso per le due versioni è dettata dal fatto che la dimensione del datatype `MPI_INT` è quattro volte superiore a quella dell'`MPI_CHAR`.
- Il numero di coppie (T_i, D_j) varierà all'interno del range $[1512, 654696]$ con incrementi di 72576 unità.

5.1 Versione Sequenziale

Analisi performance variando gli algoritmi di ordinamento. Tali test sono stati eseguiti tenendo fisso il Run Buffer Size pari a 5 coppie e variando il Communication Buffer Size, Number



(a) (b)

Figura 5.1: L'esecuzione su 2 macchine è leggermente più lenta.

of Pairs, il numero di macchine e l'algoritmo di ordinamento. Per quest'ultimo parametro sono stati utilizzati gli algoritmi di sorting interno quali quicksort, mergesort, heapsort, sort-based simulato e l'algoritmo di ordinamento esterno sort-based inversion (facente uso del quicksort per il sorting all'interno di un run).

Algoritmi di ordinamento interni:

quicksort.Il grafico mostra come il quicksort eseguito su 4 macchine sia circa due volte piu' veloce rispetto a 2 macchine (vedi figura B.7).

mergesort. Il mergesort è meno performante del quicksort. Vedi figura 5.1

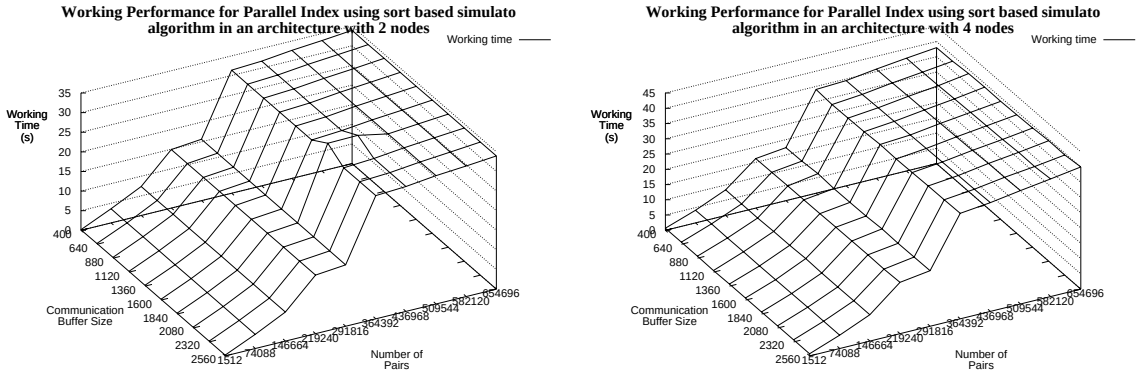
heapsort. Ottenimento del medesimo scenario del mergesort. È possibile affermare che tra i classici algoritmi di ordinamento interno, quello che permette di ottenere le migliori performance è il quicksort (vedi figura B.6).

Da notare nel grafico relativo all'esecuzione su 4 macchine, un aumento di prestazioni a partire dal Communication Buffer Size di 2080 bytes: in quel momento la rete scarica. Ad avvalorare tale nostra ipotesi è l'algoritmo di testing usato dallo script: uso di due cicli annidati, in quello più esterno varia Communication Buffer Size, in quello interno Number of Pairs; ciò conferma la nostra ipotesi.

Da un'analisi complessiva dei precedenti grafici emerge il fatto che utilizzando 4 macchine si ottengono maggiori prestazioni, ma teoricamente è vero l'incontrario; tale incongruenza è scaturita dall'aver dapprima eseguito, nella mattinata (solitamente molti utenti in laboratorio) la batteria di test su 2 macchine per gli algoritmi quicksort, mergesort e heapsort, e successivamente nel pomeriggio (generalmente meno utenti in laboratorio) il testing su 4 macchine. Pertanto tali discrepanze sono imputabili molto probabilmente alla variazione di carico della rete.

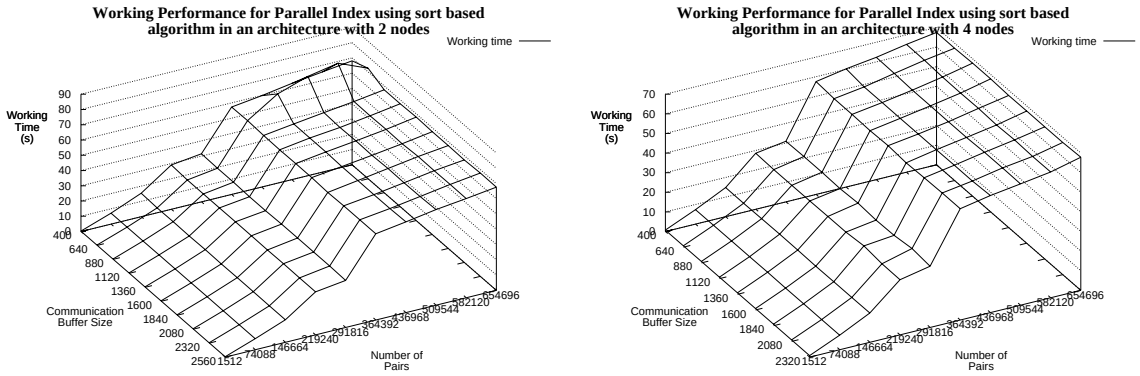
sort based simulato. La maggiore prestazione velocistica è ottenuta dall'esecuzione della versione sequenziale su 2 macchine. In questo caso, il test su 2 pc è stato svolto nella tarda mattinata durante la pausa pranzo pertanto a rete scarica, mentre quello su 4 macchine è stato eseguito nello stesso giorno ma nel pomeriggio (vedi figura 5.2).

sort based. I tempi di performance ottenuti su 2 macchine sono pressochè identici a quelli su 4 nodi (vedi figura 5.3). Sottolineamo come per lo stesso carico di lavoro, il quicksort e in generale gli algoritmi di ordinamento interno, sono in media 30 volte più veloci. Tale divario è giustificabile da due motivi: in primo luogo dalla grande disponibilità di memoria principale



(a) (b)

Figura 5.2: Applicazione dell'algoritmo sort-based simulato rispettivamente per 2 e 4 nodi variando la dimensione del buffer di run ed il numero di coppie.



(a) (b)

Figura 5.3: Applicazione dell'algoritmo sort-based rispettivamente per 2 e 4 nodi variando la dimensione del buffer di comunicazione ed il numero di coppie.

(pari a 256 MB) con conseguente capacità di contenere un alto numero di coppie senza ricorrere al meccanismo dello swapping, in secondo luogo dalle pessime performance dei dischi stessi aventi un regime di rotazione di 5400 RPM. Velocità molto bassa rispetto ai dischi che vengono normalmente venduti da circa 4 anni a 7200RPM; ciò si traduce in un alto tempo medio di latenza di rotazione:

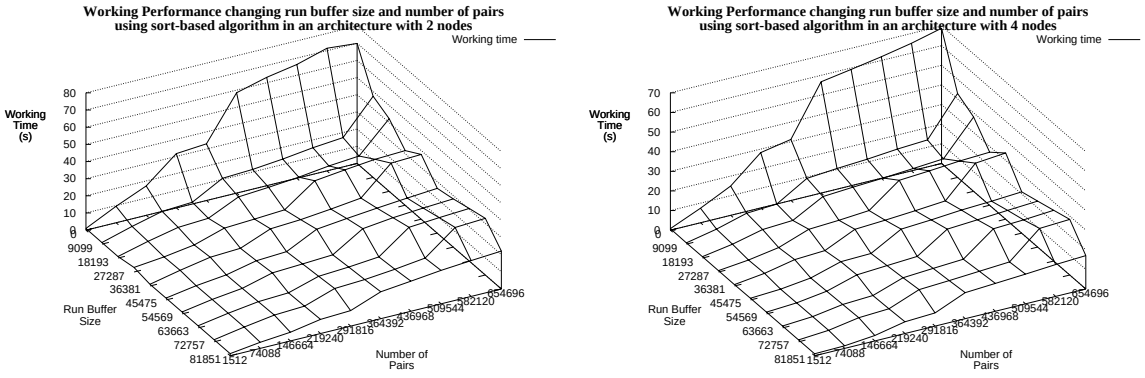
- un disco da 5400RPM ha una latenza di rotazione media di: (0,5 rotazioni / 5400 RPM) * (60 secondi * 1000 ms) = 5,6 ms.
- un disco di 7200RPM: (0,5 rotazioni / 7200 RPM) * (60 secondi * 1000 ms) = 4,2 ms.

Nel complesso, la versione sequenziale non ne beneficia dall'aumento del Communication Buffer Size indipendentemente dall'algoritmo di sorting utilizzato.

Variazione dimensione del buffer dei runs e del numero delle coppie. In tale test è stato mantenuto costante il Communication Buffer Size (ovvero 400 bytes) mentre è stato variato il Number of Pairs e il Run Buffer Size.

sort based simulato. L'esecuzione su 2 macchine è risultata leggermente più veloce: un minor numero di nodi coinvolti nella computazione, abbattono il communication overhead (vedi figura B.10).

sort based. Situazione opposta alla precedente: con un numero di nodi inferiori (nel nostro



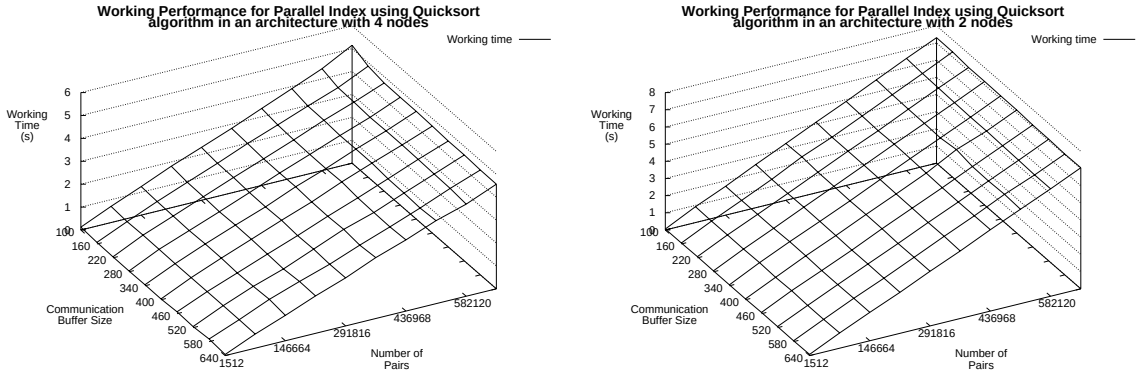
(a) (b)

Figura 5.4: Applicazione dell'algoritmo sort-based rispettivamente per 2 e 4 nodi variando il Number of Pairs e il Run Buffer Size.

caso 2) abbiamo ottenuto delle performance peggiori rispetto all'esecuzione su 4 macchine. Tale differenza è imputabile ad un rete molto carica durante lo svolgimento del test su 2 nodi. Poniamo in evidenza un significativo incremento di prestazioni passando da un Run Buffer Size di 5 fino a 27287 coppie, poi le performance si stabilizzano (vedi figura 5.4).

5.2 Versione Parallela

È stato accennato all'inizio di questo capitolo, che le variabili che entrano in gioco nel testing sono il numero di coppie, la dimensione del buffer dei run ed il buffer di comunicazione. Ci è sembrato interessante ampliare questa base aggiungendo a questi test, configurazioni leggermente differenti



(a) (b)

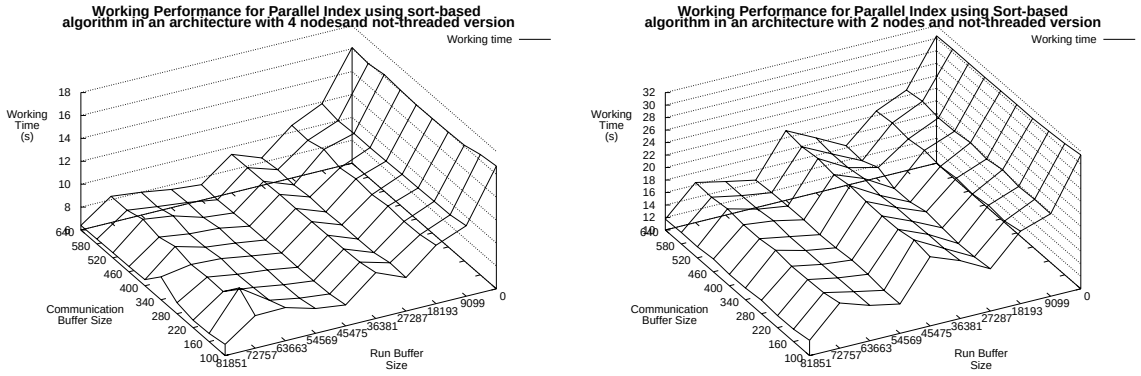
Figura 5.5: Andamento lineare crescente del tempo di computazione nel caso in cui si portino in memoria tutte coppie

dalla specifica originale che obbligava l'uso di un ordinamento esterno. Per questo motivo, si è approffondito il caso in cui tutte le coppie vengono trasferite in memoria ed ordinate con quicksort e mergesort ed il caso in cui si usi un algoritmo sort-based per così dire ibrido (questo infatti si differenzia da quello normale per il merging fatto tutto in memoria). L'architettura a ring è stata implementata in due modi diversi (threaded e not-threaded) ed anche questo ha portato ad un raddoppio dei dati ottenuti.

A questo punto, assieme alla presentazione e all'analisi dei grafici che abbiamo ritenuto più significativi (per l'elenco completo dei grafici si rimanda all'appendice B), inseriamo una carellata dei comportamenti comuni delle curve che rappresentano i tempi totali di esecuzione per poi passare, nella sezione successiva, alla descrizione dei grafici di speedup ed efficienza.

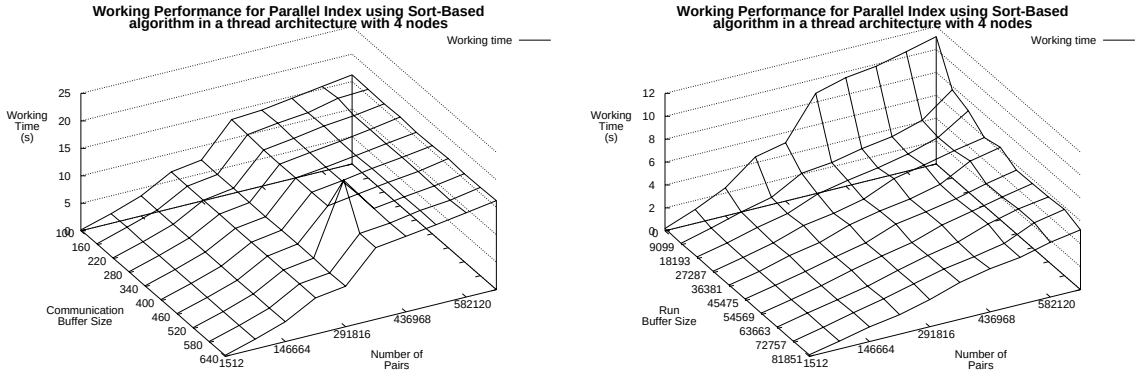
Il primo andamento riscontrato è quello **lineare**. Infatti, se osserviamo i grafici di figura 5.5 ci accorgiamo subito che indipendentemente dalla dimensione del buffer di comunicazione, i tempi totali di esecuzione crescono in maniera lineare. Il motivo di questo comportamento può essere meglio compreso se si pensa al fatto che tutti i dati vengono caricati in memoria ed ordinati (con quicksort o mergesort). I grafici di figura 5.6 rappresentano l'andamento delle performance nel caso di algoritmo di ordinamento sort-based. Come si può notare, dopo un forte decremento iniziale dei tempi fino a valori di run buffer pari a 27287 unità, l'andamento caratteristico è di tipo **ondulatorio**. Questo effetto deriva dalla scelta implementiva nell'algoritmo di ordinamento, in cui il numero di run devono essere potenza di due. I grafici per due e quattro nodi sono leggermente differenti perchè nei due casi il numero di coppie generate è diverso.

Il terzo blocco di grafici che presentiamo contiene altri due effetti caratteristici riscontrati. Il primo è un effetto **a gradini** (vedi figura 5.7 (a)), il secondo a **cappa** (vedi figura 5.7 (b)). Sebbene questi risultati sono stati rilevati con una configurazione di architettura dei nodi differente (a thread) rispetto ai grafici precedenti, l'andamento è comune anche nelle altre situazioni. Il primo effetto che si osserva nella figura, si riscontra per la scelta implementativa precedentemente accennata. In questo caso, tutti i test con $2^b \leq \mathcal{T}/MAX\_CP \leq 2^{b+1}$ ordineranno lo stesso numero di coppie, da cui l'effetto a gradini (esponenziale). Il secondo effetto è classico nel caso in cui mettiamo nell'asse delle x la dimensione del run buffer e in quello delle y il numero delle coppie. Come ci si può aspettare il tempo totale di esecuzione, aumenta all'aumentare del



(a) (b)

Figura 5.6: Variando il buffer di comunicazione non si migliorano i tempi di elaborazione



(a) (b)

Figura 5.7: Performance dell'architettura a thread. L'algoritmo di ordinamento utilizzato durante la prima fase del sort-based è il quicksort.

numero di coppie da elaborare e diminuisce quando abbiamo un run buffer più capiente.

Il grafico di figura 5.6 (a) rappresenta la medesima situazione di quello in figura 5.8. È interessante notare come la versione a thread compensi lo squilibrio nei tempi generato dalla versione not-threaded, producendo un grafico leggermente più lineare.

La versione a thread, sebbene non ottimizzata in maniera efficiente, fa diminuire i tempi di esecuzione totali secondo quanto riportato in tabella 5.1.

Un'ultima caratteristica che bisogna sottolineare è che frequentemente nei grafici si possono notare dei valori che si discostano dai valori dei vicini. Questo è molto probabilmente dovuto al carico della rete.

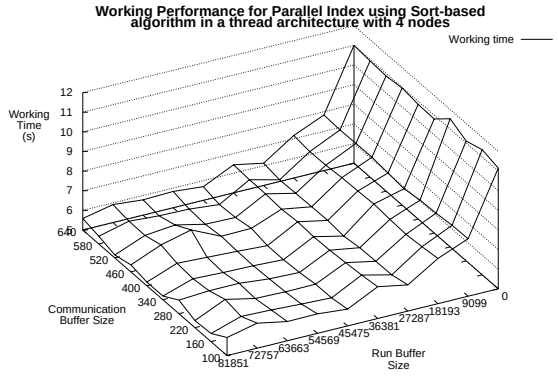


Figura 5.8: Performance dell'architettura a thread. L'algoritmo di ordinamento utilizzato durante la prima fase del sort-based è il quicksort.

Run Dim Buffer	Maxcp	Comm Dim Buffer	guadagno %
5	var.	var.	-32,26%
var	var.	400	-45%
var	654.696	var.	-31,25%

Tabella 5.1: Guadagni in termini percentuali dell'architettura a thread. Il guadagno è calcolato sullo step più dispendioso in termini di tempo

5.3 Speedup ed efficienza

Negli ultimi decenni si è assistito ad un grande aumento di potenza computazionale da parte dei processori, con ripercussioni drammatiche nel rapporto cpu-ram: bottleneck verso la memoria e incapacità, senza usare cache più grandi, di fornire dati al processore al rate richiesto.

Per arginare tali criticità, si ricorre al parallelismo riconosciuto come la tecnica per aumentare le prestazioni: diminuzione del tempo d'esecuzione, memoria aggregata ma soprattutto maggiore banda aggregata del bus front side bus. Per comparare le performance di un'applicazione sequenziale rispetto alla controparte parallela si fa uso di due specifiche funzioni, lo **speedup** (Sp(n)) e l'**efficienza** (E(n)).

Si definisce Speedup, come il tasso del tempo d'esecuzione del migliore algoritmo sequenziale (nel nostro caso la versione sequenziale del progetto) rispetto alla configurazione parallela. Ovvero:

$$Sp(n) = T_s/T_p(n)$$

dove n è il numero di processori, T_s è il tempo impiegato dall'algoritmo sequenziale e T_p il tempo riferito all'algoritmo parallelo. Il massimo speedup ottenibile, Speedup lineare, è definito come: $Sp(n) = n$.

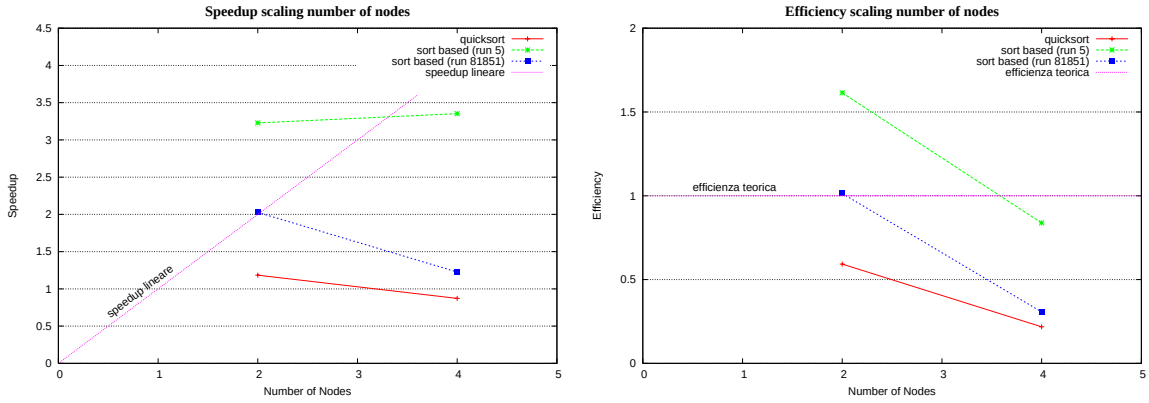
L'altro parametro molto importante è l'**efficienza**. Tale indice fornisce un'indicazione di quanto profiquamente i processori hanno partecipato all'esecuzione dell'algoritmo parallelo; una efficienza pari a 1 (ottenuta con speedup lineare $Sp(n) = n$) testimonia che tutte le cpu sono

state efficacemente usate nella computazione. L'efficienza è definita come:

$$E(n) = T_s/(n * T_p(n)) = Sp(n)/n$$

Nei grafici che seguiranno, studieremo dapprima la variazione dello speedup e dell'efficienza modificando il numero di macchine coinvolte, più precisamente 2 e 4, e gli algoritmi di sorting (quicksort e sort based) mantenendo costante il numero di coppie (Number Of Pairs uguale a 654696) e la dimensione del buffer della comunicatione (Communication Buffer Size). Successivamente, analizzeremo la variazione dello Speedup e dell'Efficienza cambiando il numero di coppie (Number Of Pairs) in un range di 1512-654696 con icremento di 72576, tendo fisso il numero di nodi nella computazione e il Communication Buffer Size.

Speedup - Efficienza variando il numero di nodi. Il più alto speedup e di conseguenza la migliore efficienza sia per 2 che 4 macchine, è registrata facendo uso dell'algoritmo di sorting esterno con una dimensione del buffer di run pari a 5 coppie. In generale, aumentando il numero di nodi non otteniamo aumenti di speedup. Tale scenario di testing è stato svolto tenendo costante il numero di coppie (Number Of Pairs a 654696) e la dimensione del buffer di comunicazione (Communication Buffer Size); sono stati presi in esame l'algoritmo di ordinamento quicksort e due versioni del sort based inversion, una con un Run Buffer Size pari a 5 coppie e l'altra pari a 81851.

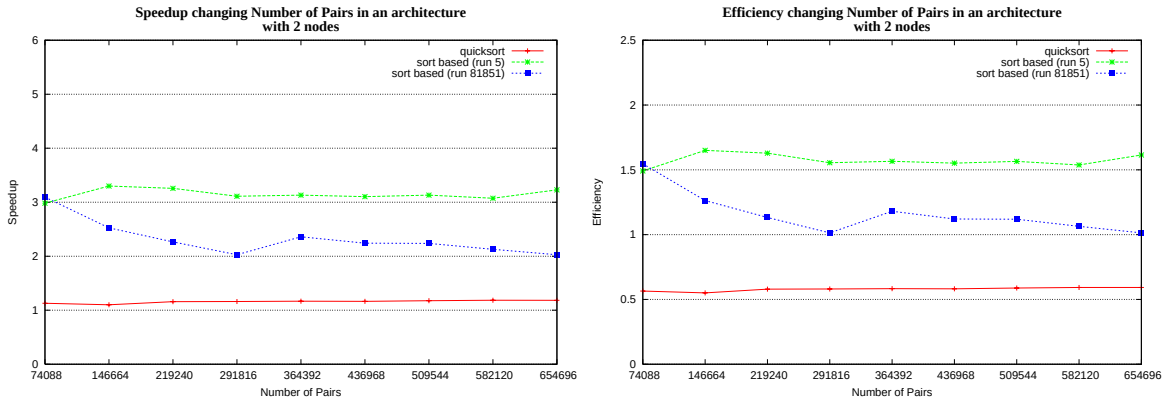


(a) (b)

Figura 5.9: Speedup ed efficienza al variare del numero di nodi

Speedup - Efficienza su 2 nodi variando il numero di coppie. A partire da 74088 coppie, all'interno dello stesso algoritmo di sorting, lo Speedup e l'Efficienza rimangono costanti. Valori di Speedup maggiori (versione parallela è 3 volte più veloce rispettto alla versione sequenziale) sono stati ottenuti utilizzando il sort based inversion con Run Buffer Size di 5 coppie. Il test si è svolto tenendo costante il numero di nodi ovvero 2, il Communication Buffer Size, e variando il Number Of Pairs tra 1512 e 654696 con incrementi di 72576 coppie; sono stati presi in esame l'algoritmo di ordinamento quicksort e due versioni del sort based inversion, una con un Run Buffer Size pari a 5 coppie e l'altra pari a 81851.

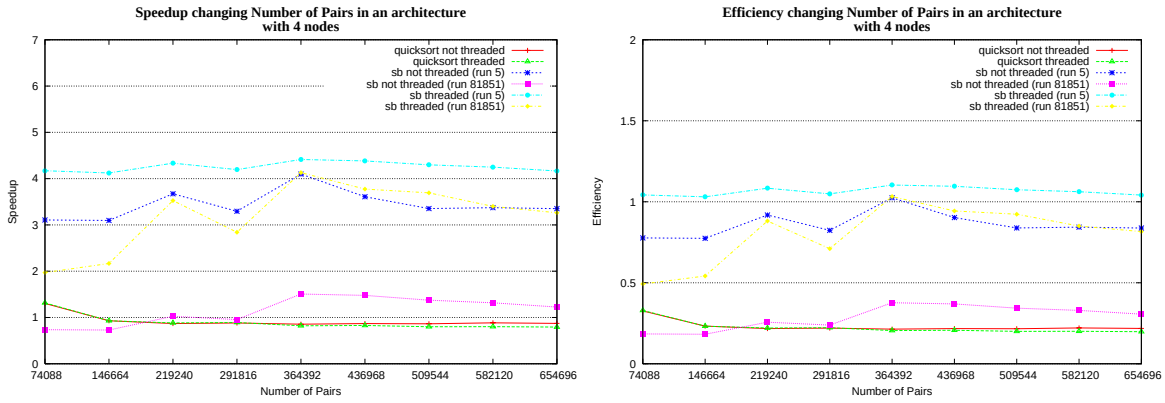
Speedup - Efficienza su 4 nodi variando il numero di coppie. I risultati ottenuti sono i medesimi del test effettuato in precedenza. Tuttavia, i migliori speedup ed efficienze si registrano nel progetto in versione parallela a thread, con top score della versione utilizzante il sort based inversion con Run Buffer Size pari a 5 coppie. Il test ha preso in esame la versione



(a) (b)

Figura 5.10: Speedup ed efficienza al variare del numero di coppie (2 nodi)

sequenziale e parallela con e senza thread, con algoritmi di sorting quicksort e due versioni del sort based inversion, una con un Run Buffer Size pari a 5 coppie e l'altra pari a 81851. Il numero di nodi coinvolti nella computazione è pari a 4; si è mantenuto costante il Communication Buffer Size, mentre è stato viariato il Number Of Pairs tra 1512 e 654696 con incrementi di 72576 coppie.



(a) (b)

Figura 5.11: Speedup ed efficienza al variare del numero di coppie (4 nodi)

5.4 Prestazioni sort-based simulato

Con scopo puramente didattico, nel progetto è stato riprodotto l'algoritmo esterno *sort-based* attraverso un'emulazione in memoria. L'intento è stato quello di enfatizzare il diverso utilizzo della memoria - disco. Le misurazioni sono state effettuate mediante KSysguard 1.2.0.

Sort-based *inversion* simulato:

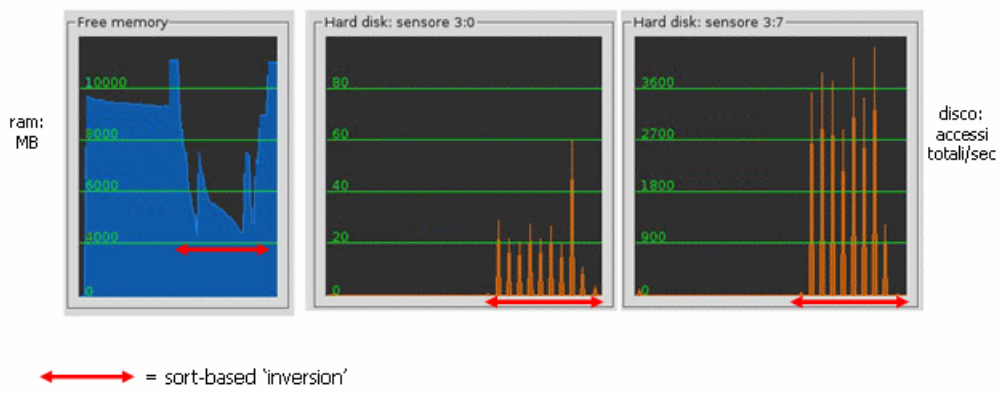


Figura 5.12: Occupazione di memoria, accessi al disco nella versione sort-based.

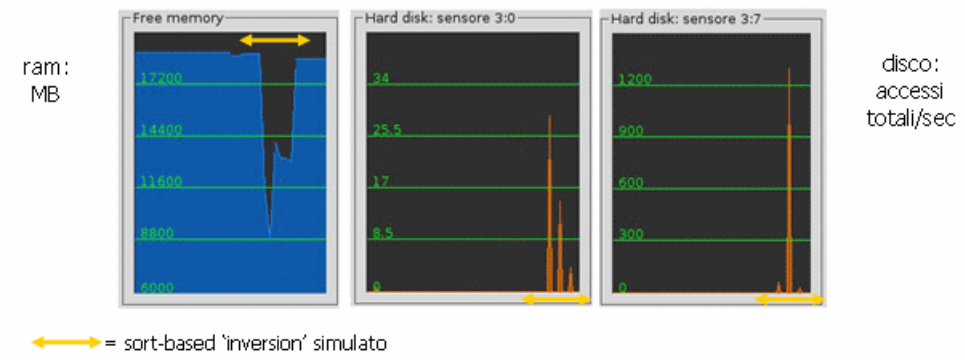


Figura 5.13: Occupazione di memoria, accessi al disco nella versione sort-based simulato.

Appendice A

Script di supporto

La versione parallela di questo progetto, come è stato ampiamente descritto in questa relazione, ha la possibilità di recuperare, durante la fase iniziale del programma, i parametri di esecuzione (MAX_CP, MAX_TERM, MAX_DOC) memorizzati in un file. Al fine di automatizzare questo processo, sono stati creati alcuni script che andremo a descrivere brevemente.

Il primo script, denominato `creaParametri` presenta la seguente schermata di help:

```
CreaParametri.sh
Sono necessari, in ordine:
1) numero di file dei parametri da creare
2) maxTerm
3) maxDoc
4) maxCP
5) nome delle macchine disponibili
```

Lo script è stato creato al fine di creare correttamente il file dei parametri in ogni macchina in cui viene lanciato il programma. Dato che la distribuzione dei termini è uniforme, allora lo script creerà p file di parametri in cui $\mathcal{T}^i \in [i * MAX\_TERM/p, (i+1) * MAX\_TERM/p - 1]$ con $i = 1, \ldots, p$.

Oltre a questo script, sono stati creati script per la scelta automatica delle macchine a disposizione, per la cancellazione automatica dei file risultato della computazione ed infine per l'esecuzione automatica dei test. Per quanto riguarda la scelta automatica delle macchine a disposizione nel laboratorio, la schermata di help presenta:

```
ScegliMacchine:
parametri necessari (in questo ordine):

1) nome del file che contiene tutti gli ip delle macchine da testare
2) numero delle macchine che si vogliono testare
3) nome del file che conterra' l'ip dei computer disponibili
```

All'interno dello script, per verificare se un computer è disponibile per una computazione remota, viene testato se il servizio sftp è attivo (il servizio denota che nella macchina è in esecuzione Linux). Lo script termina se sono state trovate un numero di macchine inferiore o pari al secondo parametro.

Infine, gli script che abbiamo utilizzato per fare i test, variano in base al tipo di dati che si vogliono inserire nel grafico 3D dei risultati. Per fare un esempio, lo script che gestisce la variazione della dimensione del buffer di comunicazione e del numero delle coppie, accetterà i seguenti parametri:

```
1)  indexStart Coppie
2) indexStop Coppie
3) indexStart Buff mess
4) indexStop Buff mess
5) step coppie
6) step buffer messaggi
7) numero macchine da usare
8) dimensione del buffer di run
```

ed avrà una struttura simile alla seguente:

```
let indexCP=$1
let indexBuff=$3
while [ $indexBuff -le $4 ]; do
  while [ $indexCP -le $2 ]; do
    mpirun -np $7 main $indexBuff $indexCP $8
    let indexCP=indexCP+$5
  done
 let indexCP=$1
 let indexBuff=indexBuff+$6
done
```

Bibliografia

[1] Algoritmi di ordinamento.
`http://www.die.supsi.ch/~bianchil/algoritmi_numerica/slides_leo/ordinamento.pdf`.

[2] Algoritmi di ordinamento e ricerca: una guida concisa.
`http://epaperpress.com/sortsearchItalian/index.html`.

[3] Algoritmo mergesort.
`http://it.wikipedia.org/wiki/Merge_sort`.

[4] Algoritmo quicksort.
`http://it.wikipedia.org/wiki/Quicksort`.

[5] Classificazione degli algoritmi di ordinamento.
`http://it.wikipedia.org/wiki/Algoritmo_di_ordinamento`.

[6] Mergesort.
`http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm`.

[7] Antonio Albano. *Costruire Sistemi per Basi di Dati*. Addison-Wesley, 2001.

[8] Barney Blaise. Posix threads programming.
`http://www.llnl.gov/computing/tutorials/pthreads/`.

[9] Michele Bugliesi. Algoritmi di ordinamento e tecniche divide-et-impera. *Corso Algoritmi e strutture dati*, a.a. 2001/02.

[10] Foster. *Designing and Building Parallel Programs*. Addison Wesley, 1995.

[11] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing, Second Edition*. Addison Wesley, 2003.

[12] Renzo Orsini. Realizzazione operatori relazionali. *Corso Basi di Dati II*, a.a. 2004/05.

[13] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[14] Andrew S. Tanenbaum. *Reti di computer Terza Edizione*. Prentice-Hall, 1996.

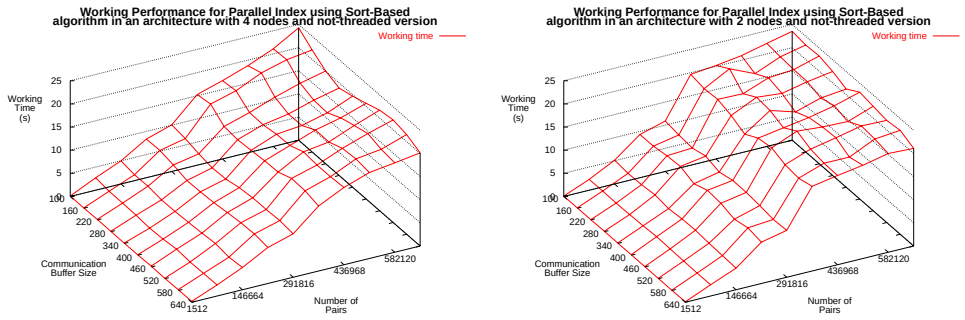
Appendice B

Grafici

Per ragioni di completezza, in questo appendice verranno presentati tutti i grafici prodotti durante la fase di testing del progetto. Per una spiegazione più approffondita, si rimanda ai capitoli precedenti.

B.1 Versione parallela

Per quanto concerne la versione parallela del progetto, si può fare riferimento alla tabella B.1 per avere una visione riassuntiva dei grafici.



(a) (b)

Figura B.1: Nell'ordine: grafico test 1 e 2

Nome test	Lab	Nro nodi	Maxcp	Maxdoc ∀ nodo	Dim buff msg	Run dim buff	Metodo di Ordinamento					Architettura Ring	
							QS	MS	HS	SB Sim	SB not-Sim	Thread	No Thread
1	14	4	378/163674	300	100/640	5					•		•
2	14	2	756/327348	600	100/640	5					•		•
3	14	2	756/327348	600	400	5/81851					•		•
4	14	4	378/163674	300	400	5/81851					•		•
5	5	4	163674	300	100/640	5/81851					•		•
6	5	2	327348	600	100/640	5/81851					•		•
7	5	4	378/163674	300	100/640	5					•	•	
8	5	4	378/163674	300	400	5/81851					•	•	
9	5	4	163674	300	100/640	5/81851					•	•	
10	5	4	378/163674	300	100/640	5	•					•	
11	5	4	378/163674	300	100/640	5		•				•	
1a	5	4	378/163674	300	100/640	5	•						•
2a	5	2	756/327348	600	100/640	5	•						•
1b	5	4	378/163674	300	100/640	5		•					•
2b	5	2	756/327348	600	100/640	5		•					•
1c	5	4	378/163674	300	100/640	5				•			•
2c	5	2	756/327348	600	100/640	5				•			•
3c	5	2	756/327348	600	400	5/81851				•			•
4c	5	4	378/163674	300	400	5/81851				•			•
5c	5	4	163674	300	100/640	5/81851				•			•
6c	5	2	327348	600	100/640	5/81851				•			•

Tabella B.1: Tabella comparativa dei test effettuati al variare dei parametri scelti per la versione parallela del progetto (QS=QuickSort, MS=MergeSort, HS=HeapSort, SB=Sort-Based)

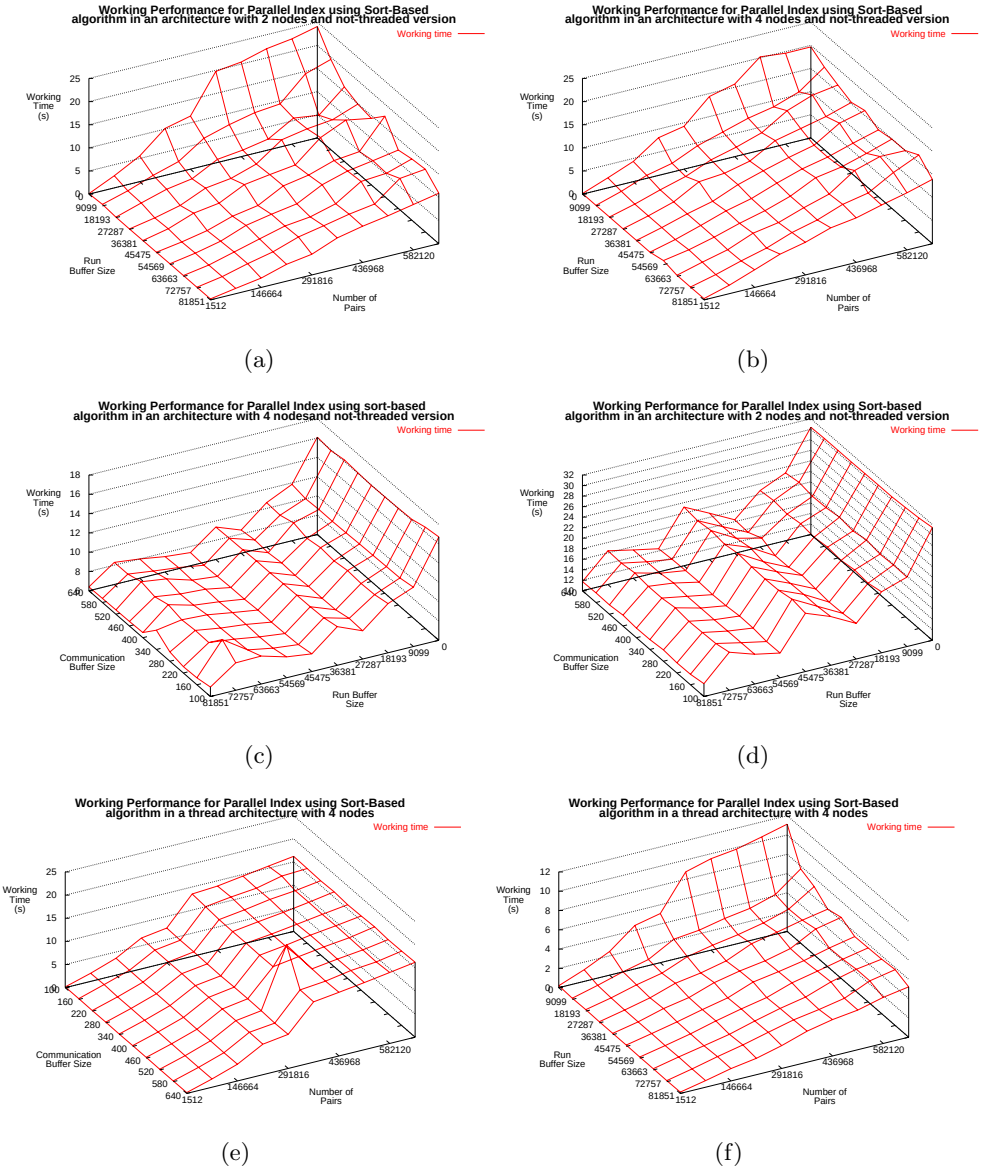
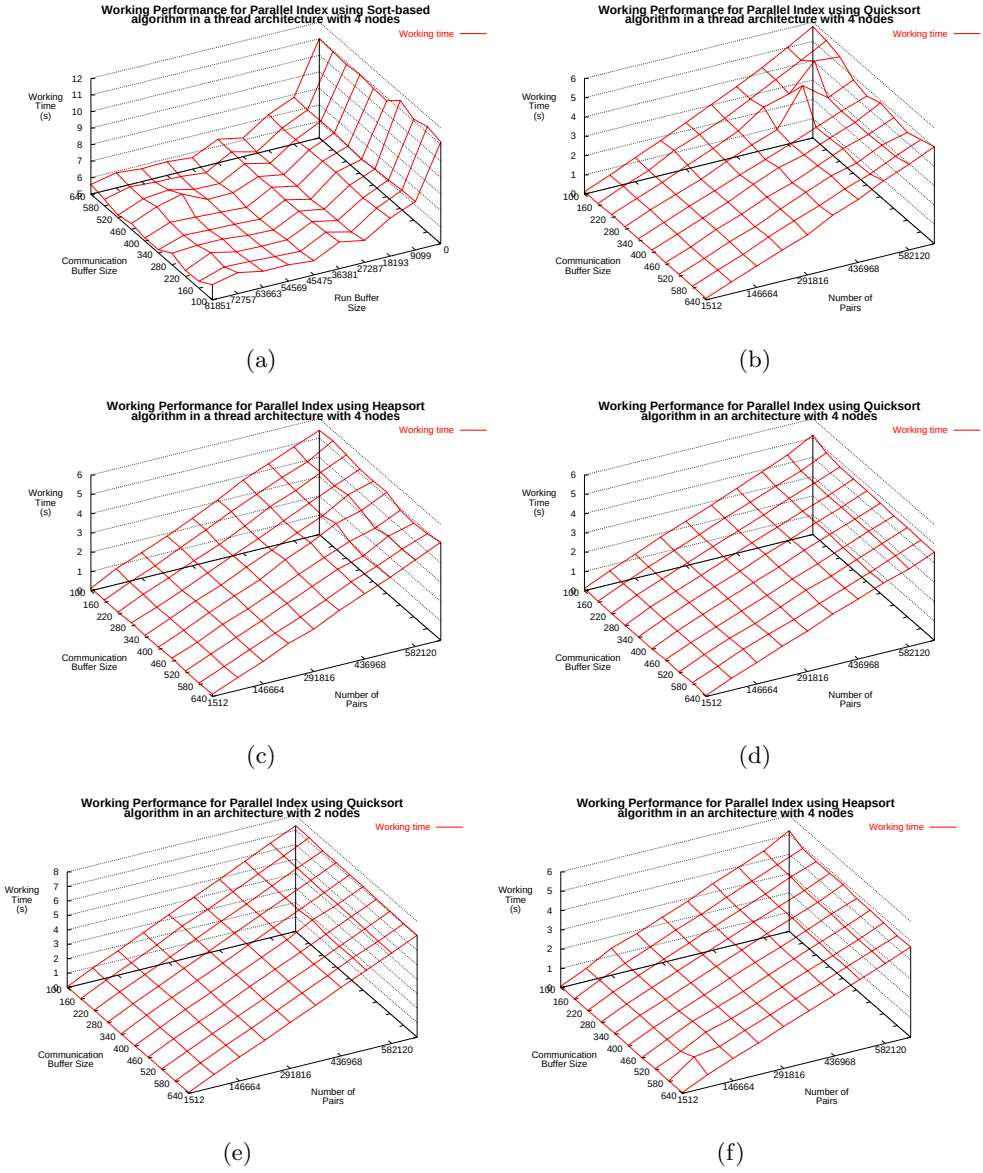


Figura B.2: Nell'ordine: grafico test 3, 4, 5, 6, 7, 8



(a) (b)

(c) (d)

(e) (f)

Figura B.3: Nell'ordine: grafico test 9, 10, 11, 1a, 2a, 1b

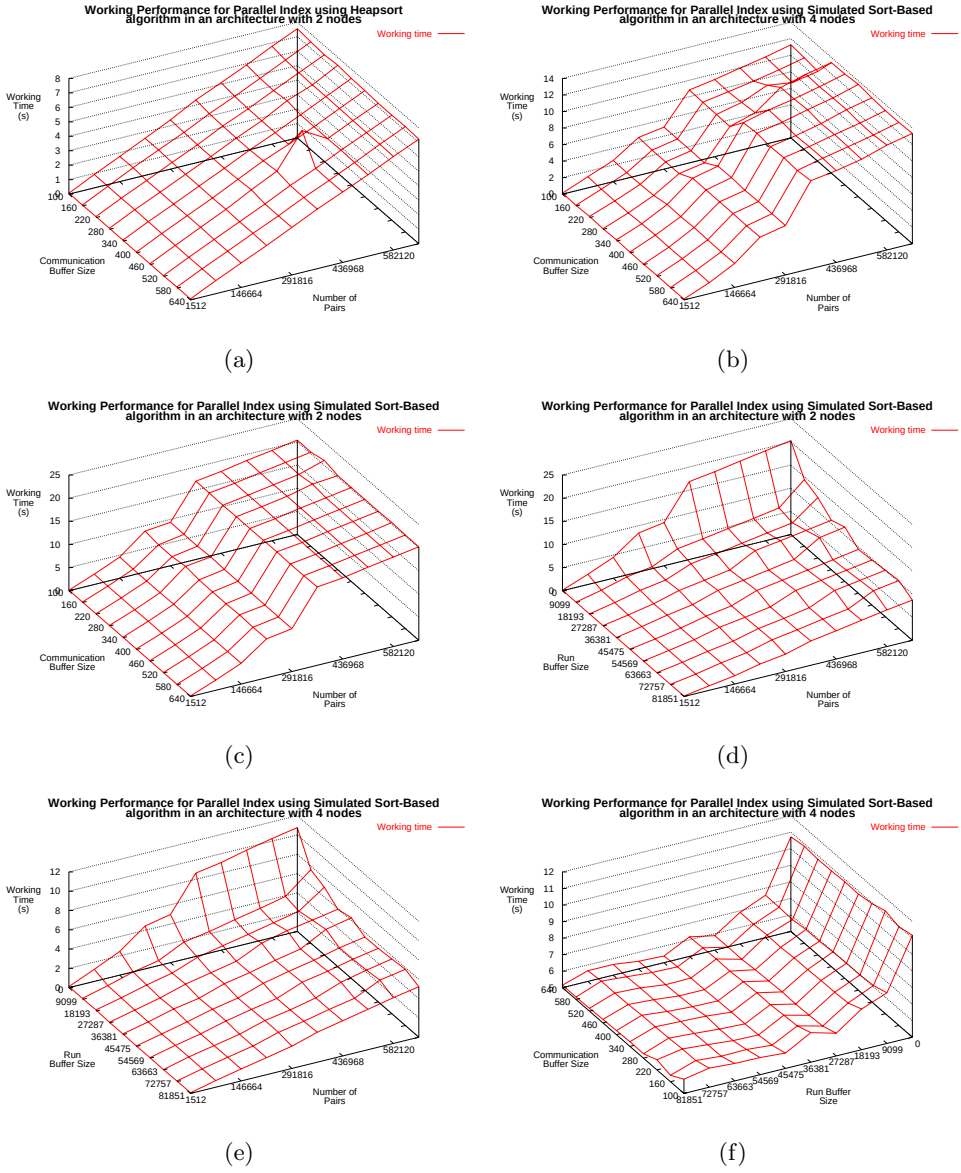
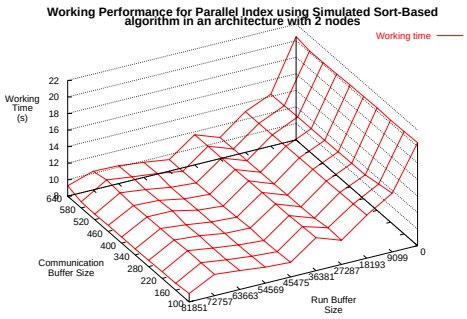


Figura B.4: Nell'ordine: grafico test 2b, 1c, 2c, 3c, 4c, 5c



(a)

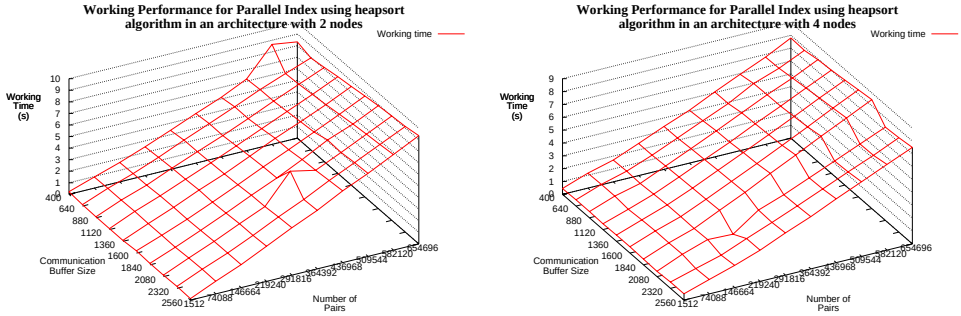
Figura B.5: Grafico test 6c

B.2 Versione sequenziale

I grafici che rappresentano i test di questa versione del progetto sono stati divisi in base ai parametri presi come riferimento.

B.2.1 A

Tempi totali di esecuzione al variare della dimensione del buffer di comunicazione e del numero di coppie.



(a) (b)

Figura B.6: Nell'ordine: heapsort 2 e 4

Nome test	Lab	Nro nodi	Maxcp	Maxdoc nodo root	Dim buff msg	Run dim buff	Metodo di Ordinamento				
							QS	MS	HS	SB Sim	SB not-Sim
Asort2	5	2	1512/654696	1200	400/2560	5					•
Asort4	5	4	1512/654696	1200	400/2560	5					•
quicksort2	5	2	1512/654696	1200	400/2560	5	•				
quicksort4	5	4	1512/654696	1200	400/2560	5	•				
mergesort2	5	2	1512/654696	1200	400/2560	5		•			
mergesort4	5	4	1512/654696	1200	400/2560	5		•			
heapsort2	5	2	1512/654696	1200	400/2560	5			•		
heapsort4	5	4	1512/654696	1200	400/2560	5			•		
simulato2	5	2	1512/654696	1200	400/2560	5				•	
simulato4	5	4	1512/654696	1200	400/2560	5				•	
B1sortbased2	5	2	654696	1200	400/2560	5/81851					•
B1sortbased4	5	4	654696	1200	400/2560	5/81851					•
B1simulato2	5	2	654696	1200	400/2560	5/81851				•	
B1simulato4	5	4	654696	1200	400/2560	5/81851				•	
gB2sortbased2	5	2	1512/654696	1200	400	5/81851					•
gB2sortbased4	5	4	1512/654696	1200	400	5/81851					•
gB2simulato2	5	2	1512/654696	1200	400	5/81851				•	
gB2simulato4	5	4	1512/654696	1200	400	5/81851				•	

Tabella B.2: Tabella comparativa dei test effettuati al variare dei parametri scelti per la versione sequenziale del progetto (QS=QuickSort, MS=MergeSort, HS=HeapSort, SB=Sort-Based)

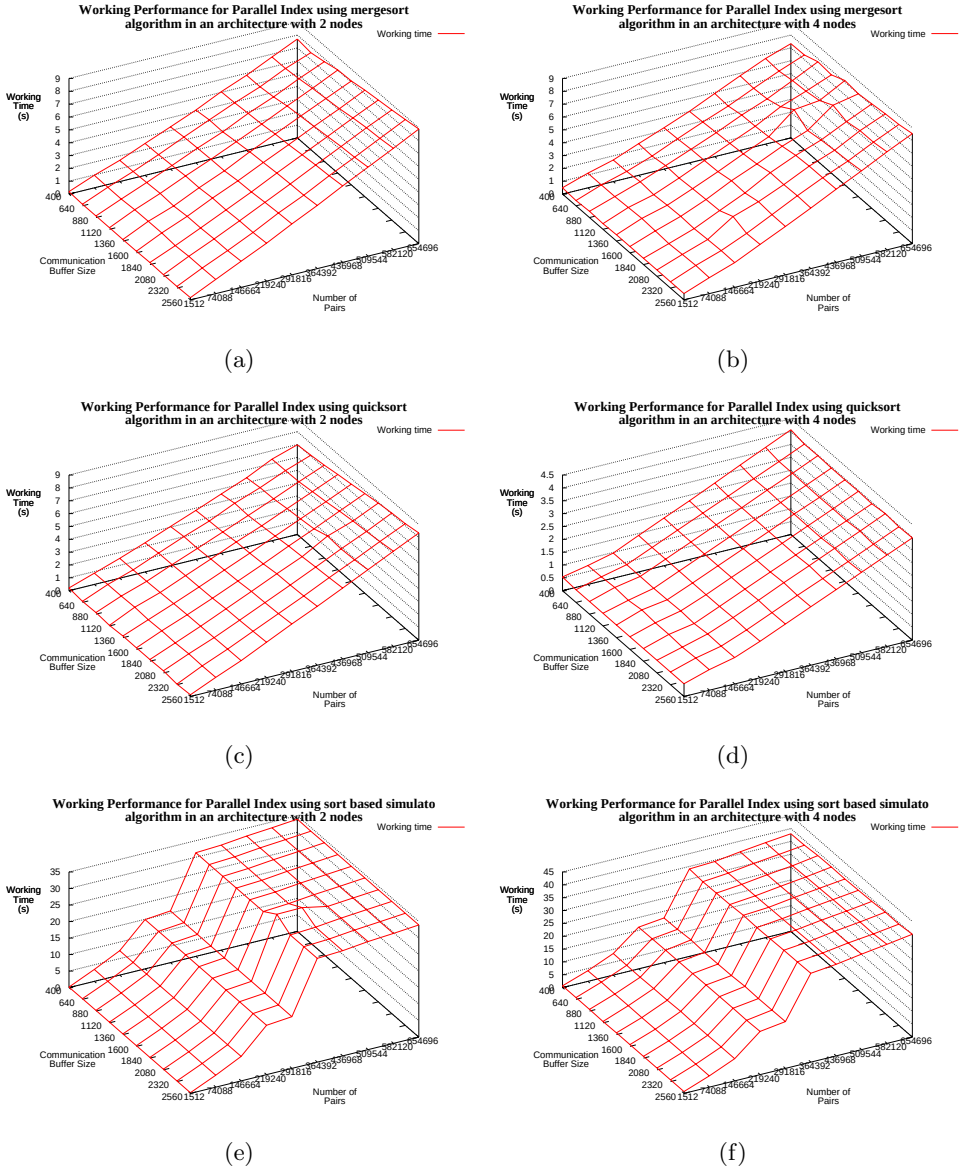
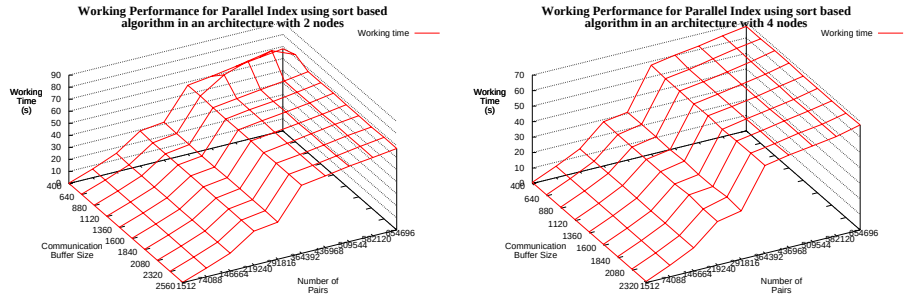


Figura B.7: Nell'ordine: mergesort 2 e 4, quicksort 2 e 4, simulato 2 e 4

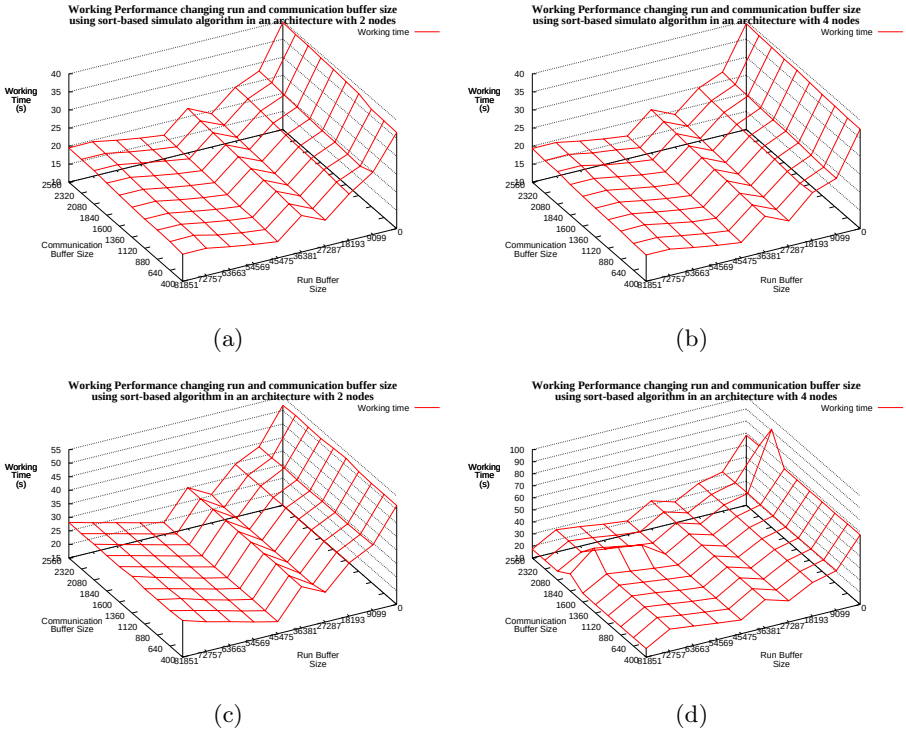


(a) (b)

Figura B.8: Nell'ordine: sort-based 2 e 4

B.2.2 B1

Tempi totali di esecuzione al variare della dimensione del buffer di comunicazione e della dimensione del buffer di run.



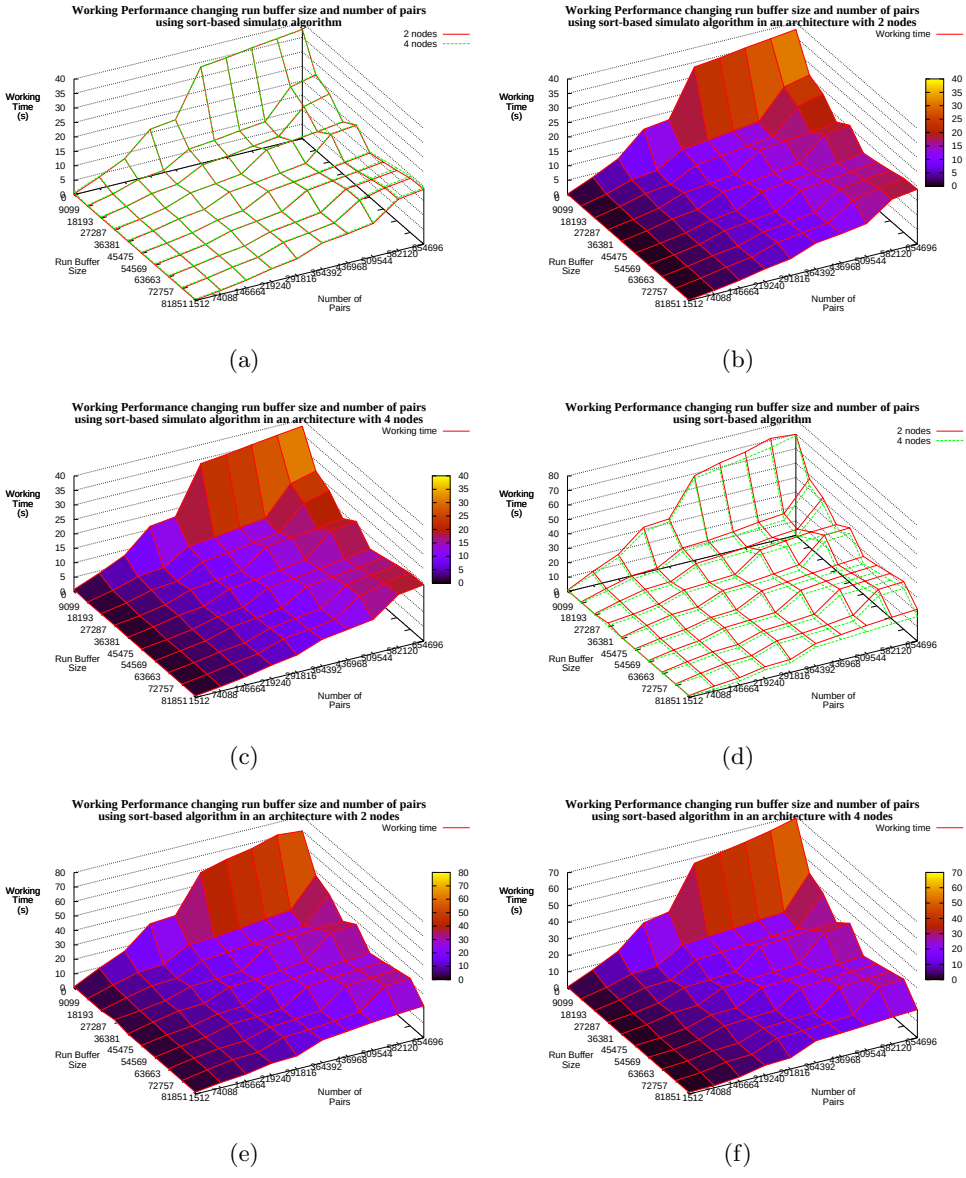
(a) (b)

(c) (d)

Figura B.9: Nell'ordine: simulato 2 e 4, sort-based 2 e 4

B.2.3 B2

Tempi totali di esecuzione al variare della dimensione del buffer di run e del numero di coppie.



(a) (b) (c) (d) (e) (f)

Figura B.10: Nell'ordine: simulato 2 e 4, sort-based 2 e 4