Università degli Studi di Padova

Corso di Laurea Magistrale
in Scienze Statistiche

TESI DI LAUREA

# SVILUPPO E IMPLEMENTAZIONE DI UN ALGORITMO PARALLELO PER MODELLI CART

Relatore: Prof. BRUNO SCARPA
Dipartimento di Scienze Statistiche

Laureando: MICHEL COMAZZETTO

ANNO ACCADEMICO 2012/2013

*Ai miei genitori, a mia sorella Katty, ai 'fioi' e a tutti quelli che ci sono, o ci sono stati, nella mia vita.*

# Indice

# Elenco delle figure

# Elenco delle tabelle

# Introduzione

Nell'ambito del *data mining* avere sufficienti quantità di dati per poter effettuare analisi e previsioni non è un problema. Dopotutto, lo sviluppo di questa disciplina, è dovuta proprio al fatto che i metodi statistici tradizionali si trovavano in difficoltà a maneggiare quantità di dati molto grandi. Tuttavia, la sempre crescente capacità di raccogliere ed immagazzinare dati sta portando ad evidenziare limiti presenti in questi algoritmi, e si stanno quindi cercando soluzioni per aggirare questi problemi.

Il recente sviluppo tecnologico ha portato a disporre di strumenti per la memorizzazione capaci di tenere in memoria una quantità di dati che fino ad una decina d'anni fa era impensabile e soprattutto permettono di farlo a prezzi contenuti. La raccolta di dati quindi è cresciuta in modo esponenziale, ma i metodi per l'estrazione di informazione da questi dati non sempre sono stati adattati a questa continua crescita.

Nell'ultimo periodo si sta cercando di offrire delle implementazioni alternative ai classici strumenti di *data mining* sfruttando le caratteristiche del *calcolo parallelo*. Per fare ciò esistono due correnti di pensiero principali: la prima riguarda l'utilizzo dell'**algebra delle matrici** per velocizzare tutta la parte computazionale che deve essere eseguita, la seconda è quella dello sviluppo di **algoritmi paralleli**.

La prima strada prevede la parallelizzazione della parte riguardante l'utilizzo dell'algebra delle matrici per suddividere le operazioni computazionalmente costose, per esempio l'inversione di matrici, per fare in modo che senza dover modificare la logica sequenziale dell'algoritmo, vi sia un incremento delle prestazioni semplicemente suddividendo i calcoli su più processori. In questo caso un processo/*thread* si occuperà di portare avanti le istruzioni dell'algoritmo mentre altri *thread* saranno attivati nel momento in cui si necessita di una maggior potenza di calcolo. Questo primo aspetto

viene implementato in molti software. Il secondo metodo, meno frequente nelle implementazioni software, prevede di sfruttare l'evoluzione delle tecnologie hardware, avuta di recente con la sempre maggior diffusione di computer multi-processore e la creazione di cluster di computer, per sfruttare, con un algoritmo modificato, la potenza totale di calcolo distribuita su più computer o più processori.

Suddividere il carico di lavoro su più processori può portare ad un significativo vantaggio nel caso in cui si debba lavorare con una quantità di dati molto grande.

L'obiettivo di questa tesi è quello di implementare in parallelo uno degli strumenti di *data mining* più utilizzati, l'albero di classificazione e regressione, ed analizzarne i benefici in termini di correttezza dei risultati e tempi di esecuzione. Alla luce dei risultati si potrà quindi valutare se la strada della parallelizzazione degli algoritmi sia effettivamente valida e porti ad avere dei vantaggi nel caso essa venga applicata anche anche ad altri algoritmi di *data mining*.

Il Capitolo 1 è volto a dare una breve introduzione all'argomento del calcolo parallelo partendo da un punto di vista puramente informatico. Saranno infatti elencate quali sono le caratteristiche che devono avere le macchine per permettere una corretta implementazione del calcolo parallelo. Vedremo quali sono le architetture disponibili e come può essere organizzata la memoria. Verrà illustrato un metodo generale per la parallelizzazione di algoritmi e saranno presentati i modelli di programmazione parallela che vengono comunemente utilizzati. Infine, verranno descritti gli indicatori che possono essere calcolati per misurare l'efficienza della parallelizzazione.

Nel Capitolo 2 la nostra attenzione si sposterà sul *data mining* e sul collegamento esistente con il calcolo parallelo. Vedremo perché può essere utile l'applicazione in questo ambito e verranno illustrate alcune strategie per il *data mining* parallelo. Infine, verrà presentata la parallelizzazione utilizzando il *multithreading*, metodo che sarà poi utilizzato per l'implementazione del nostro algoritmo.

Il Capitolo 3 offrirà una breve panoramica sui vari software che ci permettono di sfruttare il calcolo parallelo associato a strumenti di *data mining*. Verrà posta particolare attenzione all'ambiente di programmazione R e al linguaggio di programmazione C. Verrà offerta una veloce introduzione alle funzioni implementate nei diversi linguaggi con una breve analisi su vantaggi e svantaggi che si avrebbero nell'applicazione a casi reali.

Il Capitolo 4, presenterà l'algoritmo implementato per sfruttare il calcolo parallelo. Inizialmente vi sarà una breve analisi sulle diverse tipologie di alberi di classificazione e regressione e successivamente un approfondimento sulla tipologia CART che è stata scelta per l'implementazione. Saranno poi presentate le idee che stanno alla base del lavoro di tesi e le decisioni prese per lo sviluppo dell'algoritmo parallelo.

Nel Capitolo 5, utilizzando alcuni dataset di esempio, saranno effettuati dei confronti grafici per valutare la correttezza delle suddivisioni effettuate dall'algoritmo parallelo implementato e successivamente verrà valutato l'effettivo miglioramento ottenuto con l'introduzione del calcolo parallelo rispetto all'algoritmo sequenziale.

L'ultima sezione sarà dedicata alle conclusioni e ai possibili sviluppi futuri che il calcolo parallelo ci può offrire.

In Appendice A verranno riportate le principali funzioni realizzate ed utilizzate per lo svolgimento del lavoro.

# Capitolo 1

# Calcolo Parallelo

## 1.1 Introduzione

La crescente potenza di calcolo messa a disposizione dai moderni calcolatori ha permesso, nel campo della modellazione, di fronteggiare problemi numerici dalla complessità sempre maggiore in tempi relativamente brevi (Campelli e Stagni, 2011). Inizialmente per aumentare le prestazioni dei processori si è seguita la strada dell'aumento della frequenza di *clock*. Si è visto però che all'aumentare della frequenza il consumo di questi processori cresceva in modo esponenziale, arrivando ben presto ad un limite fisico e, questo limite non poteva essere aggirato con l'utilizzo di processori single core. Per questo motivo, negli ultimi anni, l'attenzione si è spostata verso lo sviluppo di processori multi-core, i quali a frequenze decisamente più basse permettono di avere prestazioni simili ai più potenti processori single-core. Il radicale cambiamento avvenuto nel comparto hardware si è ripercosso anche nella parte software, infatti, fino all'introduzione di questi sistemi i software erano pensati per essere eseguiti 'sequenzialmente' su un singolo processore. Per poter sfruttare queste nuove tecnologie, è stato dunque necessario riprogettare il software per adattarsi alla nuova struttura parallela della CPU, in modo tale da ottenere un'efficienza maggiore tramite l'esecuzione contemporanea su più processori, di più parti dello stesso programma.

Nella sua definizione più semplice, con il termine *calcolo parallelo*, intendiamo l'uso simultaneo di molteplici risorse di calcolo per eseguire un programma. Per poter

sfruttare il calcolo parallelo sarà necessario modificare, oltre all'hardware, anche il codice scritto precedentemente per un esecuzione sequenziale. Il quale, deve essere ora riscritto per permettere di sfruttare appieno le risorse, allo scopo di velocizzare le operazioni di calcolo.

Questo incremento di risorse può essere ottenuto in differenti modi:

- utilizzando un computer con un processore multi-core;
- utilizzando un computer singolo con più processori;
- utilizzando più computer con un singolo processore collegati tra loro;
- implementando una combinazione di computer multi-processore collegati in rete.

In questa tesi, considereremo i sistemi multi-processore e multi-core come equivalenti anche se nella pratica sono implementazioni diverse dello stesso concetto. Le tipologia multi-core offre alcuni vantaggi rispetto ai sistemi multi-processore, per esempio permettono di utilizzare il *clock* a frequenze maggiori, hanno un uso più efficiente della memoria *cache*, offrono risposte migliori rispetto ai multi-processore quando sono sottoposti a carichi intensi, permettono di ridurre le dimensioni del circuito stampato e allo stesso tempo utilizzare correnti più basse. Di contro però, i sistemi multi-core richiedono delle modifiche ai programmi tali per cui la complessità generale nella gestione è maggiore rispetto ai sistemi multi-processore.

Le macchine sequenziali, basate per la maggior parte sul modello di Von Neumann (Pimentel e Vassiliadis, 2004), hanno una serie di limiti che la programmazione parallela ci permette di poter superare.

I limiti che un'architettura sequenziale presenta sono:

- **Limite della memoria**: singolo computer ha risorse di memoria finite;
- **Limite della velocità di trasmissione**: la velocità di un computer sequenziale è strettamente correlata alla velocità dei dati all'interno della struttura hardware. Questa velocità è limitata, e i suoi limiti sono rappresentati dalla velocità di trasmissione attraverso i conduttori;

- **Limite della miniaturizzazione**: la tecnologia dei processori consiste nell'aumentare il numero di transistor presenti in un chip riducendone le dimensioni, ed è evidente che vi siano dei limiti fisici alla miniaturizzazione;
- **Aspetti economici**: un singolo processore sempre più veloce ha costi superiori a quelli di più processori tradizionali che, opportunamente collegati, offrirebbero la stessa potenza e prestazioni. Questo perchè andare ad ottimizzare il processore singolo richiede un profondo studio su cosa si può e cosa non si può fare e dal punto di vista ingegneristico esistono dei limiti fisici vicini ai quali un minimo miglioramento richiede un grosso sforzo.

## 1.2 Architetture di memoria

Si possono definire vari tipi di computer paralleli distinti secondo il tipo di connessione tra i vari processori e tra i processori e la memoria. La classificazione più utilizzata è la *tassonomia di Flynn* che è stata introdotta già nel 1966.

La tassonomia di Flynn si basa sulla nozione di flusso di informazioni. Queste informazioni possono essere istruzioni o dati e la relazione tra le istruzioni del programma e i dati del programma permette di classificare le macchine in quattro categorie. (Agati, 2009)

Queste categorie sono elencate in seguito e nella Figura 1.1 possiamo vedere la suddivisione rispetto ai livelli di istruzioni e dati.

- **SISD** (*Single Instruction, Single Data*)
- **SIMD** (*Single Instruction, Multiple Data*)
- **MISD** (*Multiple Instruction, Single Data*)
- **MIMD** (*Multiple Instruction, Multiple Data*)

In Figura 1.2 possiamo invece vedere come ogni categoria possa essere suddivisa ulteriormente a seconda dell'organizzazione della memoria che possono essere implementate con esse. Questo aspetto risulterà molto importante, come vedremo nel Paragrafo 1.3, infatti categorie differenti saranno propense ad utilizzare organizzazioni di memoria di tipo diverso.

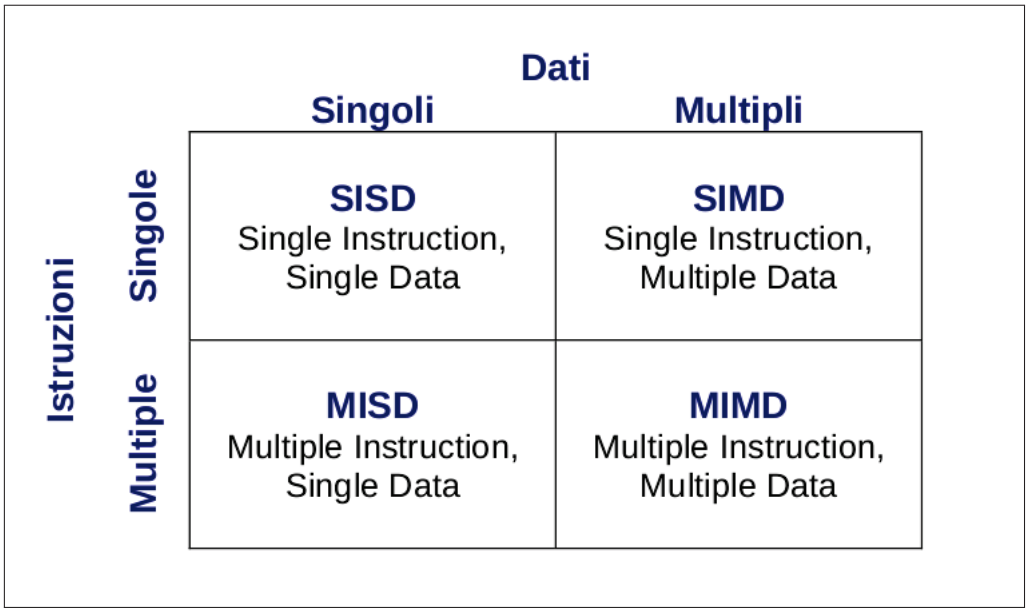


Figura 1.1. Tassonomia di Flynn

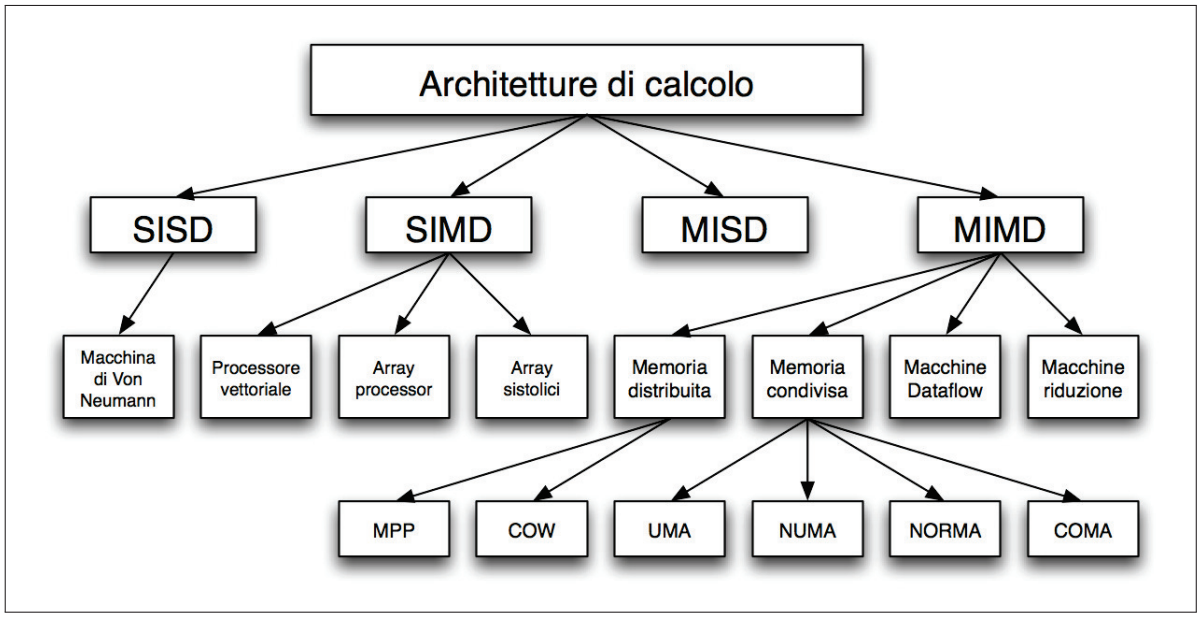


Figura 1.2. Architetture di calcolo

### Architettura *Single Instruction, Single Data* (SISD)

L'architettura SISD, come si vede in Figura 1.3 è rappresentata dalla classica struttura della macchina di Von Neumann, si tratta in pratica di un computer sequenziale. Un computer di questo tipo comprende una sola unità di elaborazione (ALU-*Arithmetic Logic Unit*), quest'ultima riceve un unico flusso di istruzioni e opera su un unico flusso di dati.

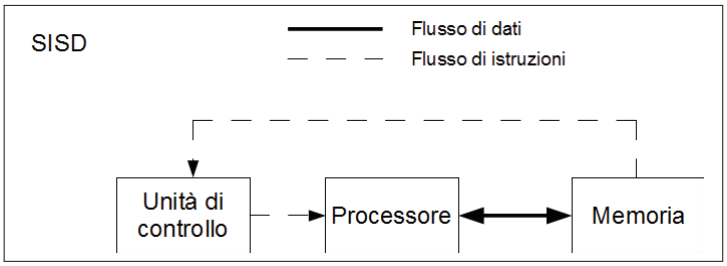


Figura 1.3. *Single Instruction Single Data* (SISD)

Ad ogni ciclo di *clock* avvengono in sequenza una serie di operazioni per l'elaborazione delle istruzioni:

- **Fetch**: viene prelevata dalla memoria centrale l'istruzione che deve essere eseguita;
- **Decode**: viene interpretata l'istruzione e stabilite le operazioni da eseguire;
- **Execute**: vengono eseguite le operazioni precedentemente stabilite.

La performance di queste macchine è misurata in MIPS (Milioni di istruzioni per secondo) o in Mhz (megahertz). Semplificando, potremmo dire che l'architettura è composta di tre parti: processore, *cache* e memoria.

### Architettura *Single Instruction, Multiple Data* (SIMD)

Questo tipo di architettura, Figura 1.4, rappresenta un computer multi-processore ovvero siamo in presenza di $n$ processori identici. Tutti i processori, ad ogni passo (quindi in modo sincrono), eseguono la stessa istruzione (Single Instruction) lo fanno però, su dati diversi. Questo è un esempio di parallelismo a livello dati. In questo

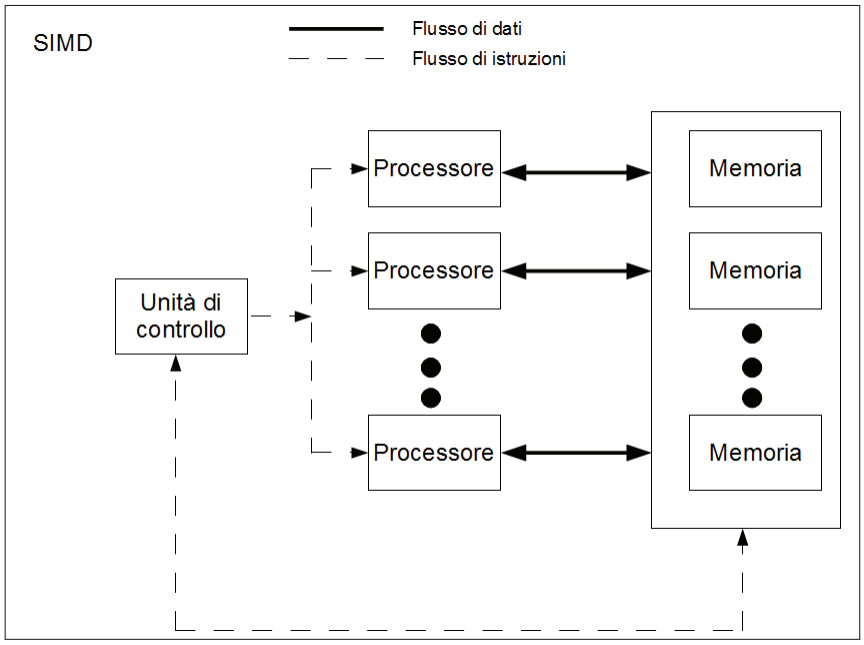


Figura 1.4. *Single Instruction Multiple Data* (SIMD)

tipo di architettura esiste una sola unità di controllo che organizza e distribuisce le istruzioni da inviare ad ogni CPU.

Una caratteristica importante che deve avere il problema da parallelizzare, utilizzando questo tipo di architettura, è che deve poter essere suddiviso in sotto-problemi identici, poiché le istruzioni che si vanno ad eseguire sono perfettamente identiche su tutti i dati. Questa richiesta non sempre può essere soddisfatta dal problema che stiamo andando ad affrontare, nel caso in cui questa strategia non sia applicabile al problema in esame, sarà necessario provare con soluzioni alternative, come per esempio la parallelizzazione MIMD. Le architetture SIMD stanno prendendo sempre più piede nelle applicazioni che riguardano elaborazioni grafiche (GPU) e nelle applicazioni multimediali.

### Architettura *Multiple Instruction, Single Data* (MISD)

L'architettura MISD rappresenta una delle strutture meno utilizzate nei computer moderni. Questa architettura prevede che un singolo flusso di dati alimenti più unità di elaborazione. Ogni unità opera sui dati indipendentemente con flussi di istruzioni differenti (Figura 1.5). Ad ogni ciclo di clock, il dato ricevuto dalla memoria viene elaborato da tutti i processori in maniera simultanea, ogni processore però utilizzerà le istruzioni che provengono dalla sua unità di controllo. Differenziando quindi i risultati in base alle istruzioni eseguite sui dati. Esistono pochissimi esempi di questo tipo di architettura, alcuni campi in cui questa architettura viene utilizzata sono:

- Filtraggio multiplo su un singolo segnale;
- Crack di un singolo segnale crittografato utilizzando più algoritmi.

### Architettura *Multiple Instruction, Multiple Data* (MIMD)

Si tratta del modello attualmente più diffuso per il calcolo parallelo, nonché il più potente. In questa architettura ogni processore esegue diverse istruzioni su diversi insiemi di dati, da qui appunto la definizione: *Multiple Instruction* (esecuzione di *stream* di istruzioni diverse su ogni processore) e *Multiple Data* (ogni processore lavora su un diverso insieme di dati). Come possiamo vedere in Figura 1.6,

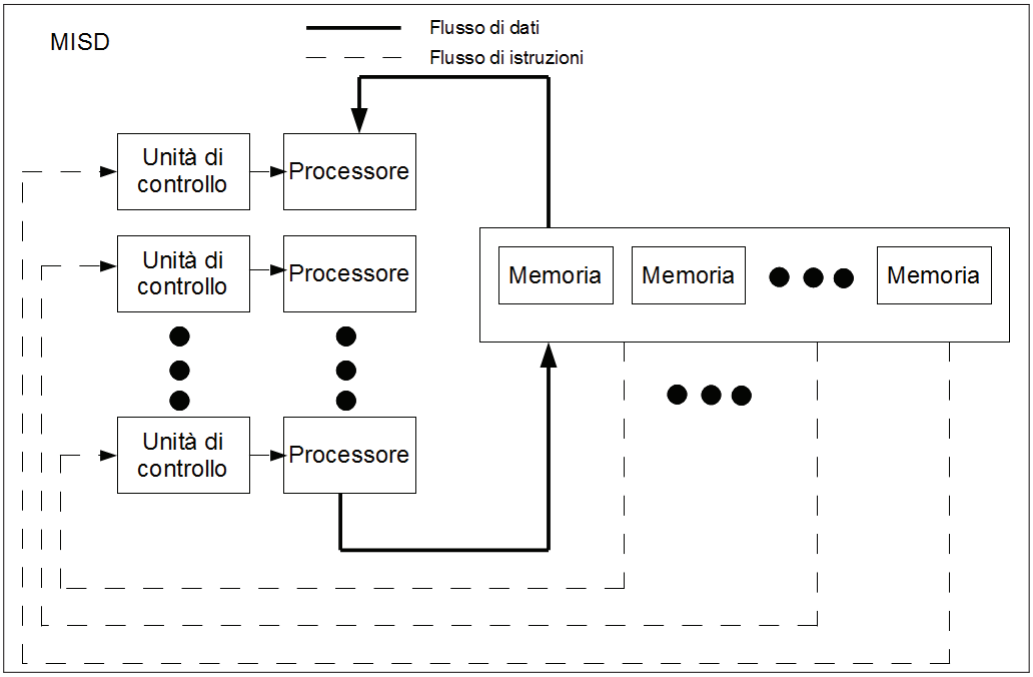


Figura 1.5. *Multiple Instruction Single Data* (MISD)

ogni processore possiede una propria unità di controllo e una propria memoria locale. L'esecuzione può essere sincrona o asincrona e può essere deterministica o non deterministica (ordine delle operazioni da svolgere prefissato).

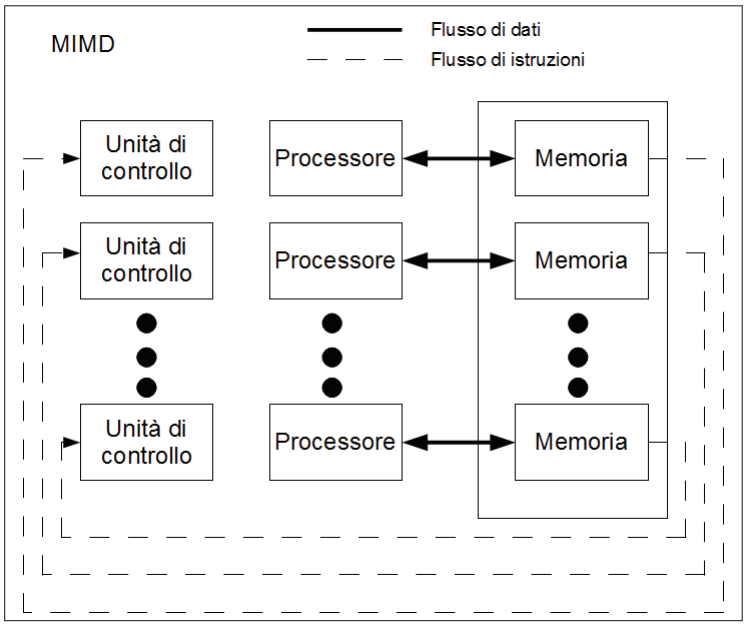


Figura 1.6. Multiple Instruction Multiple Data

## 1.3 Organizzazione della memoria

Un aspetto molto importante nella progettazione di un sistema per il calcolo parallelo riguarda l'organizzazione della memoria. Questa parte è fondamentale visto che l'obiettivo dell'aumento delle prestazioni non può prescindere dalla velocità con cui abbiamo accesso alla memoria per recuperare i dati da elaborare. È possibile suddividere le architetture viste nel Paragrafo 1.2 a seconda di come viene vista e gestita la memoria al loro interno.

Abbiamo principalmente tre tipologie con cui classificare le architetture:

- Memoria Condivisa (*Shared Memory*)
- Memoria Distribuita (*Distribuited Memory*)
- Memoria Ibrida Condivisa-Distribuita (*Hybrid Distribuited-Shared Memory*)

### Memoria Condivisa (*Shared Memory*)

Nel modello *Shared Memory* tutti i processori hanno accesso a tutta la memoria, la quale è vista come uno spazio di indirizzamento globale. Diversi processori operano in maniera indipendente l'uno dall'altro ma condividono la memoria. Le modifiche effettuate da un processore saranno quindi visibili a tutti gli altri processori collegati alla memoria, questa caratteristica di allineamento dei dati tra tutti i processori viene definita come *cache coherency*. Ogni processore, comunque, è in possesso di una piccola cache in cui possono essere memorizzate le informazioni prima di essere copiate nello spazio di memoria condiviso. Una volta che il valore viene copiato nella memoria condivisa deve essere mandato alle cache degli altri processori in modo che questi non si trovino a lavorare con valori obsoleti di quella variabile. Questa tipologia di organizzazione può essere facilmente adeguata a tutte le architetture viste nel paragrafo precedente. La struttura appena descritta è possibile osservarla in Figura 1.7.

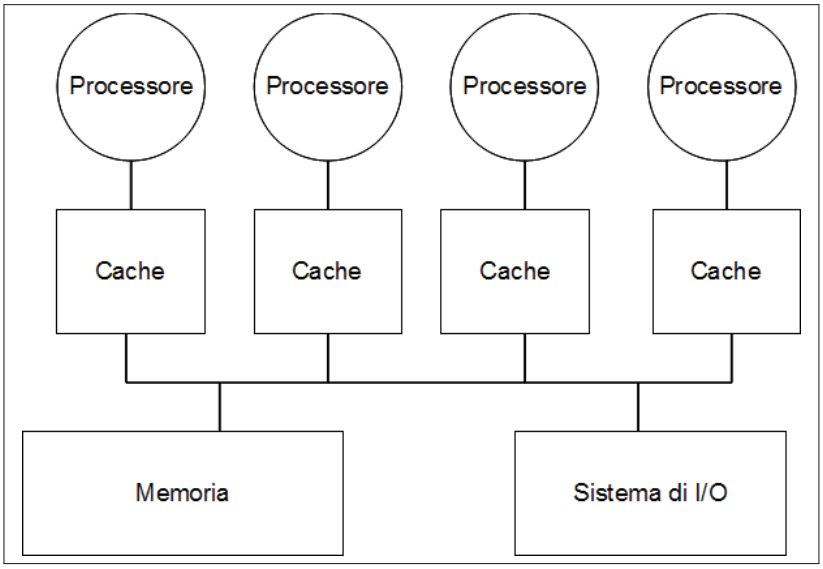


Figura 1.7. Sistema a memoria condivisa

L'accesso alla memoria può essere di due tipi:

- **Accesso alla memoria uniforme (UMA)**: la caratteristica di questo tipo di accesso è che il tempo di accesso è costante per ogni processore e per qualsiasi

zona di memoria. Questo tipo di accesso viene utilizzato su macchine SMP (*Simmetric Multiprocessor*)

- **Accesso alla memoria non uniforme (NUMA)**: in questo caso, i tempi di accesso alla memoria variano in base al processore. Una caratteristica di questo modello è che un processore può accedere direttamente alla memoria cache di un altro processore. La costruzione prevede l'utilizzo di più SMP, ed ogni SMP ha accesso diretto alla memoria di altri SMP. A questo punto però l'accesso alla memoria attraverso il bus sarà più lento.

Vediamo ora quali sono i vantaggi e gli svantaggi che l'utilizzo di questo modello porta con se.

**Vantaggi**:

- Lo spazio di indirizzamento globale rende intuitiva la programmazione parallela necessaria ad implementare l'algoritmo;
- La vicinanza della memoria ai processori garantisce uno scambio dati veloce e uniforme.

**Svantaggi**:

- Se aumento il numero di CPU aumento il relativo traffico tra queste ultime e la memoria;
- Se il sistema è *cache-coerent* aumenterà il costo per tenere aggiornata la cache;
- Il programmatore deve gestire la concorrenza per l'accesso alla memoria;
- Produrre calcolatori paralleli con questa tipologia diventa sempre più difficile e costoso all'aumentare del numero di processori;
- Nel caso vengano aggiunti successivamente nuovi processori l'applicazione necessita di essere rivista per poter gestire i nuovi processori disponibili.

La sincronizzazione dell'accesso alle strutture dati condivise è uno dei problemi maggiori nei sistemi a memoria condivisa. Sarà compito del programmatore assicurarsi che le operazioni da diversi processori sulla struttura dati condivisa lasci la

struttura in uno stato consistente. Il problema maggiore con i sistemi a memoria condivisa è che questi sistemi non sono scalabili ad un gran numero di processori. La maggior parte dei sistemi basati sui bus è limitato a 32 o meno processori a causa della dimensione del bus. Se il bus venisse sostituito da uno *switch crossbar* (sistema elettronico che collega multiple entrate a multiple uscite in modo matriciale), il sistema potrebbe scalare fino a 128 processori, sebbene il costo di questo incremento di processori aumenta incrementa in modo quadratico rispetto al numero di processori. (Dongarra *e altri*, 2003)

### Memoria Distribuita (*Distribuited Memory*)

In un sistema a memoria distribuita ogni processore ha una propria memoria locale che non può essere condivisa con un altro processore. I processori lavorano in modo indipendente l'uno dall'altro e le modifiche alla memoria, che avvengono in uno, non influenzano la memoria degli altri (non esiste quindi il concetto di *cache-coerency*). Nel caso ci debba essere uno scambio di informazioni tra processori dovrà essere il programmatore a definire quando e come deve avvenire questo scambio di dati. Questa tipologia di architettura di memoria è utilizzata nel caso in cui il calcolo parallelo venga implementato con più sistemi collegati tra loro attraverso una rete. La rete in questione può essere anche una semplice *ethernet*. È quindi particolarmente indicata nel caso di sistemi MIMD, ma potrà anche essere applicata ai casi SIMD facilmente.

**Vantaggi**:

- permette di avere un'alta scalabilità del sistema senza aumentare i costi in maniera esponenziale; Infatti, se prima l'aggiunta di nuovi processori richiedeva di dover intervenire sul programma per la gestione degli accessi alla memoria, in questo caso il problema non sussiste e l'inserimento di nuovi processori è vincolato solo al fatto di poter ampliare fisicamente la rete che collega i processori.

- Non esiste inoltre il problema del limite della memoria in quanto ogni processore aggiunto porta con se la propria memoria locale, aumentando di fatto la memoria totale del sistema. Questa gestione della memoria permette di non avere un carico di lavoro aggiuntivo per mantenere la *cache coerency*, visto che

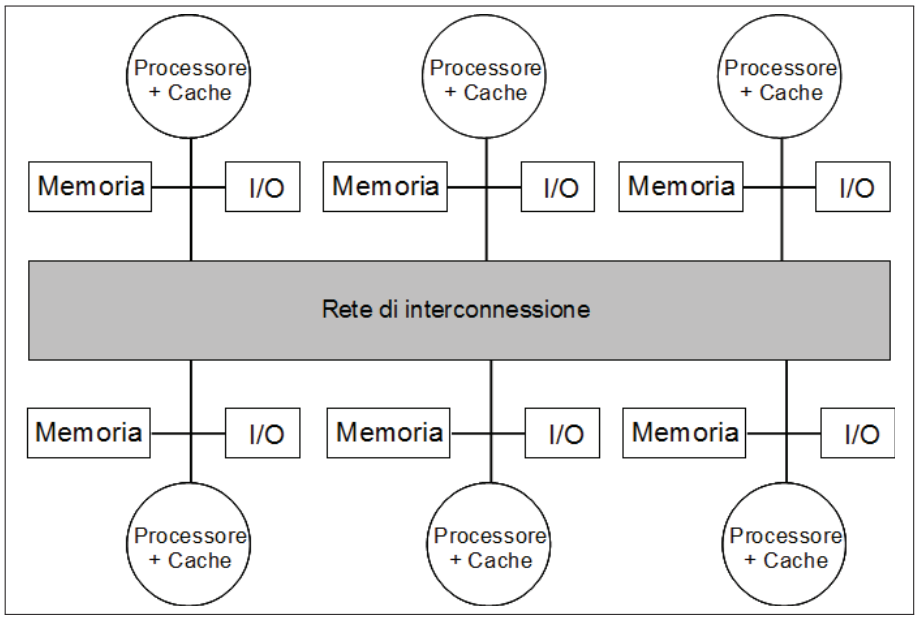


Figura 1.8. Sistema a memoria distribuita

ogni processore accede solo alla propria memoria locale (salvo casi particolari in cui è richiesto lo scambio di dati tra processori).

**Svantaggi**:

- Aumento della difficoltà nella gestione della comunicazioni tra processori, responsabilità che in questo sistema è completamente a carico del programmatore.

- L'accesso alla memoria è di tipo NUMA (Non uniform Memory Access) poiché il sistema può presentare al suo interno componenti molto diversi l'uno dall'altro.

- La comunicazione tra processori, per lo scambio di informazioni presenti nelle memorie locali, deve essere fatta utilizzando delle tecnologie per lo scambio di dati, come per esempio il *Message Passing*.

- L'utilizzo di queste tecnologie introduce dei rallentamenti nel sistema per la composizione e invio di questi pacchetti lungo la rete. A questo bisogna aggiungere che la suddivisione dei dati nelle memorie locali incide molto sulle prestazioni della macchina: è infatti fondamentale fare una suddivisione accurata in modo da limitare al minimo le comunicazioni tra le CPU.

### Memoria Ibrida Distribuita-Condivisa (*Hybrid Distribuited-Shared Memory*)

Si tratta dell'architettura di memoria che oggi prevale su tutte. I calcolatori paralleli moderni più grandi e veloci utilizzano questa tipologia. In Figura 1.9 vediamo come all'interno di questa architettura, sistemi SMP lavorano ciascuno con la propria memoria condivisa, e la connessione in rete viene utilizzata solo quando è necessario muovere dati da un sistema SMP a un altro. Le componenti a memoria condivisa sono quindi SMP con *cache-coerency*, mentre la componente distribuita della memoria è sulla rete che connette le diverse SMP. La coerenza della cache in questi sistemi è un problema complesso che di solito viene maneggiato utilizzando sofisticate unità di interfaccia alla rete. Come nei sistemi distribuiti ogni SMP può

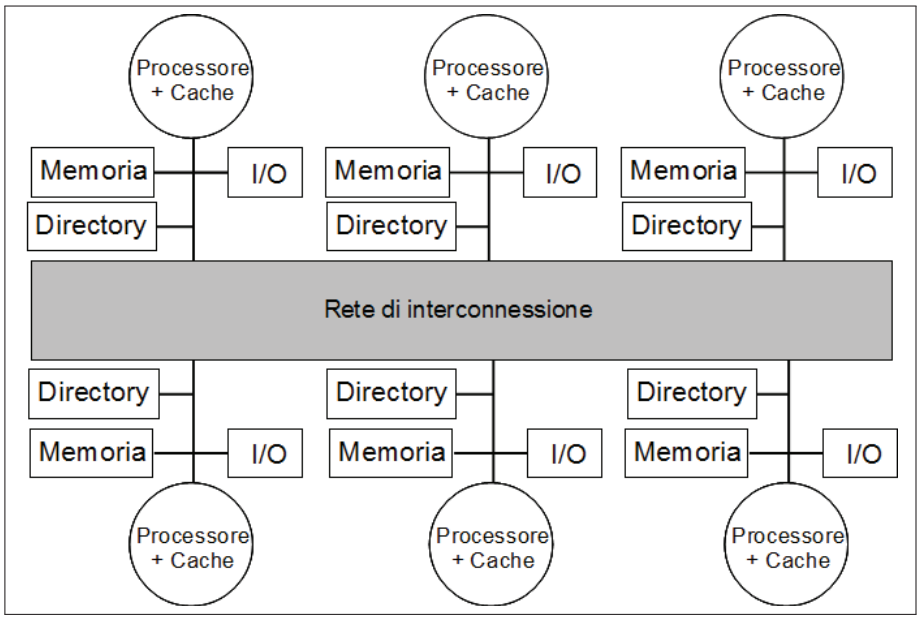


Figura 1.9. Sistema a memoria ibrida distribuita-condivisa

accedere solo alla propria memoria, mentre l'accesso alla memoria di un'altra SMP deve essere svolto attraverso la comunicazione sulla rete.

I **vantaggi** e gli **svantaggi** di questa architettura sono gli stessi che incontriamo nell'implementazione delle singole architetture a memoria condivisa e memoria distribuita. Per sistemi paralleli molto grandi, un'architettura ibrida, è una soluzione molto comune. Un cluster SMP appare come un sistema di memoria distribuita-condivisa nel quale ogni componente è un multi-processore simmetrico invece di un singolo processore. Questa conformazione permette un'alta efficienza con macchine multi-processore e permette al sistema di scalare centinaia e a volte migliaia di processori.

## 1.4 Progettazione di algoritmi paralleli

La progettazione di algoritmi che sfruttano il parallelismo si basa su una serie di operazioni che devono necessariamente essere svolte affinché il programma esegua correttamente il lavoro senza produrre risultati parziali o errati. Le macro-operazioni che devono essere svolte per una corretta parallelizzazione di un algoritmo sono:

- Scomposizione in task
- Assegnazione dei task
- Orchestrazione
- Mapping

Andiamo ora ad analizzare singolarmente le fasi e diamo una semplice definizione e descrizione.

#### Scomposizione in task

In questa prima fase viene preso il programma completo e viene suddiviso in task, ovvero in un insieme di istruzioni che potranno poi essere eseguite su processori diversi per attuare il parallelismo. Per fare questa suddivisione esistono due metodi:

- **Scomposizione *domain***: Vengono scomposti i dati del problema; l'applicazione è comune a tutti i processori che lavorano ognuno su una porzione di dati diversa. Questa metodologia viene utilizzata quando abbiamo una grande quantità di dati che devono essere elaborati.

- **Scomposizione *funzionale***: In questo caso è il problema che viene suddiviso in parti, ogni task eseguirà una particolare operazione e lo farà su tutti i dati disponibili.

### Assegnazione dei task

In questa fase viene specificato quale sarà il meccanismo con cui i vari task saranno distribuiti tra i vari processi. Questa fase è molto importante in quanto è qui che dovremmo andare a stabilire la distribuzione del carico di lavoro tra i vari processori. Bilanciare il carico è fondamentale in quanto bisogna fare in modo che tutti i processori lavorino con continuità, evitando che restino inattivi per troppo tempo. Per effettuare ciò, occorre tenere conto dell'eventuale eterogeneità del sistema cercando di assegnare più operazioni ai processori più performanti e meno a quelli più lenti. Infine, per una maggiore efficienza della parallelizzazione, è necessario limitare il più possibile le comunicazioni tra processori in quanto sono spesso fonte di rallentamenti e consumo di risorse.

### Orchestrazione

Questa fase riassume dentro di se tutte le varie fasi antecedenti all'esecuzione vera e propria del programma parallelo. In questa fase infatti vengono eseguite:

- organizzazione della distribuzione dei dati;

- organizzazione dell'ordine temporale dei task;

- organizzazione della sincronizzazione e comunicazione tra i processi;

- riduzione della comunicazione tra i processi:

- riduzione della serializzazione presso le risorse condivise;

**Mapping**

Si tratta dell'ultima fase prima dell'esecuzione. Consiste nell'associare i processi alle risorse fisiche quali: processori, memoria, ecc.

## 1.5 Modelli di programmazione parallela

Questi modelli non sono specifici e non fanno riferimento a particolari tipi di macchine o architetture di memoria, vengono perciò anche definiti *modelli multi-piattaforma*. Possono quindi essere implementati (in linea teorica) su macchine di qualsiasi tipo anche se vedremo che alcuni saranno particolarmente indicati per alcune tipologie e meno per altre.

Rispetto alle suddivisioni viste in precedenza nei Paragrafi 1.2 e 1.3, queste sono fatte ad un livello superiore e rappresentano il modo in cui il software va ad operare per eseguire il calcolo parallelo. Ogni modello quindi avrà il proprio modo per condividere le informazioni con gli altri processori, di accedere alla memoria, di suddividere il lavoro, ecc. Non esiste un modello migliore in termini assoluti rispetto agli altri, la soluzione migliore da applicare dipenderà molto dal problema che dovremmo andare ad affrontare e risolvere.

I modelli più utilizzati per la programmazione parallela sono:

- *Shared Memory*
- *Threads*
- *Message Passing*
- *Data Parallel*

### 1.5.1 *Shared Memory*

In questo modello i task condividono un'unica area di memoria nella quale scrivono e leggono in maniera asincrona. Esistono dei meccanismi che ci permettono di regolare l'accesso alla memoria condivisa, per esempio *lock* o *semafori*. Questo modello offre al programmatore il vantaggio che non deve esplicitare le comunicazioni

tra i task, anche se gestire dati di tipo locale risulta complicato per un programmatore non particolarmente esperto. Inoltre, utilizzando questo modello, risulta più facile e immediato sviluppare un programma parallelo partendo in modo incrementale da una versione seriale dell'algoritmo di interesse. Il modello *shared memory* presenta degli svantaggi nel momento in cui viene applicato ad un problema reale, infatti, nel caso in cui il problema che stiamo affrontando preveda una interconnessione particolarmente elevata tra i processi, questo modello risulta inadatto. È piuttosto complesso gestire la località dei dati e come conseguenza di ciò abbiamo una perdita di *performance* da parte del programma.

### 1.5.2 *Threads (Multi-threaded)*

In questo modello il processo può avere percorsi multipli di esecuzione, per esempio, viene eseguita inizialmente una parte sequenziale e successivamente vengono creati una serie di task che possono essere eseguiti parallelamente. Solitamente, questo tipo di modello, viene utilizzato su architetture a memoria condivisa, quindi sarà molto importante la gestione della sincronizzazione tra *thread*, poiché operando su memoria condivisa si vuole evitare che più *thread* aggiornino le stesse locazioni allo stesso tempo. Esistono diverse implementazioni di questo modello, che è stato anche oggetto di tentativi di standardizzazione che ha portato a due risultati principali:

- ***POSIX Threads (Pthreads)***: Implementazione basata su librerie in C, la gestione del parallelismo è completamente a carico del programmatore;

- **OpenMP**: Implementazione basata su direttive di compilazione, utilizzabile sia in linguaggio C che C++. Semplice da usare e permette una parallelizzazione incrementale, ovvero abbiamo una parallelizzazione a partire dall'algoritmo seriale.

Questo è il modello che abbiamo selezionato per il lavoro di questa tesi, in quanto ci permette di raggiungere gli obiettivi prefissati inizialmente utilizzando al meglio l'architettura hardware a nostra disposizione.

### 1.5.3 *Message Passing*

Il modello *message passing* viene solitamente applicato nel caso in cui ogni processore ha la propria memoria (sistemi a memoria distribuita). Più task possono risiedere sulla stessa macchina fisica oppure su un numero arbitrario di macchine. Il programmatore è responsabile di determinare parallelismo e scambio dei dati che avviene attraverso dei messaggi e per implementarlo è necessario richiedere una libreria di funzioni da richiamare all'interno del codice. Nel tempo sono state create numerose implementazioni del modello. Alcuni esempi sono disponibili già dagli anni 80 ma è solo da metà anni 90 però che si è cercato di standardizzare il modello, arrivando ad avere uno standard 'di fatto' chiamato **MPI** (*Message Passing Interface*). Il modello è chiaramente pensato per essere implementato su architetture a memoria distribuita ma, essendo i modelli di programmazione parallela multi-piattaforma possono essere anche usati con macchine a memoria condivisa.

### 1.5.4 *Data Parallel*

In questo modello abbiamo più task che operano sulla stessa struttura di dati, ma ogni task opera su una porzione differente di dati. Su un'architettura a memoria condivisa, tutti i task possono avere accesso ai dati attraverso la memoria condivisa mentre su architetture con memoria distribuita, la struttura dati è suddivisa e risiede nella memoria locale di ogni task.

Nell'implementare questo modello, il programmatore deve provvedere a sviluppare un programma che specifichi la distribuzione e l'allineamento dei dati.

## 1.6 Valutazione di un algoritmo parallelo

La nascita e lo sviluppo del calcolo parallelo ha portato con se la necessità di avere degli strumenti per poter valutare i limiti degli algoritmi e misurarne le performance per poter decidere se il loro utilizzo sia conveniente o meno. L'obiettivo fondamentale del calcolo parallelo è quello di risolvere problemi di grandi dimensioni in tempi ragionevolmente brevi. I fattori che concorrono al raggiungimento di questo obiettivo sono molteplici, per esempio, il tipo di hardware utilizzato, il grado di parallelizzabilità del problema, e di conseguenza quale modello parallelo viene

adottato. Per agevolare l'analisi delle prestazioni fornite da un algoritmo parallelo sono stati introdotti dei concetti basilari, che da una parte confrontano l'algoritmo ottenuto con quello originale, verificandone (e quantificandone) l'effettivo aumento di prestazioni ottenuto, dall'altra analizzano il comportamento dell'algoritmo al variare del numero di processi e/o *thread* impiegati.

Vengono utilizzati una serie di indici per analizzare questi valori, ognuno dei quali valuta delle componenti diverse della parallelizzazione. Alcuni indici utilizzati in questo ambito sono: *Speedup*, *Efficienza* e *Scalabilità*. Una volta definiti questi concetti di misure di prestazioni sono state elaborate una serie di leggi che tutt'oggi restano valide anche nelle nuove implementazioni, per esempio la *legge di Ahmdal* viene utilizzata per definire i limiti del calcolo parallelo, oppure la *legge di Gustafson*, che suggerisce quando una parallelizzazione del codice può risultare efficiente.

### 1.6.1 Indici di prestazione

#### Speedup

Uno degli strumenti di più facile utilizzo per confrontare un algoritmo sequenziale con il suo corrispettivo parallelo viene chiamato *speedup* (Xiao, 2010).

Viene definito come:

$$S_p = \frac{T_1}{T_p} \tag{1.1}$$

Dove:

$p$ è il numero di processori;

$T_1$ è il tempo di esecuzione della versione sequenziale dell'algoritmo;

$T_p$ è il tempo di esecuzione della versione parallela dell'algoritmo con $p$ processori.

In base a quanto si riesce a guadagnare in termini di tempo di esecuzione esistono diversi tipi di *speedup*. Lo ***speedup* ideale**, detto anche ***speedup* lineare**, si ottiene quando $S_p = p$. Nel caso in cui il nostro algoritmo abbia una *speedup* lineare significa che, per esempio, raddoppiando il numero di processori si raddoppia anche la velocità di esecuzione. Questa situazione corrisponde al caso ideale, ed un sistema in possesso di queste caratteristiche avrebbe, in linea teorica, la massima scalabilità possibile; cioè la massima versatilità al crescere o decrescere in funzione delle necessità o disponibilità del problema. Lo *speedup*, viene detto **assoluto** quando $T_1$ è il tempo di

esecuzione del miglior algoritmo sequenziale. Abbiamo *speedup* **relativo** quando $T_1$ è il tempo di esecuzione del corrispondente algoritmo parallelo su un unico processore.

Formalizzando la definizione ed applicandola a dei casi reali abbiamo quindi tre possibili equazioni che legano lo *speedup* con il numero di processori:

- $S_p = p$ *speedup* lineare (caso ideale);
- $S_p < p$ *speedup* di un caso reale;
- $S_p > p$ *speedup* superlineare (effetti di *cache*).

### Efficienza

L'efficienza è un parametro delle prestazioni e viene definito nel modo seguente:

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} \tag{1.2}$$

Questo parametro, compreso tra zero e uno, ci permette di stimare quanto i processori vengono sfruttati per la risoluzione del problema rispetto all'esecuzione di tutte le altre attività a lui assegnate, come per esempio scambio di messaggi di comunicazione e sincronizzazioni. Gli algoritmi che hanno *speedup* lineare, o processi che utilizzano un solo processore hanno efficienza pari a 1. In tutti gli altri casi, l'efficienza sarà pari a $\frac{1}{log(p)}$ e, come possiamo osservare dalla formula, all'aumentare del numero di processori questo valore tende a 0.

Generalmente quindi l'effettivo utilizzo della macchina parallela corrisponde a:

- $E_p = 1$ Caso ideale
- $E_p < 1$ Caso reale
- $E_p \ll 1$ Problemi nella parallelizzazione

### Scalabilità

La scalabilità è definita come la capacità di essere efficiente su una macchina parallela. Identifica la capacità di un sistema parallelo di incrementare la potenza di calcolo (velocità di esecuzione) in maniera proporzionale all'aumentare del numero di processori. Se si aumentasse la dimensione del problema e allo stesso tempo

aumentasse il numero di processori a disposizione, nel caso in cui un sistema sia scalabile, non avrò perdita di efficienza. Il sistema scalabile, a seconda degli incrementi dei diversi fattori, potrà mantenere la stessa efficienza oppure migliorarla.

### 1.6.2 Legge di Ahmdal

Dopo aver definito tutta una serie di indici per misurare le performance degli algoritmi parallelizzati si è sentita la necessità di capire quale fossero i limiti che la parallelizzazione ha nella realtà. Una delle prime leggi che andremo ad analizzare sarà la *legge di Ahmdal*, la quale, permette di trovare lo *speedup* massimo di un sistema in cui è possibile parallelizzare solo una parte del programma in esecuzione e la dimensione del problema non aumenta. Questo, è un caso molto frequente nella realtà, poiché il numero di algoritmi che sono totalmente parallelizzabili non sono molti. In questi casi dunque, non essendo possibile parallelizzare per intero l'algoritmo, il tempo di esecuzione sarà vincolato alla parte sequenziale dell'algoritmo, costituendo di fatto un limite per lo *speedup* ottenibile dalla parallelizzazione. Il tempo di esecuzione minimo sarà dunque il tempo di esecuzione della parte sequenziale, soglia al di sotto della quale non riusciamo a scendere. Da ciò deduciamo che lo *speedup* massimo di un programma non totalmente parallelizzato è dato da:

$$S = \frac{1}{1 - P} \tag{1.3}$$

Dove $1 - P$ è la parte di programma non parallelizzato.

Aumentando la precisione della definizione appena fatta possiamo indicare lo *speedup* complessivo utilizzando la seguente formula:

$$SpeedupComplessivo = \frac{1}{(1 - P) + \frac{P}{S}} \tag{1.4}$$

Dove:

1: tempo di esecuzione algoritmo sequenziale;

$(1 - P)$: porzione di tempo di esecuzione non parallelizzabile;

$\frac{P}{S}$: porzione di tempo di esecuzione parallelizzabile divisa per il suo *speedup* S.

Il grafico in Figura 1.10 mostra l'andamento dello *speedup* al variare del numero di processori e dalla porzione di programma parallelizzabile. La cosa che subito si

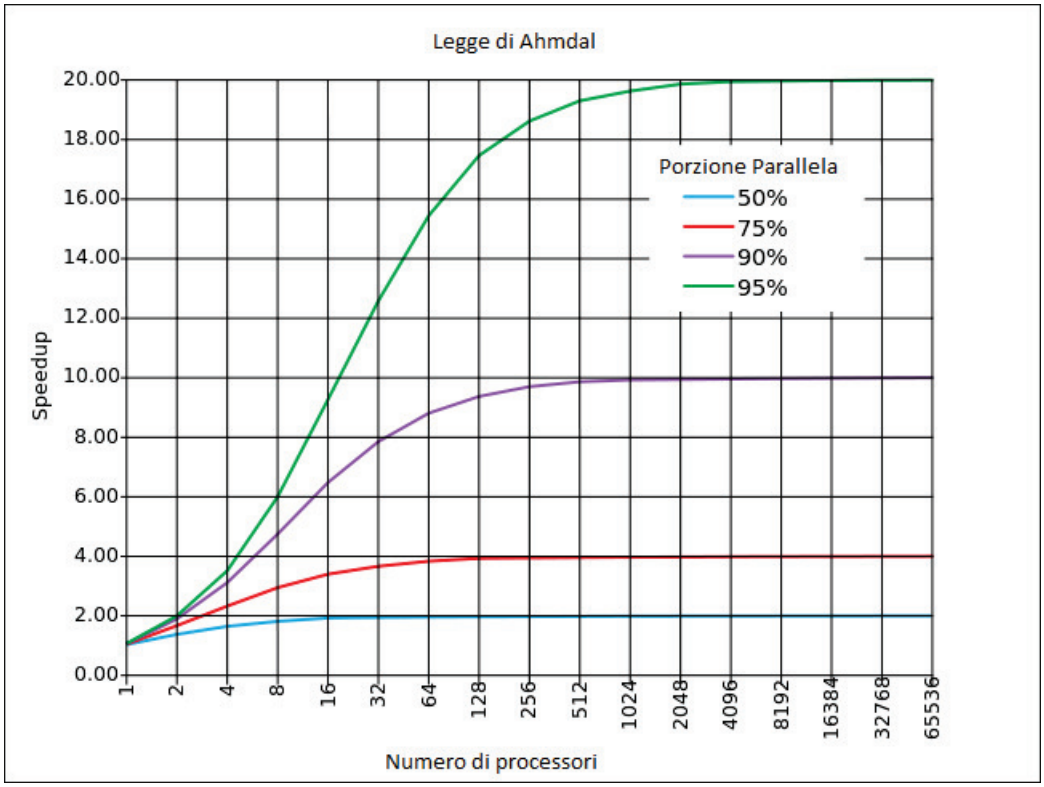


Figura 1.10. Legge di *Amdahl* al variare del livello di parallelizzazione del programma e del numero di processori

può notare è che, ad un fissato numero di processori, lo *speedup* cambia di molto a seconda della percentuale di parallelizzazione del programma. Per esempio, con 16 processori, abbiamo che al 50% di operazioni parallelizzate lo *speedup* è circa a 2, al 75% vale circa 3, al 90% siamo già ad uno *speedup* di 6 infine con una parallelizzazione del 95% delle operazioni arriviamo ad uno *speedup* di 9. Ulteriore considerazione da fare è la presenza di limiti nell'incremento dello *speedup* all'aumentare dei processori. Vediamo che da un certo punto in poi tutte le curve si assestano attorno al limite visto in precedenza di $\frac{1}{(1-P)}$, rendendo di fatto inutile l'inserimento di un gran numero di processori se il programma non può essere parallelizzato di più. Di fatto questa legge ci suggerisce di aumentare il più possibile la parte parallela del programma poiché il limite di processori non ci permette di ottenere dei grossi miglioramenti di performance oltre un certo punto.

### 1.6.3 Scalabilità e legge di Gustafson

Come abbiamo visto, l'analisi del programma parallelo al variare del numero di processori impiegati è fondamentale per capire il guadagno che è possibile trarre dalla parallelizzazione. Questo tipo di analisi va sotto il nome di analisi di scalabilità. Un algoritmo può essere definito scalabile quando un incremento dei processori dà luogo a un aumento proporzionale dell'efficienza dell'algoritmo, ovvero del suo speedup. Dal paragrafo sullo speedup, abbiamo visto che aumentare il numero di processori oltre un certo limite non apporta più benefici significativi, arrivando a una sorta di limite dello speedup. Se però aumentiamo la dimensione del problema osserviamo un incremento dello speedup massimo ottenibile. Se andiamo a variare quindi, sia numero di processori che dimensione del problema, possiamo mantenere costante l'efficienza dell'algoritmo permettendo di risolvere problemi di dimensioni maggiori semplicemente aumentando il numero di processori.

Dal comportamento descritto qui sopra, è stata formulata una legge che va sotto il nome di *legge di Gustafson*.

Questa legge si basa su due considerazioni:

- All'aumentare della dimensione del problema la parte sequenziale rimane costante;

- All'aumentare del numero di processori, deve restare costante il lavoro eseguito da ognuno di essi.

Se $T$ è il tempo totale di elaborazione, $T_s$ il tempo impiegato per elaborare la parte sequenziale e $T_p$ quello per la parte parallela, si ha $T = T_s + T_p$ e lo speedup su $n$ processori è dato da:

$$S_p(N) = \frac{T_s + NT_p}{T_s + T_p} = s + Np = N + (1 - N)s \tag{1.5}$$

Dove:
$s = \frac{T_s}{T_s+T_p}$;
$p = \frac{T_p}{T_s+T_p}$;
$s + p = 1$

A prima vista, l'andamento previsto dalla *legge di Gustafson* sembrerebbe contraddittorio rispetto al modello elaborato da *Amdahl*. La differenza sostanziale che incontriamo tra le due leggi è che la *legge di Gustafson* assume che la frazione di tempo impiegata per eseguire la parte sequenziale non sia più costante al variare della dimensione del problema, ma diminuisca all'aumentare di questa. In sostanza quando andiamo ad aumentare la dimensione del problema ciò che veniva fatto in sequenziale resta tale mentre aumenta la parte parallela, la differenza però è che da un punto di vista complessivo se la parte sequenziale aumenta in modo lineare la capacità di calcolo parallela può aumentare in modo esponenziale ($n^p$) permettendo di fatto una riduzione complessiva dei tempi di esecuzione.

# Capitolo 2

# Data mining e calcolo parallelo

## 2.1 Perché utilizzare il data mining parallelo

Con il termine *data mining* vengono indicate una serie di attività di elaborazione in forma grafica o numerica di grandi raccolte o flussi continui di dati con lo scopo di estrarre informazione utile a chi detiene i dati stessi (Azzalini e Scarpa, 2004).

Questa disciplina, nata recentemente con lo sviluppo di nuove tecnologie, si colloca come punto d'intersezione tra diverse aree scientifiche, in particolare: la statistica, l'intelligenza artificiale e la gestione dei database.

La nascita del *data mining* è legata in modo indissolubile allo sviluppo tecnologico degli ultimi vent'anni. Queste innovazioni tecnologiche hanno di fatto rivoluzionato il modo di lavorare in tantissimi ambiti, da quello scientifico passando per quello aziendale e tecnologico, fino ad arrivare agli aspetti della vita quotidiana. In qualsiasi settore è diventato semplice ed economico disporre di quantità rilevanti di informazioni su un fenomeno di interesse. Questo fatto è legato a due fattori: uno è lo sviluppo dei metodi automatici di rilevazione dei dati, l'altro è il potenziamento dei sistemi di memorizzazione elettronica dell'informazione e l'associato abbattimento dei costi connessi (Azzalini e Scarpa, 2004).

Avere una grossa quantità di dati da cui estrarre informazione non rende le cose più facili, anzi, trovare informazione utile in mezzo ad una mole così grande di dati rischia di essere estremamente complicato e di richiedere tempi molto lunghi. I

principali problemi che emergono nel *data mining* riguardano proprio questi aspetti: la numerosità campionaria, quante unità statistiche considero nell'analisi, e la dimensionalità dei dati, ovvero le caratteristiche rilevate su ciascuna unità.

L'onore computazionale di questa tipologia di problemi sarà quindi molto elevata. Lo sviluppo recente delle tecnologie sta indirizzando verso l'utilizzo del *calcolo parallelo* per l'esecuzione di questi algoritmi.

Si tratta in sostanza di utilizzare come base gli algoritmi di *data mining* costruiti per una esecuzione sequenziale, modificarli ed infine utilizzarli per sfruttare le nuove risorse a disposizione, come computer multi-processore o cluster di computer, per ridurre i tempi di esecuzione degli algoritmi con la certezza di avere allo stesso tempo risultati validi.

In seguito illustreremo quali possono essere le strategie per parallelizzare un algoritmo (Paragrafo 2.2) e la parallelizzazione attraverso il *multithreading* (Paragrafo 2.3).

## 2.2 Strategie per il *data mining* parallelo

Nel *data mining* la parallelizzazione può essere utilizzata per due motivi: il primo è che analizzare la variazione dei dati appare algoritmicamente complesso e richiede una potenza di calcolo molto elevata; il secondo è che i dataset sono grandi e crescono molto velocemente. I computer che lavorano in parallelo possono maneggiare in modo piuttosto efficace questa grande mole di dati, sebbene alcuni problemi di *data mining* stiano già raggiungendo i limiti dell'hardware attuale. Gli algoritmi di *data mining* possono essere parallelizzati in molti modi diversi. Poiché disegnare e implementare programmi paralleli è costoso, non è pratico testare tutte le possibili implementazioni e confrontarle per scegliere poi la migliore. Fortunatamente, stanno rapidamente maturando delle misure per calcolare la complessità pratica per la programmazione parallela, rendendo quindi non più necessario il confronto tra tutti gli algoritmi. Vedremo ora come utilizzare queste misure per la valutazione della parallelizzazione degli algoritmi di *data mining*.

Una tipica applicazione di *data mining* inizia con un insieme di dati che descrivono le interazioni tra più variabili esplicative e una variabile risposta. Il requisito

essenziale è che gli individui mostrino delle variazioni e che la variabile risposta misuri queste variazioni. Ciò che vogliamo è capire quali siano le relazioni che creano questa variazione tra gli individui. Spesso ciò si traduce nella suddivisione in gruppi degli individui, gruppi in cui gli individui sono quanto più omogenei al loro interno ed eterogenei tra gruppi diversi.

L'output di un algoritmo di data mining può assumere le seguenti forme:

- *Un insieme di* ***concetti*** *sulle interazioni* (ovvero frasi che descrivono queste interazioni), per esempio, 'l'80% dei consumatori compra pasta e pane nella stessa visita al punto vendita'. Questi concetti sono molto utili vista la facilità di utilizzo anche per utenti con una preparazione non prettamente statistica.

- *Un insieme di* ***parametri*** *del modello*, questa tipologia viene spesso utilizzata per costruire classificatori, per esempio, per suddividere i consumatori in diverse categorie.

Nel resto del paragrafo ci riferiremo ai risultati come se fossero tutti dei concetti, anche se ovviamente la deduzione del concetto sarà fatta sulla base dei parametri che l'algoritmo seleziona come rilevanti per il problema che gli viene sottoposto. L'utilizzo quindi di concetto o insieme di variabili sarà intercambiabile e la costruzione di concetti equivale alla selezione di variabili dall'insieme complessivo delle variabili a nostra disposizione.

Il *parallel data mining* richiede di dividere il lavoro in modo che tutti i processori siano utili all'avanzamento verso la soluzione il più velocemente possibile.

Abbiamo visto nel Paragrafo 1.4 quali sono le operazioni che vengono effettuate per la parallelizzazione e cosa deve essere tenuto in considerazione e analizzato. Partendo da quella suddivisione è possibile avere un'ulteriore classificazione raggruppandole in macro operazioni che potremmo definire hardware dipendenti.

Queste tre macro-componenti principali possono essere definite come:

- Calcolo

- Accesso al dataset

- Comunicazione tra processori

Nei computer paralleli moderni, l'accesso al data set è probabilmente l'operazione più costosa da eseguire, seguita dalla comunicazione, mentre il calcolo è l'operazione relativamente più economica. Il costo del calcolo decresce più velocemente rispetto agli altri due costi, i quali sono responsabili della maggior parte del carico di lavoro necessario all'algoritmo. I tre componenti sono fortemente collegati tra loro. Per esempio, suddividere il calcolo per renderlo più veloce crea una maggiore comunicazione e spesso richiede di dover accedere più volte al dataset. Trovare la soluzione più efficace richiede un attento bilanciamento delle tre operazioni e dei loro costi. Negli algoritmi paralleli di *data mining* spesso ogni processore ha una serie di concetti/variabili che deve calcolare e questi sono validi localmente. Attraverso la comunicazione poi viene deciso cosa deve essere reso globale e a disposizione di tutti e cosa non è necessario memorizzare per la corretta esecuzione dell'algoritmo.

Distinguiamo tre strategie di base per parallelizzare algoritmi di *data mining*:

- **Ricerca indipendente**: ogni processore accede a tutto il dataset, ma ognuno in uno spazio di ricerca diverso, ovvero inizia da una posizione iniziale casuale.

- **Approccio parallelizzato**: ogni processore costringe se stesso a generare un particolare sottoinsieme dei possibili concetti. Esistono due varianti:

    1. Ogni processore genera concetti completi, ma con restrizioni sulle variabili valutate nella stessa posizione.

    2. Ogni processore genera concetti parziali, concetti coinvolgono alcuni sottoinsiemi di variabili, ma le variabili possono assumere qualsiasi valore. Validare questi concetti richiede di esaminare un sottoinsieme delle colonne.

- **Approccio replicato**: ogni processore lavora su una partizione del dataset (per righe) ed esegue l'algoritmo sequenziale. Poiché ogni processore vede solo parte delle informazioni, costruiscono concetti che sono validi localmente, ma che potrebbero essere validi anche a livello globale. Questi concetti vengono detti concetti approssimati. I processori si scambiano i concetti approssimati per cercare la loro correttezza globale, e quello che fanno in questo modo è conoscere le parti di dati che non possono vedere.

Oltre a queste tre strategie ne esistono altre, ma possiamo considerare queste come soluzioni principali per la creazione di algoritmi paralleli. La strategia di *Ricerca Indipendente* è molto buona quando si desidera ottenere in output un'unica soluzione considerata ottima, e lavora bene in caso di problemi di minimizzazione. L'*approccio parallelizzato* invece, cerca di ridurre sia la quantità di memoria utilizzata da ogni processore, per memorizzare concetti e variabili, sia la frazione di dati alla quale ogni processore deve accedere. Tuttavia essi tendono a non avere successo in entrambi i casi, poiché il numero di concetti potenziali è grande e molto spesso i concetti grandi sono generati meglio da concetti più piccoli riconosciuti come validi. L'*approccio replicato* non è particolarmente nuovo, ma è spesso il metodo migliore per incrementare le *performance* in un applicazione di *data mining*.

Dispone di due importanti vantaggi:

1. Richiede la suddivisione del data set e distribuisce i costi di accesso a tutti i processori;

2. I dati che devono essere scambiati tra le varie fasi, sono spesso molto più piccoli dei concetti stessi, quindi, come conseguenza, abbiamo una comunicazione meno costosa.

Questa tecnica può a volte generare concetti validi a livello locale e non estendibili a tutto il dataset, ma, poiché i concetti sono validi con il sottoinsieme dei dati sui quali vengono calcolati, questo non accade spesso. I risultati di un algoritmo di *data mining* possono a volte essere molto grandi rispetto al dataset in input. Ne consegue che l'insieme dei concetti può essere estremamente grande anche ad uno stadio intermedio dell'algoritmo. Quando questo accade l'*approccio replicato* non funziona molto bene, perché ogni processore tende ad avere in memoria tutti i concetti che sono stati generati (Skillicorn, 1999).

## 2.3 Parallelizzazione attraverso il *multithreading*

Per multithreading si intende un metodo con il quale vengono elaborati processi multipli e separati (chiamati *thread*) all'interno della stessa applicazione. Ciascuna elaborazione contiene quindi singole parti di codice (della stessa applicazione) in

esecuzione in modo indipendente (*thread*). Il multithreading può avvenire su singolo processore o su processori diversi in contemporanea. I singoli *thread* devono ovviamente condividere le risorse di sistema, e questo coinvolge strettamente il sistema operativo che, a seconda dell'architettura, gestisce in modo diverso il *multithreading* (Fonte: www.mucia.org/vocabolario/M/multithreading.htm).

Questa tecnica si distingue da quella alla base dei sistemi multiprocessore per il fatto che i singoli *thread* condividono lo stesso spazio d'indirizzamento, la stessa *cache* e lo stesso *translation lookaside buffer* (si tratta di un buffer utilizzato per velocizzare la traduzione degli indirizzi virtuali). Il *multithreading* migliora le prestazioni dei programmi solamente quando questi sono stati sviluppati suddividendo il carico di lavoro su più thread che possono essere eseguiti in parallelo. I sistemi multiprocessore sono dotati di più unità di calcolo indipendenti, un sistema multithread invece è dotato di una singola unità di calcolo che si cerca di utilizzare al meglio eseguendo più *thread* nella stessa unità di calcolo. Le tecniche sono complementari, a volte i sistemi multiprocessore implementano anche il multithreading per migliorare le prestazioni complessive del sistema (Fonte: http://it.wikipedia.org/wiki/Multithreading).

La parallelizzazione prevede vi siano delle fasi di calcolo alternate a fasi di comunicazione per sincronizzare gli eventi e scambiare dati.

La **granularità** è definita come il rapporto tra il tempo di calcolo e il tempo di comunicazione. Nel nostro caso specifico la comunicazione è intesa come comunicazione tra *thread*.

Se la granularità è sottile (*fine grain*), vuol dire che pochi calcoli sono svolti tra eventi di comunicazione, questo può portare ad un sovraccarico di comunicazione tra *thread*. Una granularità sottile facilita il bilanciamento del carico (*load balancing*)

Viceversa, se la granularità è grossolana (*coarse grain*) vuol dire che le fasi di calcolo sono prevalenti a quelle di comunicazione. Utilizzando questo metodo può essere più difficile eseguire il *load balancing*.

A seconda della granularità del codice esistono tre tipologie principali di *multithreading*.

- *Coarse-Grained Multithreading*: prevede che il processore esegua un singolo *thread* fino a quando questo non viene bloccato da un evento che normalmente ha una elevata latenza, ovvero che interrompa l'esecuzione e lasci trascorrere

dei cicli di *clock*. In questo caso il processore provvede a eseguire un altro *thread* che era pronto per l'esecuzione senza dover rimanere in attesa.

- *Fine-Grained Multithreading*: organizza più *thread* e ne esegue in contemporanea le istruzioni al fine di occupare al meglio le unità d'elaborazione. Con questo metodo viene garantita l'alternanza di tutti i *thread*, permettendo di ottenere un rendimento complessivo più alto,aumentando però la comunicazione per mantenere aggiornate le diverse parti.

- *Simultaneous Multithreading*: In questo *multithreading* si è in grado di gestire una parallelizzazione sia a livello di *thread* (grana grossa) sia a livello di istruzioni (grana fine). I moderni processori hanno più unità di calcolo che vengono utilizzate eseguendo le istruzioni dei singoli *thread* in parallelo. Gli attuali processori sono in grado di eseguire solamente poche istruzioni in parallelo di un singolo *thread* per via del ridotto parallelismo a livello di istruzioni che normalmente i *thread* possiedono. Quindi spesso alcune unità di elaborazione rimangono inutilizzate. Per evitare questo si eseguono più *thread* in contemporanea e si utilizzano le istruzioni dei singoli *thread* per mantenere le unità di elaborazione sempre operative.

# Capitolo 3

# Software

## 3.1 Cos'è R, vantaggi e svantaggi

R è un ambiente di sviluppo per l'analisi statistica dei dati. È un software libero, ha licenza GNU-GPL ed è utilizzabile su diversi sistemi operativi. R è stato sviluppato partendo da un altro linguaggio statistico simile (S), con il quale presenta una buona compatibilità di codice. Non si tratta di un pacchetto statistico vero e proprio, si tratta in realtà di un ambiente di programmazione, la sua popolarità in statistica è dovuta alla presenza di molti metodi statistici implementati dalla comunità di sviluppatori che segue il progetto. Questi metodi permettono di risolvere larga parte delle operazioni statistiche. Esiste quindi un programma base, contenente una serie di metodi standard (raggruppati in pacchetti), il quale verrà poi ampliato con l'aggiunta di pacchetti per includere altri metodi che possono risultare utili all'utente. Questo linguaggio di programmazione presenta una serie di vantaggi nel momento in cui viene utilizzato; per esempio oltre ad essere libero, *opensource* e multi-piattaforma, R presenta una console interattiva per le attività di esplorazione dei dati e permette di eseguire un numero elevatissimo di differenti calcoli statistici, grazie anche ai pacchetti aggiuntivi. R è diventato per tutte queste motivazioni uno dei software preferiti nell'era delle grandi basi di dati, ma è proprio a questo punto che si sono evidenziati i limiti di questo linguaggio di programmazione. Innanzitutto si tratta di un linguaggio interpretato, questo è un grosso limite nel momento in cui si devono svolgere operazioni con un alto costo computazionale, in quanto

l'esecuzione di un linguaggio interpretato è più lenta rispetto ad un linguaggio compilato. Altri limiti sono dovuti al fatto che R non era stato pensato per le grandi basi di dati quando fu progettato nel 1995. All'epoca lo spazio su disco era molto costoso e la RAM lo era ancora di più, inoltre internet era ancora ad uno stato poco più che embrionale. Termini come analisi di dati di grandi dimensioni o calcolo ad alte prestazioni erano piuttosto rari al di fuori di università e laboratori di ricerca. L'evoluzione dell'hardware in questi anni ha portato ad una forte diminuzione dei costi per la memoria fisica, la RAM e anche per la potenza di calcolo (processori). L'interesse verso la raccolta e l'analisi dei dati è cresciuta di pari passo, se non a velocità maggiore, rispetto all'hardware, aumentando sempre più la richiesta di risorse. Questa crescente richiesta di analisi dei dati ha portato in primo piano due dei limiti di R:

1. è *single-threaded*;

2. utilizzo della memoria (*memory bound*).

*Single-threaded*, nel senso che il linguaggio non prevede costrutti espliciti per il parallelismo come per esempio i *thread*. Non può quindi avere dei vantaggi dall'installazione su macchine con più CPU in modo automatico, ma richiederà una serie di operazioni per poter eseguire il codice in parallelo. Con il termine *memory bound*, si intende il fatto che con R tutto il dataset deve essere salvato nella memoria per poter essere utilizzato dalla memoria, limite piuttosto vincolante nel caso di grandissime basi di dati. Nel corso degli anni si sono cercate soluzioni a questi problemi, le proposte sono state varie (dalla matematica delle matrici, fino ad arrivare all'utilizzo di database relazionali per recuperare i dati in parti più piccole). La soluzione che andremo ad analizzare sarà il coinvolgimento del parallelismo, distribuire il lavoro su più CPU supera uno dei più grandi limiti di R (essere *single-threaded*).

## 3.2 Parallelizzazione in R

Nel corso del tempo sono stati implementati in R diversi strumenti per effettuare il calcolo parallelo. Molti utenti nuovi al calcolo parallelo si ritrovarono a scrivere porzioni di codice in parallelo con grandi aspettative sulle performance, salvo poi

andare a constatare che l'algoritmo sequenziale risultava più veloce rispetto a quello parallelo. Questo problema è particolarmente presente in R a causa delle caratteristiche di implementazione proprie di R. Vediamo quali sono quindi i problemi che causano queste ridotte performance degli algoritmi e come possono essere risolti grazie a una profonda conoscenza dell'hardware e del software utilizzati.

Nel paragrafo 1.6.1, abbiamo visto che una misura per verificare le performance del nostro algoritmo è lo *speedup*. Prendiamo, per esempio, una matrice di $1000 \times 1000$, e supponiamo di eseguire delle operazione su di essa prima con un algoritmo sequenziale, poi con un algoritmo parallelo su una macchina con 4 processori e infine su una macchina con 12 processori. Dal punto di vista teorico mi aspetterei che lo *speedup* sia lineare, e incrementi di 4 e 12 volte rispetto all'algoritmo sequenziale, ma una volta eseguito vediamo che lo *speedup* è notevolmente inferiore a quello teorico che ci aspetteremmo. Questa differenza, tra valori teorici e valori reali, è dovuta al fatto che si deve anche tenere conto di tutte quelle operazioni che devono essere effettuate per il controllo e la gestione delle varie componenti del parallelismo (comunicazione tra processori, accesso alla memoria, ecc.).

## 3.3 Ottimizzazione delle prestazioni

In informatica un problema comune è quello del compromesso nelle prestazioni tra tempo e spazio. Al fine di avere una rapida esecuzione del programma potrebbe essere necessario utilizzare più spazio in memoria. D'altra parte al fine di risparmiare spazio in memoria potrebbe essere necessario accontentarsi di un codice più lento. Nel linguaggio R questo compromesso è di particolare interesse per le seguenti motivazioni:

1. R è un linguaggio interpretato. Molti dei comandi sono scritti in C e quindi vengono eseguiti in codice macchina in modo molto rapido. Altri comandi invece, sono propri del codice R, e quindi devono essere interpretati, rendendo l'esecuzione del codice più lento di quanto ci si aspetterebbe.

2. Tutti gli oggetti della sessione di R sono immagazzinati nella memoria. Più precisamente vengono memorizzati nello spazio degli indirizzi di memoria di R. R pone un limite alla memoria utilizzabile, e questo limite è di $2^{31} - 1$ byte

per ogni oggetto, anche su macchine a 64 bit o in caso di una RAM molto grande.

### 3.3.1 Scrivere del codice più veloce in R

I principali strumenti per rendere il codice di R più veloce sono:

- Ottimizzare il codice R attraverso la vettorizzazione;
- Scrivere la parte chiave del programma (quella computazionalmente più intensiva) in altri linguaggi compilati come C o C++;
- Scrivere il codice in una qualche forma parallelizzata.

Il primo approccio, richiede la comprensione della programmazione funzionale in R e su come R utilizza la memoria. Il secondo richiede la conoscenza di un linguaggio di programmazione aggiuntivo, infine il terzo richiede lo studio di una soluzione alternativa per eseguire il lavoro su più processori.

In questo lavoro, non ci addentreremo molto sulla parte di programmazione funzionale e ottimizzazione di R, il cui approfondimento lo si può trovare, ad esempio, nel libro di (Matloff, 2011), andremo a prendere in considerazione invece gli altri due aspetti, andando ad unire sia la velocità di esecuzione tipica del linguaggio C sia la velocità data dalla parallelizzazione dell'algoritmo.

## 3.4 Strumenti per parallelizzare in R

Per superare la limitazione di R di essere single-threaded sono implementati una serie di pacchetti per aggirare il problema e rendere possibile la parallelizzazione di algoritmi. Questi pacchetti permettono l'esecuzione su diversi processori di più istruzioni simultaneamente, riducendo così i tempi di calcolo. Come visto nei capitoli precedenti però, lo svantaggio sarà dato da un maggior dispendio di risorse per la comunicazione tra processori e un maggior numero di accessi alla memoria. I principali pacchetti, con caratteristiche diverse l'uno dagli altri, sono:

- `snow`

- `multicore`
- `parallel` (pacchetto recente, arrivato dopo la versione di R 2.14.0)

Utilizzando questi pacchetti sarà possibile svolgere la parallelizzazione degli algoritmi. Non ci addentreremo nello specifico di ogni pacchetto (in rete esiste una vasta documentazione dei pacchetti, vedi CRAN), ci limiteremo a descriverne le caratteristiche principali, i vantaggi e gli svantaggi nel loro utilizzo e successivamente faremo il confronto con uno strumento non implementato in R. Una descrizione approfondita di questi pacchetti è possibile trovarla in (McCallum e Weston, 2011).

### 3.4.1 `snow`

`snow` (*Simple Network of Workstations*) è probabilmente il pacchetto più utilizzato per la programmazione parallela in R. Viene utilizzato quando è disponibile un cluster di computer, in cui è possibile eseguire il codice R in parallelo. Può essere utilizzato con diverse tipologie di cluster, con supporto per la comunicazione sia *socket* sia MPI. L'MPI (*Message Passing Intefrace*) consiste in un sistema standardizzato di comunicazione ideato da un gruppo di ricercatori per funzionare su un'ampia gamma di calcolatori paralleli. Esso definisce la sintassi e il significato di una serie di routine basilari per la scrittura di programmi paralleli in Fortran, C o C++. `snow` fornisce un supporto per eseguire facilmente funzioni R in parallelo e ci permette di risolvere sia il fatto che R è *single-threaded*, sia il problema della dimensione della memoria disponibile. La maggior parte delle funzioni parallele eseguite in snow sono variazioni della funzione `lapply()` di R (funzione utilizzata per vettorizzare le operazioni all'interno di un programma). Ciò rende snow piuttosto semplice da utilizzare. Per implementare queste operazioni parallele `snow` utilizza un architettura *master/worker*, dove il processo *master* invia le operazioni da eseguire ai processi *worker*, i quali eseguono le operazioni e restituiscono al *thread* master i risultati ottenuti. Come citato in precedenza, una caratteristica importante di `snow` è che può operare in cluster con diversi meccanismi di trasporto tra *master* e *workers*. `snow` può essere utilizzato con connessioni diverse tipologie di trasporto delle informazioni come per esempio *socket*, MPI, PVM o *Network Spaces*. Si tratta di sistemi per la comunicazione e lo scambio di messaggi tra processori e tra processori

e memoria. Ogni tipologia definisce una serie di routine e convenzioni che dovranno utilizzare i processori per lo scambio di informazioni e dati. Alcune di queste tipologie richiedono dei pacchetti extra da aggiungere per poter essere utilizzate.

Alcuni esempi, in cui può essere utilizzato,sono: simulazioni Monte Carlo, *bootstrap*, validazione incrociata, algoritmi di apprendimento o ricampionamento dei dati. Tutte operazioni che richiedono di essere replicate sempre uguali per un numero $n$ di volte piuttosto alto. È disponibile un buon supporto per la generazione di numeri casuali in modo parallelo, funzione molto importante soprattutto quando si eseguono operazioni come simulazioni, *bootstrap* e apprendimento automatico. `snow` non fornisce direttamente meccanismi per gestire dati di grandi dimensioni, come ad esempio la distribuzione dei dati ai vari *worker*. Gli argomenti di input devono essere immagazzinati in memoria per poter essere utilizzati da `snow`, inoltre tutte le attività vengono tenute in memoria finché non sono restituite alla funzione chiamante in una sorta di lista di risultati. `snow` ci permette quindi di lavorare su una grande quantità di dati, ma dovrà essere l'utente a definire come devono essere utilizzate le risorse.

In breve elenchiamo ora i vantaggi e gli svantaggi nell'utilizzo di questo pacchetto per effettuare la parallelizzazione:

**Vantaggi**

- Pacchetto noto e con una elevata maturità;
- Aumenta la velocità di esecuzione senza aumentare eccessivamente la complessità.

**Svantaggi**

- Può non essere così facile da configurare.

### 3.4.2 `multicore`

`multicore` è un popolare pacchetto di programmazione parallela per l'uso su macchine multi-processore o multi-core. La prima versione è stata rilasciata nel 2009 e al momento è disponibile per *Linux* e *Mac* ma non per *Windows* a causa del modo in cui avviene la creazione dei *thread*. I nuovi *thread* vengono creati utilizzando

una operazione di duplicazione dei *thread* chiamata *fork*, Questa operazione consiste nella creazione da parte del thread padre di una copia di se stesso (figlio), e questo modo di creare nuovi thread è diverso tra macchine Linux o Mac OS e Windows. Questo pacchetto è diventato popolare in quanto sfrutta in modo intelligente ed efficace il *fork*, permettendo di implementare una versione parallela della funzione `apply()` ancora più facile da usare, rispetto a quella creata da `snow`.

Utilizziamo questo pacchetto quando abbiamo uno script R che impiega molto tempo ad essere eseguito e al suo interno utilizza la funzione `apply()` per vettorizzare. In questo caso attraverso pochi semplici comandi è possibile parallelizzare l'esecuzione diminuendo i tempi di attesa del risultato finale dello script.

`multicore` è destinato, per come è scritto e implementato, a girare su sistemi *Posix* (*Linux* e *MAC*) multi-processore o *multi-core*. Può essere anche utilizzato su singole macchine posizionate all'interno di un cluster di macchine Linux, ma non può gestire più macchine collegate contemporaneamente, come può fare `snow` ad esempio. Poiché `multicore` è piuttosto efficiente, è in grado di gestire problemi paralleli a grana più fine rispetto a snow, ma è ancora destinato per lo più a problemi a grana grossa.

Non può competere con la programmazione *multi-thread* per l'esecuzione di parallelismo a grana fine, come per esempio operazioni vettoriali (motivo per cui non è stato utilizzato per lo sviluppo del nostro *CART* parallelo). Questo pacchetto, a differenza di `snow`, che permetteva di risolvere sia il problema del *single-threaded* di R sia il problema legato alla memoria, riesce a risolvere solo il primo, in quanto essendo eseguito su un'unica macchina non vi è la possibilità di ampliare ulteriormente la memoria. Tuttavia, poiché *fork* copia i dati solo quando questi vengono modificati `multicore` riesce comunque ad avere un uso efficiente della memoria.

**Vantaggi**

- Semplice ed efficiente;
- Facile da installare ed eseguire su macchine multi-processore o multi-core;
- Non richiede complicate operazioni di configurazione.

**Svantaggi**

- Può essere utilizzato solo su una macchina;

- Non è supportato dal sistema operativo Windows;
- Non offre il supporto alla generazione di numeri causali in parallelo (funzione molto importante in alcune applicazione come abbiamo visto nel paragrafo precedente).

### 3.4.3 `parallel`

Un nuovo pacchetto di programmazione parallela è stato incluso dalla versione 2.14.0 di R. Si tratta di un pacchetto derivato dalla combinazione dei pacchetti `snow` e `multicore`, fornisce le stesse funzioni di questi pacchetti integrandole insieme, in modo da utilizzare tutte le funzioni disponibili con un unico pacchetto. Questo è uno sviluppo interessante, dal momento che rende il calcolo parallelo potenzialmente implementabile in altri pacchetti standard, senza che l'utente finale si renda conto di sfruttarlo. Una caratteristica importante di questo pacchetto è l'integrazione con un nuovo tipo di generatore di numeri casuali parallelo.

Essendo basato su `snow` e `multicore`, il funzionamento di questo pacchetto è molto simile a quanto abbiamo descritto nei paragrafi precedenti. Abbiamo delle funzioni che vanno a sostituire le funzioni `apply()` per la vettorizzazione che ci permettono di eseguire diverse operazioni sui dati in contemporanea. Nel nostro lavoro non viene preso in considerazione in quanto si tratta di un pacchetto ancora nuovo, e presenta dei limiti in alcune implementazioni. Una su tutte è che al momento può essere utilizzato solo su macchine *Windows*, e la presenza di cluster che utilizzano *Windows* non è molto elevata.

**Vantaggi**

- Non richiede di essere installato;
- Risolve sia il problema di R del *single-threaded*, sia quello legato alla memoria;
- Offre un ottimo supporto alla generazione di numeri casuali in parallelo.

**Svantaggi**

- Può essere utilizzato facilmente solo su macchine *Windows*:
- Può essere difficile da configurare su macchine *Linux*.

## 3.5 Strumenti per parallelizzare in C

Come abbiamo visto in precedenza, parallelizzando con R possiamo ottenere una notevole accelerazione nell'esecuzione del nostro codice. Questo consente di mantenere la comodità e la potenza espressiva di R mentre migliora i tempi di esecuzione su dataset di grandi dimensioni e su grandi applicazioni. Se la parallelizzazione con R fornisce un miglioramento sufficiente delle prestazioni allora va tutto bene, ma R in parallelo è sempre R, con ancora tutti i problemi legati, per esempio, al fatto di essere un linguaggio interpretato. Una soluzione potrebbe essere quella di utilizzare un altro linguaggio per le parti computazionalmente più dispendiose, e poi richiamarla all'interno del programma R. Vedremo che sarà possibile scrivere del codice sequenziale in R e richiamare poi del codice C parallelo per svolgere quelle operazioni computazionalmente onerose.

La parte di codice parallelo in C che studieremo in questo lavoro riguarda solo i sistemi *multi-core* o multi-processori con memoria condivisa. Attualmente un numero sempre maggiore di dispositivi utilizza delle CPU multi-core, e qualsiasi CPU di questo tipo può essere sfruttata per il calcolo parallelo. Questi calcoli vengono eseguiti contemporaneamente con *thread* analoghi, i quali sono oggetti molto simili ai *workers* che abbiamo trovato con il pacchetto `snow`. Nelle applicazioni di calcolo intensive, generalmente, vengono impostati tanti thread quanti sono i core presenti nella CPU, questo per cercare di sfruttare al meglio le risorse disponibili al fine di ridurre i tempi di esecuzione. Se la macchina dispone di più core, viene strutturata come un sistema a memoria condivisa, ovvero tutti i core accederanno alla stessa RAM. Questa particolare struttura, come abbiamo visto nel Capitolo 1 rende più facile la comunicazione tra core, ma al contempo crea dei problemi, in quanto occorre evitare che più core modifichino lo stesso indirizzo di memoria nello stesso istante (gestione della concorrenza).

Lo strumento che andremo a valutare per effettuare queste parallelizzazioni è OpenMP.

### 3.5.1 OpenMP

#### Concetti generali e principi

OpenMP è un API (*Application Program Interface*), che supporta la programmazione multi-processore a memoria condivisa in C, C++ e Fortran sulla maggior parte delle architetture di processori e sistemi operativi. OpenMP differisce dagli altri strumenti per la parallelizzazione per tre caratteristiche fondamentali:

- Portabilità (il codice può essere eseguito su qualsiasi piattaforma lo supporti senza necessità di cambiamenti);
- È possibile utilizzare le stesse direttive con diversi linguaggi di programmazione;
- Astrazione (il compilatore si occupa solo di creare e gestire i *thread*).

OpenMP è composto da una serie di direttive del compilatore, routine di libreria e variabili di ambiente che influenzano il comportamento del programma in fase di esecuzione, cercando allo stesso tempo di essere semplici e chiare ma anche permettere un controllo completo sulla parallelizzazione. (Meadows *e altri*, 2009)

#### Modello concettuale

Il modello concettuale prevede che un programma che utilizza OpenMP contenga al proprio interno parti eseguite in maniera sequenziale e parti parallele. Le parti eseguite in modo sequenziale vengono sempre eseguite dallo stesso *thread* (detto *thread master*), mentre la parte parallela utilizzerà molti *thread* alla volta per ridurre i tempi di calcolo. Un programma che viene eseguito utilizzando le direttive OpenMP rappresenta un processo. Questo processo, durante la sua esecuzione, attiva altri processi più leggeri detti *thread* nel momento in cui si entra nella regione parallela. A questo punto, ogni *thread* esegue le istruzioni che gli vengono assegnate utilizzando le variabili presenti nella memoria. Le variabili possono essere definite nello *stack* del *thread* (variabile privata) o nella memoria condivisa, accessibile quindi da tutti i *thread* (variabile pubblica). I *thread* quindi andranno a suddividersi il lavoro da svolgere, e questa suddivisione, chiamata *work-sharing*, può avvenire in diversi modi (Chergui e Lavallée, 2012)

- Esecuzione di un ciclo dividendo le iterazione tra i *thread* (*Parallel Loop*)
- Esecuzione di più sezioni di codice ma solo una per *thread* (*Parallel Section*)
- Esecuzione di più occorrenze della stessa procedura da diversi *thread* (*Orphaning*)

Sarà responsabilità dello sviluppatore introdurre le direttive OpenMP opportune per la parallelizzazione del proprio codice. All'inizio dell'esecuzione del programma, il sistema operativo crea una regione parallela con il modello *fork-join*. Questo modello prevede la suddivisione della parallelizzazioni in due fasi, appunto la fase di *fork* e la fase di *join*. Supponiamo che un processo sia in esecuzione in modo sequenziale, arrivato ad un certo punto vi saranno delle istruzioni che permetteranno al processo principale di suddividersi in processi più piccoli e leggeri (*thread*), questa fase viene definita come *fork*. Una volta che tutti i *thread* avranno eseguito le operazioni a loro assegnate vi sarà una fase, detta *join*, che si occuperà di unire i risultati provenienti dai diversi thread per poi proseguire con l'esecuzione principale del processo principale.

Quando si entra in una regione parallela, il *thread* master crea/attiva una squadra di thread figli che scompaiono/ibernano alla fine della parte parallelizzata, mentre il *thread* master continua l'esecuzione del programma fino all'inizio di una nuova regione parallela.

A seconda del livello di profondità a cui viene effettuata la parallelizzazione del codice possiamo avere differenti granularità del parallelismo (Paragrafo 2.3)

- Grana fine (*Loop level parallelism*): abbiamo il *fork/join* attorno ai cicli intensivi, come viene rappresentato in Figura 3.1)
- Grana grossa: *fork* all'inizio del codice, *join* alla fine, come mostrato in Figura 3.2

### Modello di memoria

Il modello di memoria si basa su alcuni punti fondamentali:

- Tutti i *thread* possono accedere ad una memoria condivisa in cui tutti possono scrivere e leggere;

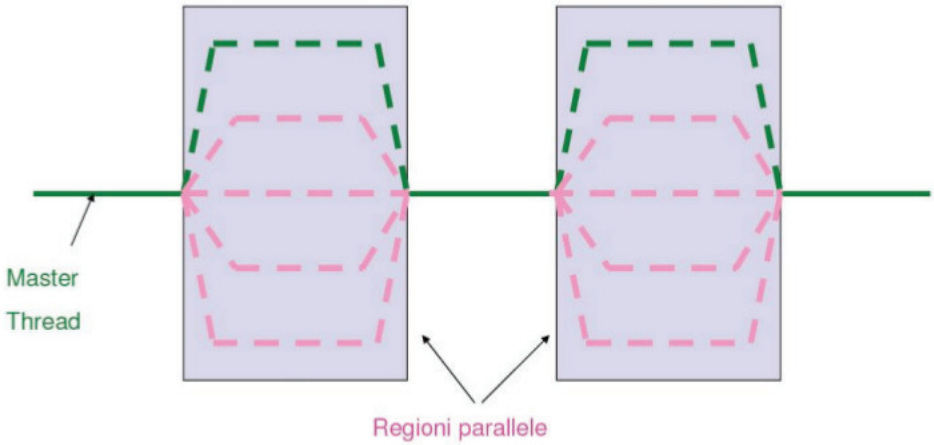


Figura 3.1. Parallelismo a '*grana fine*'

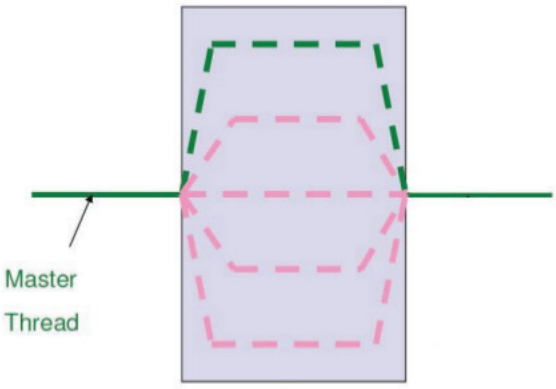


Figura 3.2. Parallelismo a '*grana grossa*'

- Ogni *thread* ha uno spazio per variabili proprie accessibili in modo esclusivo e memorizzate nello *stack* del *thread*;
- Le variabili originali sono quelle definite nel programma seriale;
- OpenMP fornisce degli strumenti per coordinare i *thread*, ma la corretta parallelizzazione è compito dello sviluppatore.

La prima cosa che occorre sapere nel momento in cui vogliamo utilizzare OpenMP per parallelizzare porzioni di codice è che di *default* le variabili sono condivise. Esistono tuttavia delle direttive che ci permettono di modificare la visibilità delle variabili. Sarà infatti possibile dichiarare come private delle variabili, queste invece di essere immagazzinate nella memoria condivisa, verranno inserite all'interno di una *cache*. Questa *cache* sarà visibile solo dal *thread* a cui quella *cache* è collegata.

# Capitolo 4

# Parallelizzazione Algoritmo CART

## 4.1 Alberi di regressione e classificazione

### 4.1.1 Introduzione

Il modo più semplice per approssimare una qualunque funzione $y = f(x)$, con $x \in \mathbb{R}$, è quello di usare una funzione approssimante a gradini, cioè una funzione costante a tratti su intervalli (Azzalini e Scarpa, 2004). Questa è l'idea che sta alla base dell'utilizzo degli alberi di classificazione e regressione nel *data mining*.

Per definire quale sia la migliore funzione da utilizzare esistono principalmente tre aspetti da stabilire:

1. quante suddivisioni effettuare dell'asse in cui sono rappresentate le variabili x (ascissa);
2. dove scegliere i punti di suddivisione;
3. quale valore di ordinata (valore della funzione $f(x)$) assegnare ad ogni intervallo.

Serviranno quindi dei metodi per poter definire tutti gli aspetti precedentemente elencati al fine di ottenere la miglior funzione approssimante della funzione originale. È possibile estendere con i dovuti aggiustamenti lo schema sopra elencato anche al caso di funzioni in $p$ variabili. Questo modo di rappresentare la funzione permette di poter utilizzare un albero binario per la visualizzazione. L'albero è costituito da

una struttura composta di varie parti, chiamati nodi, i quali rappresentano delle affermazioni di tipo logico. Per capire in quale nodo foglia dovrà essere contenuta l'osservazione si dovrà iniziare dal nodo radice e, a seconda che l'affermazione sia vera o falsa, scorrere l'albero a sinistra o a destra del nodo che stiamo analizzando. Una volta raggiunto un nodo foglia sappiamo che quella particolare osservazione apparterrà a quel gruppo. Il nodo foglia sarà rappresentato dal valore della funzione approssimante.

Si tratta in pratica di un metodo gerarchico per descrivere una partizione dell'insieme delle unità statistiche basata sulla relazione tra una variabile risposta e più variabili esplicative. Tutte queste variabili potranno essere sia quantitative sia qualitative.

### 4.1.2 Algoritmi seriali per alberi

Vediamo come vengono costruiti ricorsivamente gli alberi utilizzando le strategie di crescita note.

Dato un *training set* $T$ con le classi $C_1$,$C_2$,...,$C_k$. L'albero di decisione sarà costruito ricorsivamente usando la strategia *depth-first divide-and-conquer greedy* (Shafer *e altri*, 1996), con i seguenti casi:

- **Caso 1**: $T$ contiene tutti i casi che appartengono alla stessa classe $C_j$. Il nodo foglia per $T$ viene creato ed è assegnato alla classe $C_j$.
- **Caso 2:** $T$ contiene casi che appartengono ad una classe o più. Viene scelta la miglior divisione del singolo attributo, viene testato e suddiviso $T$ in singole classi che contengono molti casi. La suddivisione di $T$ da dei sottoinsiemi di $T$ che sono: $T_1$,$T_2$,...,$T_n$ ($T_n$ nel caso di alberi non binari). La divisione su $T$ è scelta nell'ordine di ottenere risultati mutualmente esclusivi.

Uno dei metodi utilizzati più di frequente per la crescita degli alberi negli algoritmi che vengono implementati è il *metodo di Hunt* (Hunt *e altri*, 1966) (anche se non è l'unico metodo possibile). Con questa metodologia di lavoro abbiamo che l'albero di decisione è costruito in due fasi:

1. Crescita

2. Potatura

Nell'*algoritmo di Hunt* per la costruzione degli alberi di decisione i dati di stima (*training*) sono partizionati ricorsivamente usando la strategia *depth-first greedy* (Shafer *e altri*, 1996), fino a che tutti i record appartengono alle etichette delle classi. Il dataset è residente in memoria ed è ordinato ad ogni nodo con l'obiettivo di determinare il miglior attributo per dividere l'insieme delle unità statistiche. Uno degli svantaggi dell'implementazione degli alberi di decisione seriale è la bassa accuratezza della classificazione quando l'insieme dei dati di stima è grande. Allo scopo di ridurre il tempo di esecuzione, vista la grande complessità computazionale associata a grandi insiemi di dati di stima, l'intero insieme di dati di stima viene caricato nella memoria.

Altri metodi, che non utilizzano l'*algoritmo di Hunt* per l'implementazione degli alberi sono gli algoritmi SLIQ (Mehta *e altri*, 1996) e SPRINT (Shafer *e altri*, 1996).

Nell'implementazione seriale di SPRINT e SLIQ, l'insieme dei dati di stima viene partizionata ricorsivamente usando la tecnica *breadth-first* fino a che tutto il dataset appartiene alla stessa classe e c'è un ordinamento temporale. Anche i dati di stima non sono residenti in memoria ma sono su disco, che fa si che sia possibile scalare i dati (utilizzare dataset molto grandi). Questo approccio migliora l'accuratezza della classificazione e riduce gli errori di classificazione.

Gli alberi di decisione basati sull'*algoritmo di Hunt* (Hunt *e altri*, 1966) possono essere classificati come classici alberi di decisione che possono essere implementati solo in modo seriale. L'implementazione di questi algoritmi con *pattern* paralleli è uno degli obiettivi che sta perseguendo la ricerca in questo settore.

Gli svantaggi associati all'uso di alberi di decisione classici sono:

- **Maneggiare dati con 'rumore'**: non sempre producono modelli di decisione con alta accuratezza di classificazione quando i dati di stima contengono rumore o troppi dettagli;

- **Produzione dello stesso albero**: dato lo stesso insieme di stima e le stesse condizioni, producono sempre lo stesso albero invece di produrre più alberi con una flessibilità di scegliere quello meno propenso a sbagliare;

- **Importanza dell'errore**: Differenti errori sorgono utilizzando questi algoritmi, ma alcuni errori hanno maggiore priorità di altri e necessitano di essere minimizzati per avere delle buone classificazioni. Gli errori sono il risultato delle decisioni prese nella costruzione dell'albero.

Vediamo ora un piccolo schema con le principali implementazioni per gli alberi di classificazione e regressione, e le relative caratteristiche:

### IDE3 (*Iterative Dichotomiser 3*)

Questa prima tipologia di albero, ideata da Quinlan (Quinlan, 1986) (Quinlan, 1987), si basa sull'*algoritmo di Hunt* ed è implementato in modo seriale. Come tutti gli algoritmi di questo tipo viene costruito l'albero finale utilizzando le due fasi di crescita e potatura. Durante la costruzione dell'albero i dati sono ordinati con l'obiettivo di scegliere l'attributo migliore per la suddivisione. Questo modello accetta solo attributi categoriali per la costruzione dell'albero, e ciò fa si che non sia molto utilizzabile in applicazioni reali dove i dati sono di natura sempre diversa. Prima di poter costruire l'albero sarà necessario eseguire una pre-analisi dei dati, ma nonostante ciò l'algoritmo non da risultati particolarmente accurati quando c'è troppo rumore o molti dettagli nell'insieme di stima.

### C4.5

Di fatto, l'algoritmo C4.5 è un miglioramento dell'IDE3 (Quinlan, 1986). Le caratteristiche di implementazione seriale e l'essere basato sull'*algoritmo di Hunt* non mutano rispetto al metodo precedente. Ciò che cambia è che ora vengono accettati attributi sia continui che categoriali per la costruzione, cosa che permette un utilizzo molto più ampio dell'algoritmo. Come IDE3, i dati sono ordinati ad ogni nodo dell'albero allo scopo di determinare il miglior attributo di suddivisione. Il metodo utilizzato per questa divisione delle osservazioni è il *metodo di impurità*. Passando alla fase di potatura dell'albero, in C4.5 la potatura viene eseguita rimpiazzando il nodo interno con un nodo foglia che riduce l'*error-rate*. Il metodo di valutazione per effettuare la potatura di un ramo permette di ridurre l'errore di classificazione dovuti al rumore o ai troppi dettagli.

### CART (*Classification And Regression Tree*)

Come suggerisce il nome permette la costruzione sia di alberi di classificazione che di regressione. Rendendo di fatto CART (Breiman *e altri*, 1984) l'unico metodo basato sul *metodo di Hunt* che può anche essere usato per la regressione. Molto simile a C4.5 per caratteristiche di costruzione e implementazione, CART permette di costruire l'albero con attributi sia numerici che categoriali e permette inoltre la gestione dei dati mancanti. Utilizza diversi criteri per suddividere i dati, i quali sono ordinati ad ogni nodo per determinare la miglior suddivisione. Per esempio l'albero di classificazione costruito con CART è basato sulla suddivisione binaria degli attributi. Infine, la potatura è effettuata utilizzando una porzione dei dati di stima. Questo sarà il metodo che prenderemo come modello per lo sviluppo del nostro algoritmo parallelo che vedremo nel Paragrafo 4.4.

### SLIQ (*Supervised Learning In Ques*)

È il primo algoritmo che non si basa sull'algoritmo di Hunt. Si tratta di un algoritmo veloce e scalabile che può essere implementato in serie o in parallelo, utilizzando per la costruzione sia attributi numerici sia categoriali.

Partiziona i dati utilizzando *breadth-first greedy strategy* (Mehta *e altri*, 1996) che è integrato con una tecnica di pre-ordinamento durante la fase di costruzione dell'albero. Con la tecnica di pre-ordinamento, l'ordinamento dei nodi, è eliminato e rimpiazzato con un ordinamento unico, con l'uso di una lista della struttura dei dati per ogni attributo in modo da determinare il miglior punto di suddivisione. Uno degli svantaggi è che utilizza una lista di dati di struttura che è residente in memoria che è una restrizione sui dati e sulla loro dimensione massima. Infine, occorre dire che utilizza il *Minimum Description Length Principle* (MDL) per la potatura.

### SPRINT (*Scalable Parallelizable Induction of decision Tree algorithm*)

Si tratta di un classificatore veloce e scalabile che non è basato sull'*algoritmo di Hunt*, ma che piuttosto partiziona i dati di stima ricorsivamente usando *breadth-first greedy technique* (Mehta *e altri*, 1996) prima che ogni partizione appartenga alla stessa foglia o classe (Shafer *e altri*, 1996). È un espansione di SLIQ e ne conserva tutte le caratteristiche principali. Come il metodo SLIQ, anche SPRINT utilizza un

'ordinamento in una volta' degli item e non ha restrizione sulla dimensione dei dati di input, infatti la lista di attributi e l'istogramma non sono residenti in memoria. Tutto questo permette di rendere SPRINT utilizzabile per grandi dataset poiché rimuove tutte le restrizioni sui dati.

## 4.2 Alberi CART

### 4.2.1 Algoritmo e specifiche

Il CART (Breiman *e altri*, 1984) è una procedura non parametrica che può essere utilizzata per analizzare sia dati quantitativi sia dati qualitativi applicando la stessa metodologia. I CART vengono utilizzati quando si vogliono analizzare relazioni che sono complesse e non lineari, e che a causa di queste caratteristiche sono tendenzialmente difficili da identificare. Questa tipologia di analisi ci permette inoltre di andare a selezionare quali sono le variabili più importanti per spiegare determinati fenomeni di una popolazione. La struttura ad albero binario dell'output infatti, ci permette di trattare strutture dati molto complesse producendo in uscita grafici che sono di facile interpretazione da parte dell'utente. Ultima motivazione per cui vengono utilizzati, ma non per questo meno importante, è la previsione. Possiamo infatti andare a fare previsioni sulla popolazione senza imporre particolari ipotesi sulla forma distributiva delle variabili utilizzate nell'analisi.

Gli elementi chiave di un'analisi CART sono un insieme di regole per:

1. suddividere ogni nodo dell'albero;
2. stabilire quando un albero è completo (massima crescita possibile);
3. assegnare una etichetta (o un valore nel caso della regressione) ad ogni nodo terminale.
4. potare l'albero per ottenere il compromesso migliore tra costo e complessità dell'albero, cioè per la dimensione dell'albero.

Il metodo CART si sviluppa guardando a tutte le possibili suddivisioni, per tutte le variabili considerate nell'analisi. Quindi, per ogni nodo, il metodo andrà a valutare tutte le possibili suddivisioni di quel nodo calcolandola con tutte le variabili.

A questo punto verrà scelta la suddivisione che soddisfi nel modo migliore un determinato criterio scelto in precedenza. Fatto ciò, l'algoritmo continuerà ad andare sempre più in profondità fintanto che non viene raggiunta la dimensione massima dell'albero. Storicamente la costruzione della prima generazione di alberi di classificazione avveniva dividendo ogni nodo finché non smetteva di essere soddisfatto un qualche criterio di bontà di bipartizione. Quando la qualità di un particolare nodo cadeva sotto una certa soglia, il nodo raggiunto veniva dichiarato terminale. Quando tutti i rami uscenti dal nodo radice raggiungevano dei nodi terminali, il processo di costruzione dell'albero veniva considerato ultimato. Sebbene sia piuttosto datata questa tecnica è implementata in molti software commerciali, purtroppo conduce a valutazioni a volte non perfettamente corrette. La nuova filosofia del CART è diversa, invece di cercare di stabilire se un nodo è terminale o meno, il CART procede facendo crescere l'albero finché è possibile esaminando tutti i possibili sotto-alberi. Tra tutti i sotto-alberi possibili si sceglierà il migliore in base ad un criterio detto del minimo costo-complessità.

### 4.2.2 Metodi in R per la creazione di alberi CART

All'interno di R è possibile trovare già numerosi metodi implementati in modo efficiente dentro a pacchetti liberamente scaricabili. I principali pacchetti R che vengono utilizzati per gli alberi CART sono il pacchetto `tree` e il pacchetto `rpart`. I due metodi si differenziano per il tipo di oggetto restituito e una serie di dettagli. L'algoritmo implementato è esattamente lo stesso: entrambi infatti si basano sull'algoritmo CART introdotto da (Breiman *e altri*, 1984). Altro metodo per l'utilizzo di alberi CART in R è il pacchetto `CORElearn`. Questo pacchetto però ha una particolarità rispetto ai due precedenti, infatti utilizza la parallelizzazione durante la selezione delle variabili da utilizzare per la suddivisione del nodo. Come detto nel Paragrafo 4.2.1, prima di scegliere quale variabile utilizzare per la suddivisone del nodo in esame, vengono calcolate tutte le possibili suddivisioni con tutte le variabili per decidere in base ad un criterio scelto in precedenza quale sia la variabile migliore. In `CORElearn` questa parte di selezione della migliore variabile viene eseguita su più processori. I diversi processori lavoreranno sugli stessi dati ma ognuno valuterà la suddivisone con una diversa variabile. Ciò contribuisce a rendere più veloce il

processo di selezione, diminuendo di conseguenza i tempi di esecuzione dell'intero algoritmo.

## 4.3 Idea algoritmo parallelo e specifiche di realizzazione

L'idea che sta alla base di questo lavoro, come è emerso nei capitoli precedenti, è quella di provare a sfruttare le risorse hardware disponibili, parallelizzando gli algoritmi di *data mining*. Riuscire ad eseguire più operazioni contemporaneamente può portare ad un risparmio significativo di tempo, e i metodi per ridurre questi tempi di esecuzione possono essere vari. Come abbiamo visto nel Paragrafo 4.2.2 esiste il pacchetto `CORElearn` che parallelizza la crescita dell'albero, questa parallelizzazione però è fatta solamente sulla selezione delle variabili. Non siamo di fronte ad un vero e proprio algoritmo parallelo, ma vengono applicate delle tecniche per ridurre i tempi di esecuzione sfruttando più risorse in contemporanea portandoci quindi a definire anche questa una parallelizzazione. Ciò che si è voluto fare in questo lavoro è diverso. L'obiettivo era quello di ottenere un vero e proprio algoritmo parallelo che permettesse di sfruttare più processori nella crescita dell'albero e non solo nella selezione delle variabili. La base da cui siamo partiti è stato l'algoritmo dell'albero CART implementato nel pacchetto `tree` di R. Algoritmo già implementato e dal cui codice siamo partiti per le modifiche necessarie al raggiungimento del nostro obiettivo.

L'idea, ancora non presa in considerazione in altri lavori sull'argomento, prevede che dopo un'iniziale crescita dell'albero non ricorsiva, si entri nella regione parallela in cui ad ogni nodo viene applicato l'algoritmo ricorsivo di crescita che fa sì che i rami dell'albero crescano simultaneamente. Infine, occorre specificare che dentro ad ogni nodo ci saranno dati diversi, poiché la crescita dei diversi rami è indipendente l'uno dall'altro, come illustreremo nel Paragrafo 4.4.2.

Per implementare il nostro algoritmo parallelo abbiamo deciso di utilizzare il linguaggio C. Inizialmente l'idea era quella di implementarla completamente in R, ma dopo studi e ricerche è stato ritenuto più efficiente andare a lavorare direttamente con la parte in C del pacchetto originale.

Tale decisione è stata presa principalmente per due motivi:

1. Il primo motivo è che volevamo sfruttare ciò che era già presente e funzionante. I pacchetti scaricabili da **CRAN** (server contenente tutti i pacchetti implementati in R) contengono funzioni che al loro interno richiamano parti di codice in altri linguaggi di programmazione quali C, C++ e Fortran.

2. Il secondo, che è anche legato al primo, è l'efficienza con cui il linguaggio C permette di eseguire cicli e altre operazioni rispetto ad R.

A questo punto abbiamo eseguito una ricerca su quali fossero gli strumenti adatti al raggiungimento del nostro obiettivo, e per quanto riguarda gli strumenti per la programmazione parallela la scelta è ricaduta su **OpenMP**, le cui caratteristiche sono state illustrate precedentemente nel Paragrafo 3.5.1.

Questa scelta è stata fatta basandoci su due ragionamenti principali. Il primo è che, essendo questo un primo approccio all'argomento della parallelizzazione degli algoritmi, non ci è sembrato opportuno andare a complicare la situazione inserendo anche la gestione della comunicazione tra processori e tutte le altre necessità che avrebbe un cluster di computer su cui eseguire l'algoritmo. I cluster di computer risultano particolarmente utili e di facile utilizzo quando è possibile andare a sfruttare funzioni che gestiscono in modo automatico la comunicazione, come per esempio nel caso di algoritmi computazionalmente intensivi che replicano sempre le stesse funzioni all'interno di cicli (bootstrap, ecc.).

Il secondo invece, riguarda più un discorso legato all'hardware oggi disponibile. Nella maggior parte delle macchine attuali sono disponibili processori *multi-core*, ed una implementazione che permetta di sfruttare questa componente potrebbe risultare utile per una vasta categoria di utenti che non possono disporre di cluster di computer ma allo stesso tempo potranno ottenere dei miglioramenti nelle performance degli algoritmi.

Riprendendo le classificazioni illustrate nel Capitolo 1 e le considerazioni appena espresse, il modello di macchina pensato per lo sviluppo del nostro algoritmo prevede l'utilizzo di un organizzazione della *memoria di tipo condiviso*. Questo, come detto in precedenza, ci permette di ridurre l'*overhead* causato dalla comunicazione tra processori ma viceversa richiede ci siano una serie di controlli per evitare l'accesso

simultaneo alle stesse variabili presenti nella memoria. Per quanto riguarda l'architettura di memoria, è stata utilizzata la *Single Instruction Multiple Data* poiché bene si presta ad implementazioni che sfruttano processori *multi-core*.

## 4.4 Implementazione algoritmo parallelo via OpenMP

### 4.4.1 Algoritmo sequenziale

Andiamo ora ad illustrare il funzionamento dell'algoritmo sequenziale per la crescita dell'albero. Analizziamo le macro operazioni compiute dall'algoritmo, partendo dai controlli per la correttezza dei dati fino ad arrivare all'organizzazione finale dell'output generato dall'algoritmo.

Possiamo immaginare l'algoritmo suddiviso in una serie di macro operazioni:

1. Verifica validità dataset;
2. Inizio parte eseguita in C;
3. Assegnazione variabili passate e allocazione nuove variabili;
4. Parte di codice ricorsivo per la crescita dell'albero;
5. Preparazione variabili per successive elaborazioni;

L'elenco precedente ci permette di capire come si sviluppa l'algoritmo durante l'esecuzione. Occorre tenere presente che le uniche parti che vengono eseguite in R sono la prima e l'ultima, ovvero la verifica della validità del dataset e la preparazione delle variabili per le successive elaborazioni grafiche o di riepilogo, mentre tutte le altre parti vengono eseguite completamente nella parte in C.

Analizziamo quali sono le azioni che vengono svolte all'interno di ogni macro-operazione.

Nel momento in cui viene richiamata la funzione vengono eseguiti inizialmente una serie di controlli per verificare se il dataset passato alla funzione sia effettivamente un dataset valido, con il quale sia possibile eseguire tutte le operazioni necessarie alla crescita dell'albero. Questi controlli vengono effettuati utilizzando una serie di funzioni già presenti in R. Una volta verificato ciò, verrà richiamata

la funzione implementata in C (chiamata `BDRgrow1()`). Da questo punto in avanti tutte le operazioni di calcolo dei punti di suddivisione, calcolo dei criteri di stop dell'algoritmo, suddivisone delle osservazioni, ecc. vengono eseguiti in C. Questa scelta è stata fatta proprio a causa della maggiore efficienza di questo linguaggio di programmazione rispetto ad R, visto il grande sforzo computazionale necessario all'esecuzione di tale algoritmo. Un volta all'interno della parte in C, la prima operazione che viene eseguita è quella di assegnare le variabili inizializzate in R, e passate alla funzione, a delle variabili globali che verranno utilizzate per tutta l'esecuzione del programma.

A questo punto, terminata la fase di inizializzazione, può essere avviata la parte ricorsiva dell'algoritmo, in cui verrà richiamata inizialmente una funzione per il calcolo della suddivisione del nodo radice e poi via via verrà sviluppato l'albero suddividendo di volta in volta i nuovi nodi creati.

Il criterio di fermata dell'algoritmo prevede di non suddividere un nodo se vi è un decremento della devianza inferiore a un valore minimo, o nel caso in cui il nodo abbia una numerosità ritenuta troppo bassa per ulteriori divisioni al suo interno.

Una volta terminata la fase ricorsiva abbiamo le variabili globali, create all'inizio del programma, che conterranno tutti i valori necessari alla rappresentazione della soluzione trovata con l'algoritmo.

A questo punto, termina la funzione richiamata in C e riprende l'esecuzione della parte scritta in R, in cui a questo punto vengono presi dalle variabili i risultati e organizzati in modo da poter essere facilmente visualizzabili e rappresentati attraverso, per esempio, grafici e tabelle di riepilogo.

### 4.4.2 Modifiche per algoritmo parallelo

Abbiamo descritto nel paragrafo precedente quali sono i passi compiuti durante l'esecuzione dell'algoritmo sequenziale. Vediamo ora quali sono stati i cambiamenti portati a questo algoritmo.

L'idea di base, utilizzata per la parallelizzazione, è stata quella di creare una prima parte in cui, dopo aver specificato il numero di *thread*, si procede con la suddivisione dei dati fino ad arrivare ad avere un numero di nodi pari al numero di *thread* definiti.

A questo punto, utilizzando le direttive proprie di OpenMP, abbiamo suddiviso il lavoro ed eseguito l'algoritmo sequenziale originale su ogni *thread*. Ciò è possibile in quanto l'algoritmo CART esegue una *ottimizzazione miopica* della devianza, nel senso che una volta che i dati vengono separati, le successive suddivisioni migliori vengono ricercate solo all'interno dell'insieme ottenuto al passo precedente e non rispetto a tutte le osservazioni. Questo ha permesso di poter dividere fino ad un certo punto i dati in modo sequenziale, e poi una volta suddivisi utilizzare la ricorsione vista in precedenza per completare la suddivisione delle osservazioni.

### 4.4.3 Implementazione

Vediamo ora la parte di codice parallelizzata con le direttive OpenMP.

```
int q=0;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(q)
{
        #pragma omp for schedule(static,1)
        for(i=0; i<NUM_THREADS; i++)
        {
         q=omp_get_thread_num();
         BDRgrow1_R(XR, yR, wR, levelsR, junkR,
                nobsR, nvarR, nodeR, varR, cutleftR,
                cutrightR, nR, devR, yvalR, yprobR,
                minsizeR, mincutR, mindevR, nnodeR,
                whereR, nmaxR, GiniR, orderedR, i);
        }
}
```

Analizziamo brevemente la porzione di codice sopra riportata. Questo codice è il risultato di numerose prove effettuate utilizzando diversi metodi di parallelizzazione attraverso le direttive OpenMP. I comandi specifici di OpenMP che vediamo in questa porzione di codice sono:

- `omp_set_num_threads(NUM_THREADS)` : è una *routine* di libreria che specifica il numero di *thread* che saranno avviati dalla regione parallela. NUM_THREADS

è una variabile d'ambiente che viene inizializzata all'entrata e che resterà invariata per tutta l'esecuzione del programma.

- `#pragma omp parallel private(q)` : si tratta di una direttiva di OpenMP, la quale andrà a costruire ed avviare un numero di *thread* pari a quanti sono stati definiti dalla *routine* di libreria. L'argomento `private(q)` indica che la variabile `q` sarà una variabile privata del singolo *task*, e quindi visibile solo all'interno di esso e non all'esterno del segmento di codice.

- `#pragma omp for schedule(static,1)` : direttiva che permette di distribuire le iterazioni del ciclo `for` ai *thread* inizializzati. L'attributo `schedule` ci permette di decidere come verrà fatta questa suddivisione. Nel nostro caso specifico poiché ogni iterazione andrà ad essere eseguita su un *thread* diverso abbiamo impostato i valori `static`, che definisce i *chunk* (parti) tutti della stessa dimensione e `1` che indica il numero di iterazioni che ogni *thread* dovrà eseguire di quel ciclo.

- `omp_get_thread_num` : questa routine di libreria estrae il nome del *thread* che sta eseguendo, in quel momento, quella porzione di codice. (Miller e Mattson, 2011) (Kumbaro *e altri*, 2007) (Org., 2002)

Per permettere di eseguire la parte parallela sfruttando l'algoritmo sequenziale noto abbiamo dovuto creare delle variabili globali che raccogliessero al termine dell'esecuzione di ogni *thread* i vari risultati. Valori che una volta completata l'esecuzione parallela necessitano di modifiche adeguate per essere poi uniti per ottenere il risultato finale dell'albero. Queste variabili, indicate con una `R` finale, sono un insieme di puntatori a puntatori che ci permetteranno di avere tante variabili quanti sono i *thread*, e di tenere traccia dei vari rami dell'albero che dovranno essere poi uniti insieme per formare l'albero completo.

Una volta raccolti tutti i dati provenienti dai diversi *thread*, si procede con l'unione vera e propria dei risultati. Questa operazione è piuttosto delicata in quanto richiede molta attenzione per ciò che riguarda la numerazione dei nodi e il mantenimento dei collegamenti logici tra il contenuto di tutte le variabili coinvolte nella crescita dell'albero.

Vedremo, nel paragrafo successivo, quali sono stati i problemi durante lo sviluppo dell'algoritmo e mostreremo le porzioni di codice implementate per la loro soluzione.

### 4.4.4 Problemi e soluzioni

Una grossa parte di lavoro per lo sviluppo è stata fatta inizialmente per ricercare tutta la documentazione necessaria alla scrittura di codice parallelo. Nel web è presente una quantità enorme di materiale a riguardo, selezionare ciò di cui si ha bisogno in base alle caratteristiche del problema e delle risorse hardware a propria disposizione non è facile e richiede molto tempo.

Dopo aver individuato gli strumenti più adatti al raggiungimento del proprio obiettivo inizia la parte di progettazione dell'algoritmo. Si studia l'algoritmo sequenziale e si cerca di capire se è possibile utilizzare la struttura nota come base di partenza o se sarà necessario riscriverlo da zero. Infine dopo queste fasi si procederà con la scrittura vera e propria del codice con annesse tutte le difficoltà di un linguaggio di programmazione come C. Che è estremamente potente e adattabile a tante situazioni, ma che non avendo molti controlli automatici da parte del linguaggio, costringe il programmatore a doversi occupare di tutto, non potendo tralasciare nessun dettaglio, poiché ogni errore può causare numerosi problemi durante l'esecuzione del programma.

Durante lo sviluppo del nostro algoritmo abbiamo incontrato una serie di difficoltà, vedremo ora quali sono state le principali che hanno richiesto uno studio approfondito del problema e mostriamo il codice che ci ha permesso di risolverli.

#### Creazione *thread* e variabili globali

La prima problematica che abbiamo dovuto affrontare una volta iniziato a implementare il codice era pensare ad un modo per immagazzinare i risultati prodotti dai vari *thread* per poterli mettere insieme al termine dell'esecuzione dell'algoritmo. Le variabili globali utilizzate dall'algoritmo sequenziale erano dichiarate come nella porzione di codice riportata qui sotto, ed era necessario mantenerle per poter poi inserire i risultati finali.

```
// VARIABILI GLOBALI
static double mindev;
```

```
static double *X, *y, *w, *n, *dev, *yval, *yprob;
static Sint *levels, *node, *var, *where, *ordered;
static int nobs, nvar, nmax, minsize, mincut, nnode,
                          Gini;
static char **cutleft, **cutright;
```

Per poter avere una struttura che fosse il più possibile flessibile ed adattabile abbiamo deciso di costruire delle variabili che risultassero dei puntatori ad altri puntatori. In questo modo avremmo potuto vedere l'insieme dei risultati prodotti dai diversi thread come diverse matrici, le quali saranno tutte inizializzate della dimensione massima possibile che conosciamo già dopo l'esecuzione della parte sequenziale.

```
// Variabili da passare ai thread
double *mindevR;
double **XR, **yR, **wR, **nR, **devR, **yvalR, **ydevR,
                *yprobR;
Sint **levelsR, **nodeR, **varR, **whereR, **orderedR;
int *nobsR, *nvarR, *nmaxR, *minsizeR, *mincutR, *nnodeR,
         *GiniR;
char ***cutleftR, ***cutrightR;
int **junkR;
```

Potremmo quindi accedere ai diversi elementi come se dovessimo accedere a delle matrici utilizzando la notazione: `nomeVariabile[nomeThread][indiceElemento]`

Con questo metodo ci mettiamo al riparo dal problema della concorrenza tra *thread*, evitando che nello stesso momento due o più *thread* vogliano accedere o modificare lo stesso indirizzo di memoria. Cosa che succede puntualmente nel caso tutti i *thread* vogliano accedere contemporaneamente alle variabili globali iniziali viste sopra.

### Unione risultati finali

Come abbiamo avuto modo di vedere in precedenza la parallelizzazione permette di ridurre i tempi di calcolo ma, dall'altro lato, aumenta la comunicazione e il numero di accessi alla memoria. Al termine della regione parallela avremmo quindi

i risultati provenienti dai diversi *thread* eseguiti. Il problema è che a questo punto servirà un metodo per ricostruire la forma dell'albero completo con tutti i suoi rami. Ciò è possibile unendo e modificando opportunamente le numerazioni dei nodi e l'ordinamento delle variabili.

Vediamo un piccolo esempio di prova per chiarire il problema.

Dopo l'esecuzione dell'algoritmo abbiamo questi risultati nei vettori contenenti i numeri dei nodi:

Abbiamo un primo vettore che contiene la crescita sequenziale iniziale fino al punto in cui il numero di nodi è uguale al numero di possibili *thread* (variabile node) `node = 1 2 4 5 3 6 7`

I nodi `4,5,6,7` saranno i punti di partenza per le suddivisioni dei futuri thread.

```
NODI OTTENUTI NELLA CRESCITA RICORSIVA
nodeR[0] = 1 2 4 5 3 6 12 13 7 14 28 29 15 30 31
nodeR[1] = 1 2 4 8 9 18 36 37 19 38 39 5 10 11 3
nodeR[2] = 1 2 4 8 9 5 10 20 21 11 3
nodeR[3] = 1 2 3 6 12 13 7
```

Abbiamo bisogno di un modo per rinominare i nodi dentro a `nodeR` per poterli in una fase successiva inserirli in una variabile unica evitando che vi siano dei nomi replicati.

Il codice utilizzato prevede l'utilizzo di alcune proprietà degli alberi binari e consiste nella funzione riportata qui di seguito:

```
for (i=0;i<NUM_THREADS;)
{
 j=0;
 while (nodeR[i][j] != 0)
 {
  int max_pow = pow(2.0,
                (int)my_log((double)nodeR[i][j], 2.0));
  nodeR[i][j]= max_pow*NUM_THREADS - max_pow +
                nodeR[i][j] + max_pow*i;
  j++;
 }
```

```
 i++;
}
```

Come vediamo nei risultati qui sotto ogni *thread* sarà partito da uno dei 4 nodi indicati in precedenza e avrà sviluppato in modo indipendente il ramo a lui relativo.

```
NODI OTTENUTI NELLA CRESCITA RICORSIVA
nodeR[0] 4 8 16 17 9 18 36 37 19 38 76 77 39 78 79
nodeR[1] 5 10 20 40 41 82 164 165 83 166 167 21 42 43 11
nodeR[2] 6 12 24 48 49 25 50 100 101 51 13
nodeR[3] 7 14 15 30 60 61 31
```

Un appunto è necessario per spiegare la numerazione dei nodi. La numerazione arriva a valori grandi a causa del metodo utilizzato per assegnare una numerazione univoca. Con questo metodo infatti i numeri vengono 'riservati' anche nel caso in cui un nodo non venga ulteriormente suddiviso.

Un piccolo esempio può aiutare nella comprensione. Supponiamo di avere un nodo, chiamato 1, che si divide in due. Questi due nodi verranno identificati con la numerazione 2 e 3. Se il nodo 2 non potrà essere ulteriormente diviso ma lo sarà il nodo 3, avremmo che la successiva numerazione dei nodi figli di 3 saranno 6 e 7. Mentre i valori 4 e 5 sono 'riservati' per un eventuale suddivisione del nodo 2 anche se non effettivamente presenti.

Una volta eseguita questa operazione di numerazione univoca dei nodi non resta che ricostruire in una variabile globale l'albero completo correttamente ordinato. Facendo in modo che tutte le variabili collegate siano anch'esse riordinate in modo corretto per mantenere il legame logico tra tutte le variabili.

Riportiamo di seguito il codice della funzione creata a questo scopo.

```
create_complete_tree()
{
 int found=0;
 int i_r=0;
 int j_r=0;
 int k_r=0;

 int node_backup[nmax];
```

```
for (int b=0;b<nmax;b++)
{node_backup[b]=0;}

while (node [i_r]!=0)
{node_backup[i_r]=node[i_r];i_r++;}
i_r=0;

while (i_r<NUM_THREADS)
{

       if (node[k_r]==nodeR[i_r][j_r])
       {
    found=1;
        int s_k=k_r;
        while ((node[s_k])!=0)
        {
               node[s_k+nnodeR[i_r]]=node[s_k+1];
               var[s_k+nnodeR[i_r]]=var[s_k+1];
               dev[s_k+nnodeR[i_r]]=dev[s_k+1];
               yval[s_k+nnodeR[i_r]]=yval[s_k+1];
               n[s_k+nnodeR[i_r]]=n[s_k+1];
               cutleft[s_k+nnodeR[i_r]]=cutleft[s_k+1];
               cutright[s_k+nnodeR[i_r]]=cutright[s_k+1];
               s_k++;
               }
               while (nodeR[i_r][j_r]!=0)
               {
                       node[k_r]=nodeR[i_r][j_r];
                       var[k_r]=varR[i_r][j_r];
                       dev[k_r]=devR[i_r][j_r];
                       yval[k_r]=yvalR[i_r][j_r];
                       n[k_r]=nR[i_r][j_r];
                       cutleft[k_r]=cutleftR[i_r][j_r];
                       cutright[k_r]=cutrightR[i_r][j_r];
```

```
                        k_r++;
                        j_r++;
                }
                i_r++;
                j_r=0;
        }
        else
        {
                k_r++;
        }
}
k_r=0;
for(int j=0;j<NUM_THREADS;j++)
{
        for(int i=0;i<nobsR[j];i++)
        {
                whereR[j][i]=nodeR[j][whereR[j][i]];
                while (node[k_r++]!=whereR[j][i]){};
                whereR[j][i]=k_r;
                k_r=0;
        }
}
int s_z=0;
int i_n=0;
int i_m = 0;

for(int j=NUM_THREADS-1;j>=0;j--)
{
        while (nodeR[j][0]!=node_backup[s_z]) {s_z++;};

        while (i_n<nobs)
        {
                if (where[i_n]==s_z)
                {
                        where[i_n]=whereR[j][i_m];
```

```
                        i_m++;
                }
                i_n++;
        }

        i_n=0;
        i_m=0;
        s_z=0;
 }
}
```

### Comunicazione tra R e C

Un altro problema che abbiamo dovuto affrontare implementando l'algoritmo parallelo del CART è stato quello della comunicazione tra R e C. Il problema non è emerso nella fase di comunicazione vera e propria, poiché R mette a disposizione una serie di strumenti per comunicare con C, ma riguarda il modo in cui vengono utilizzate le matrici nei due differenti linguaggi di programmazione. All'interno del nostro programma ci troviamo a dover utilizzare una matrice X, la quale contiene tutte le variabili esplicative e i relativi valori assunti dalle unità statistiche. In C, le matrici bidimensionali vengono memorizzate utilizzando l'ordine per riga mentre in R vengono riempite utilizzando l'ordine per colonna (Matloff, 2011). Per esempio:

```
C (matrice 3x4)
1  2  3  4
5  6  7  8
9 10 11 12

R (matrice 3x4)
1  4  7  10
2  5  8  11
3  6  9  12
```

Ciò significa che nel momento in cui si riempie la matrice con i valori in R verrà prima riempita la prima colonna e poi si inizierà con la seconda colonna mentre in

C si riempirà la prima riga e una volta terminata si passerà alla seconda. Nella comunicazione tra R e C le variabili sono scambiate utilizzando dei puntatori e non sono copiate da una parte all'altra. È quindi necessario creare una funzione apposita per riempire le matrici nello stesso modo. Questa funzione è stata implementata in C utilizzando una serie di cicli che permettono di riempire la matrice nel modo desiderato. Nonostante non sia completamente ottimizzata riesce a svolgere il proprio compito in tempi accettabili anche con discrete quantità di dati. Nel caso di sviluppi futuri un possibile obiettivo potrebbe sicuramente essere l'ottimizzazione di questa funzione.

# Capitolo 5

# Valutazione performance

Arrivati a questo punto andiamo a valutare le prestazioni su alcuni dataset per verificare la correttezza dell'implementazione dell'algoritmo e vedere se è possibile trarre un reale vantaggio in termini di tempi di esecuzione dalla parallelizzazione.

Andremo a confrontare l'algoritmo implementato eseguendo sugli stessi dati prima l'algoritmo sequenziale (un solo *thread*), e successivamente eseguiremo la crescita dell'albero utilizzando prima due e poi quattro *thread*.

Nel Paragrafo 5.1, utilizzeremo un piccolo dataset per verificare non tanto le performance dell'algoritmo, poichè immaginiamo che su dataset piccoli l'*overhead* di creazione e gestione dei *thread* non conduca a miglioramenti nelle prestazioni, ma per verificare se l'algoritmo suddivide correttamente le osservazioni. Andremo così a confrontare dal punto di vista grafico se l'algoritmo sequenziale implementato nella libreria `tree` porta alle stesse suddivisioni dell'algoritmo parallelo. Nel Paragrafo 5.2 dopo aver descritto brevemente i dataset a cui faremo riferimento andremo a confrontare i tempi di esecuzione dell'algoritmo e andremo a calcolare gli indici di prestazione.

Abbiamo utilizzato per queste analisi dei dataset di dimensioni non particolarmente grandi poiché nel caso in cui il nostro algoritmo fosse riuscito ad ottenere dei risultati validi questi potrebbero essere facilmente riconducibili a dataset di dimensioni maggiori. Infatti, proprio per come è costruito l'algoritmo, i tempi di chiamata a sistema per la creazione dei thread, la suddivisione del problema e infine l'unione

dei risultati può portare ad un aumento dei tempi di esecuzione rispetto all'algoritmo sequenziale, se ciò non accade per dataset piccoli dove il costo computazionale non è molto alto possiamo avere delle buone garanzie sul fatto che anche con dataset di dimensioni maggiori siamo in grado di ottenere prestazioni migliori.

Dopo aver descritto i dataset e svolto le analisi nei Paragrafi 5.2.1 e 5.2.2, nel Paragrafo 5.2.3 commenteremo l'algoritmo alla luce di tutti i risultati ottenuti.

Le analisi e le esecuzioni dell'algoritmo sono state eseguite su un computer dotato di un processore multi-core `Intel Core i3 CPU M 330  2.13 GHz`×4 con 4 GB di memoria RAM. Il sistema operativo presente è ubuntu 12.04 LTS versione a 32 bit. La versione di R utilizzata è la versione 2.14.1.

## 5.1 Confronto grafico

I primi dati che andiamo a prendere in considerazione riguardano delle misure di performance e le caratteristiche di 209 CPU. Questi dati si possono trovare all'interno della libreria `MASS` in R. Fonte: (Ein-Dor e Feldmesser, 1987)

Il dataset è composto dalle seguenti variabili:

```
name: Nome del produttore e del modello
syct: Tempo di ciclo in nanosecondi
mmin: Minima memoria principale in kilobyte
mmax: Massima memoria principale in kilobyte
cach: Dimensione della cache in kilobyte
chmin: Numero minimo di canali
chmax: Numero massimo di canali
perf: Prestazioni pubblicate di un benchmark rispetto a un IBM 370/158-3
estperf: Prestazioni stimate (da Ein-Dor & Feldmesser)
```

In questo dataset vogliamo capire come sono collegate alla variabile risposta `perf` le caratteristiche e le prestazioni dei vari processori. Si tratta di un piccolo dataset che però ci permette di capire in modo veloce se l'algoritmo sta effettivamente suddividendo in modo corretto le osservazioni con il metodo da noi implementato.

### 5.1.1 1 THREAD (Algoritmo sequenziale)

Per la crescita dell'albero abbiamo passato alla funzione oltre ai dati su cui eseguire l'algoritmo, la formula che lega la variabile risposta alle variabili esplicative è una formula del tipo:

```
log10(perf) ~ syct+mmin+mmax+cach+chmin+chmax
```

I parametri per la funzione tree.control() per la crescita dell'albero utilizza i valori di default dei parametri `minsize=10`, `mincut=10` e `mindev=0.01`. Il risultato dell'albero sequenziale implementato all'interno della libreria `tree` ci porta a dei risultati di questo tipo.

```
node), split, n, deviance, yval
      * denotes terminal node

 1) root 209 43.12000 1.753
   2) cach < 27 143 11.79000 1.525
     4) mmax < 6100 78  3.89400 1.375
       8) mmax < 1750 12  0.78430 1.089 *
       9) mmax > 1750 66  1.94900 1.427 *
     5) mmax > 6100 65  4.04500 1.704
      10) syct < 360 58  2.50100 1.756
        20) chmin < 5.5 46  1.22600 1.699 *
        21) chmin > 5.5 12  0.55070 1.974 *
      11) syct > 360 7  0.12910 1.280 *
   3) cach > 27 66  7.64300 2.249
     6) mmax < 28000 41  2.34100 2.062
      12) cach < 96.5 34  1.59200 2.008
        24) mmax < 11240 14  0.42460 1.827 *
        25) mmax > 11240 20  0.38340 2.135 *
      13) cach > 96.5 7  0.17170 2.324 *
     7) mmax > 28000 25  1.52300 2.555
      14) cach < 56 7  0.06929 2.268 *
      15) cach > 56 18  0.65350 2.667 *
```

Nella Figura 5.1 possiamo vedere graficamente l'albero nel suo completo sviluppo.

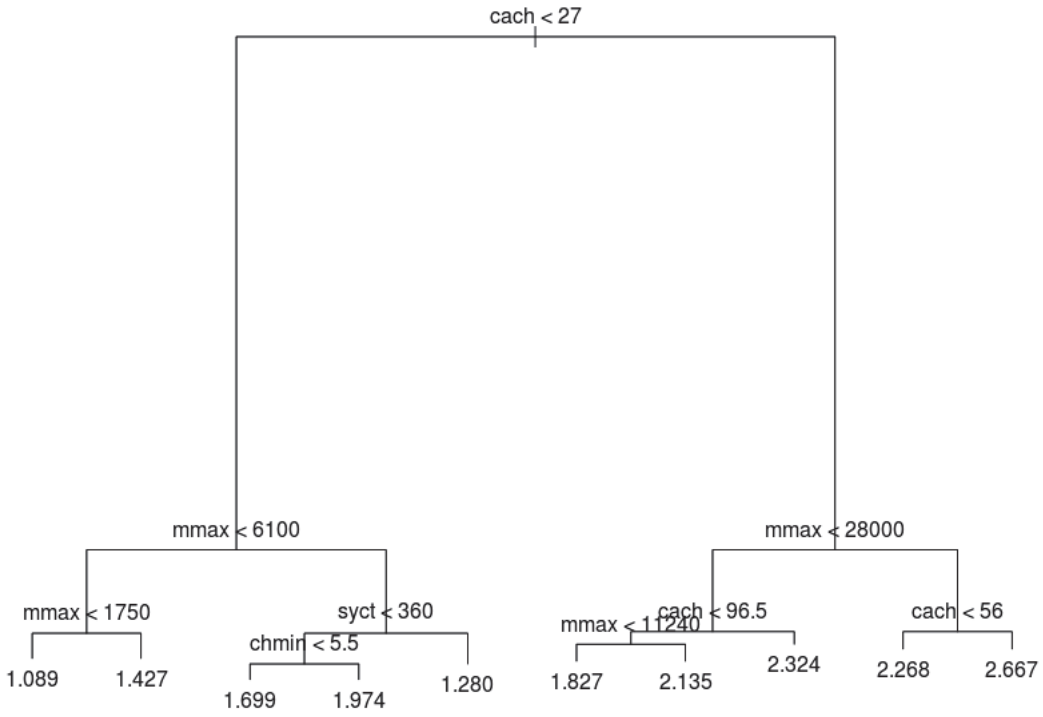


Figura 5.1. Albero seriale costruito con i dati sulle CPU

Vediamo ora i tempi di esecuzione calcolati utilizzando la funzione `system.time()` di R:

```
   user  system elapsed
  0.008   0.000   0.008
```

La funzione `system.time()` permette di calcolare il tempo di utilizzo della CPU da parte della funzione. All'intero di `system.time()` viene richiamata la funzione `proc.time()`. Questa funzione determina quanto è il tempo reale di utilizzo della CPU da parte del processo in esecuzione attualmente. La funzione ritorna tre valori,

i primi due rappresentano lo *user time* e il *system CPU time* del processo R corrente e tutti i processi figli. Il terzo elemento rappresenta il tempo reale trascorso da quando il processo è stato avviato.

Lo *user time* è il tempo impiegato dalla CPU per l'esecuzione delle istruzioni dell'utente per richiamare il processo. Il *system time* è il tempo impiegato dalla CPU per l'esecuzione da parte del sistema per processo richiamato.

Come vediamo, date le piccole dimensioni del dataset, i tempi di esecuzione sono molto bassi. Ciò che ci aspetteremo sarà che l'algoritmo parallelo a causa del tempo impiegato per avviare i diversi *thread*, il numero maggiore di accessi alla memoria, l'unione dei risultati e altre operazioni, non porti grandi vantaggi con dataset così piccoli, ma che addirittura ne rallenti l'esecuzione.

### 5.1.2 2 THREAD

Risultati della crescita dell'albero:

```
node), split, n, deviance, yval
      * denotes terminal node

 1) root 209 43.12000 1.7530
   2) cach < 27 143 11.79000 1.5250
     4) mmax < 6100 78  3.89400 1.3750
       8) mmax < 1750 12  0.78430 1.0890
        16) chmax < 3.5 7  0.19260 0.9741 *
        17) chmax > 3.5 5  0.37090 1.2490 *
       9) mmax > 1750 66  1.94900 1.4270
        18) mmax < 2500 17  0.56680 1.3250 *
        19) mmax > 2500 49  1.14600 1.4620
          38) chmax < 4.5 14  0.35280 1.3550 *
          39) chmax > 4.5 35  0.56750 1.5050 *
     5) mmax > 6100 65  4.04500 1.7040
      10) syct < 360 58  2.50100 1.7560
        20) chmin < 5.5 46  1.22600 1.6990
          40) cach < 0.5 11  0.20210 1.5310 *
```

```
          41) cach > 0.5 35  0.61630 1.7510 *
        21) chmin > 5.5 12  0.55070 1.9740
          42) mmin < 3550 7  0.11510 1.8560 *
          43) mmin > 3550 5  0.19780 2.1410 *
      11) syct > 360 7  0.12910 1.2800 *
   3) cach > 27 66  7.64300 2.2490
     6) mmax < 28000 41  2.34100 2.0620
      12) cach < 96.5 34  1.59200 2.0080
        24) mmax < 11240 14  0.42460 1.8270 *
        25) mmax > 11240 20  0.38340 2.1350
          50) chmax < 14 10  0.07836 2.0380 *
          51) chmax > 14 10  0.11540 2.2330 *
      13) cach > 96.5 7  0.17170 2.3240 *
     7) mmax > 28000 25  1.52300 2.5550
      14) cach < 56 7  0.06929 2.2680 *
      15) cach > 56 18  0.65350 2.6670
        30) chmax < 48 13  0.16790 2.6030
          60) mmin < 12000 6  0.03069 2.5050 *
          61) mmin > 12000 7  0.03014 2.6870 *
        31) chmax > 48 5  0.29360 2.8330 *
```

Nella Figura 5.2 possiamo vedere il grafico completo dell'esecuzione con due *thread*.

I tempi di esecuzione ottenuti dall'algoritmo sono:

```
   user  system elapsed
  0.012   0.000   0.007
```

### 5.1.3 4 THREAD

Risultati della crescita dell'albero:

```
node), split, n, deviance, yval
      * denotes terminal node
```

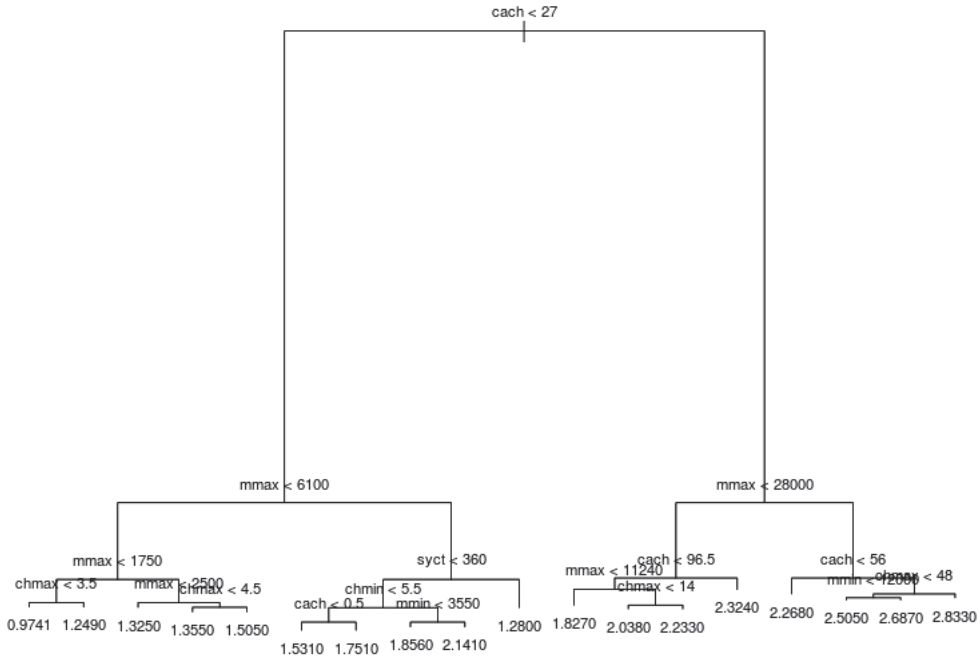


Figura 5.2. Albero costruito con i dati sulle CPU utilizzando 2 thread

```
1) root 209 43.120000 1.7530
  2) cach < 27 143 11.790000 1.5250
    4) mmax < 6100 78  3.894000 1.3750
      8) mmax < 1750 12  0.784300 1.0890
       16) chmax < 3.5 7  0.192600 0.9741 *
       17) chmax > 3.5 5  0.370900 1.2490 *
      9) mmax > 1750 66  1.949000 1.4270
       18) mmax < 2500 17  0.566800 1.3250
         36) mmin < 518 7  0.345900 1.4030 *
         37) mmin > 518 10  0.148200 1.2710 *
       19) mmax > 2500 49  1.146000 1.4620
         38) chmax < 4.5 14  0.352800 1.3550
           76) syct < 190 5  0.072220 1.2440 *
           77) syct > 190 9  0.184400 1.4170 *
         39) chmax > 4.5 35  0.567500 1.5050
           78) syct < 80.5 5  0.036060 1.3720 *
           79) syct > 80.5 30  0.428700 1.5270 *
    5) mmax > 6100 65  4.045000 1.7040
     10) syct < 360 58  2.501000 1.7560
       20) chmin < 5.5 46  1.226000 1.6990
         40) cach < 0.5 11  0.202100 1.5310 *
         41) cach > 0.5 35  0.616300 1.7510
           82) chmin < 1.5 20  0.257500 1.7970
            164) mmax < 14000 13  0.087620 1.7570 *
            165) mmax > 14000 7  0.109000 1.8720 *
           83) chmin > 1.5 15  0.260900 1.6900
            166) chmax < 9 8  0.133800 1.6310 *
            167) chmax > 9 7  0.066010 1.7590 *
       21) chmin > 5.5 12  0.550700 1.9740
         42) mmin < 3550 7  0.115100 1.8560 *
         43) mmin > 3550 5  0.197800 2.1410 *
     11) syct > 360 7  0.129100 1.2800 *
  3) cach > 27 66  7.643000 2.2490
```

```
    6) mmax < 28000 41  2.341000 2.0620
     12) cach < 96.5 34  1.592000 2.0080
       24) mmax < 11240 14  0.424600 1.8270
         48) chmax < 26 9  0.219800 1.7750 *
         49) chmax > 26 5  0.138500 1.9190 *
       25) mmax > 11240 20  0.383400 2.1350
         50) chmax < 14 10  0.078360 2.0380
          100) chmax < 9 5  0.040400 1.9820 *
          101) chmax > 9 5  0.007291 2.0930 *
         51) chmax > 14 10  0.115400 2.2330 *
     13) cach > 96.5 7  0.171700 2.3240 *
    7) mmax > 28000 25  1.523000 2.5550
     14) cach < 56 7  0.069290 2.2680 *
     15) cach > 56 18  0.653500 2.6670
       30) chmax < 48 13  0.167900 2.6030
         60) mmin < 12000 6  0.030690 2.5050 *
         61) mmin > 12000 7  0.030140 2.6870 *
       31) chmax > 48 5  0.293600 2.8330 *
```

Nella Figura 5.3 possiamo vedere il grafico dell'albero alla fine della crescita:
Tempi di esecuzione:

```
  user  system elapsed
 0.032   0.004   0.016
```

### 5.1.4 Osservazioni

Dal confronto grafico effettuato in questo paragrafo è possibile notare alcune particolarità. La prima cosa che possiamo osservare è che l'algoritmo parallelo è stato costruito in modo da avere lo stesso output dell'algoritmo già implementato in R. Questo è un grosso vantaggio poiché ci permette di utilizzare tutte le funzioni di analisi già implementate per l'algoritmo sequenziale.

Una seconda cosa che è possibile notare è la differente profondità dell'albero. Nella 5.1 possiamo osservare che i livelli di suddivisione delle osservazioni sono quattro

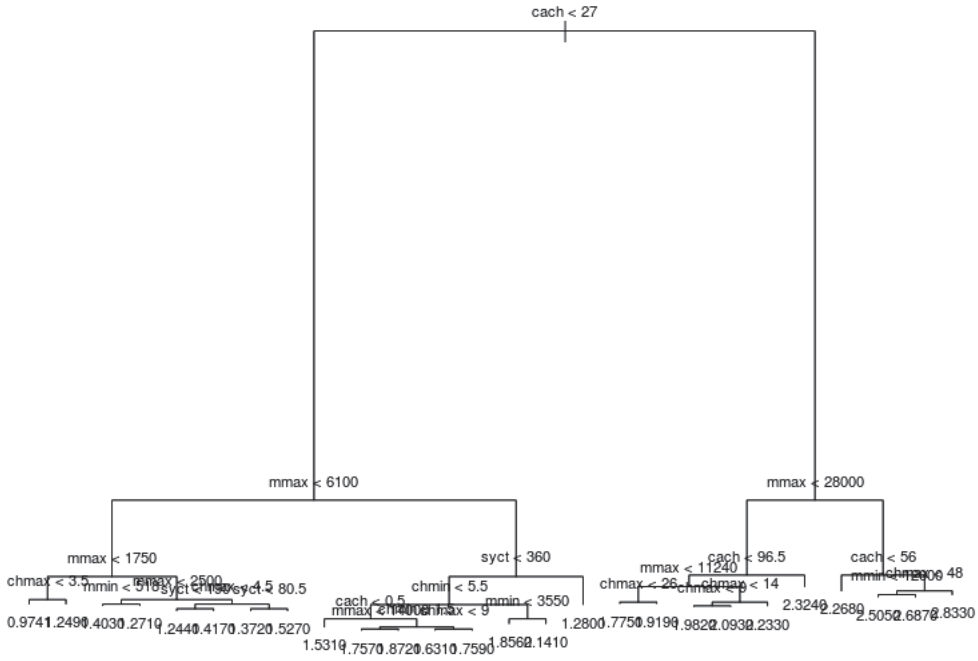


Figura 5.3. Albero costruito con i dati sulle CPU utilizzando 4 thread

mentre per gli altri grafici 5.2 e 5.3 questi livelli aumentano, facendo andare l'albero più in profondità. Questo è dovuto al fatto che nell'algoritmo parallelo, dopo la prima suddivisione, ovvero quella che ci porta ad avere tanti nodi quanti sono i *thread*, dentro ad ogni nodo verrà applicato nuovamente da zero l'algoritmo ricorsivo per la crescita. In sostanza è come ricominciare la crescita di un nuovo albero non tenendo conto dei livelli superiori che sono già stati costruiti. Ciò ci permette, utilizzando la stessa soglia di devianza, di suddividere in modo più fine le osservazioni. Starà allo statistico poi decidere se queste suddivisioni più fini sono o meno utili ai fini dell'analisi che si deve effettuare.

## 5.2 Confronto tempi di esecuzione

Applichiamo ora gli algoritmi illustrati nel paragrafo precedente ad alcuni dataset per valutare le performance.

Dobbiamo tenere presente che i tempi riportati in seguito sono il frutto di una media di più esecuzioni. Infatti, poiché le chiamate a sistema, la creazione di *thread*, ecc. dipendono dal livello di utilizzo in quel momento delle risorse *hardware*, prendere un singolo valore non sarebbe stato veritiero. Abbiamo quindi eseguito più volte l'algoritmo e preso i tempi come una media aritmetica delle varie esecuzioni.

### 5.2.1 Dataset su dati di connessione UMTS

I dati si riferiscono ad una società di telecomunicazioni cellulari, la quale è interessata ad analizzare alcuni dati che ha a disposizione con lo scopo di conoscere meglio le caratteristiche dei suoi clienti. La società oltre al normale servizio telefonico fornisce una serie di prodotti aggiuntivi di cui i clienti in possesso di cellulari UMTS possono usufruire a pagamento.

La direzione marketing è interessata ad acquisire una conoscenza sulle caratteristiche dei clienti che utilizzano il servizio di 'navigazione' su internet da telefonino. A puro titolo di esemplificazione, un aspetto di questa conoscenza può riguardare un'indicazione circa il fatto che navigano di più coloro che anche telefonano di più o se invece la navigazione su internet 'sostituisce' in qualche modo la telefonata.

Oppure può essere utile vedere se chi naviga molto utilizza anche altri servizi specifici dei telefono UMTS forniti dalla società di telefonia, quali le videochiamate o i videomessaggi.

La società è anche interessata a prevedere per ciascun cliente quali caratteristiche lo portano ad utilizzare più comunemente questo servizio.

Il dataset contiene le informazioni di 9662 clienti e le variabili disponibili per l'analisi sono:

| | |
|---|---|
| BROWSINGEv | Numero di connessioni alla rete internet |
| EMAILEv | Numero di eventi di email (email inviate o ricevute) da cellulare |
| EMAILKili | Quantità di trasmissione (in kilobyte) nell'attività di emailing da cellulare |
| EMAILVal | Valore (in euro) dell'attività di emailing da cellulare |
| MMSEv | Numero di MMS inviati |
| MMSKili | Quantità di trasmissione (in kilobyte) inviando MMS |
| MMSVal | Valore (in euro) dell'invio di MMS |
| NEWSEv | Numero di eventi di consultazione di News |
| NEWSVal | Valore (in euro) dell'attività di consultazione di News |
| PORNOEv | Numero di immagini/video pornografiche scaricate |
| PORNOVal | Valore (in euro) dell'attività di scaricamento di materiale pornografico |
| SHOWBIZVal | Valore della visita a siti di business |
| SMSEv | Numero di SMS inviati |
| SMSVal | Valore (in euro) dell'invio di SMS |
| STADIOLIVEv | Numero di accessi alle immagini dal vivo legate allo sport |
| STADIOLIVEVal | Valore (in euro) degli accessi alle immagini dal vivo legate allo sport |
| VIDEOCALLEv | Numero di videochiamate effettuate |
| VIDEOCALLMin | Durata delle videochiamate effettuate |
| VIDEOCALLVal | Valore (in euro) delle videochiamate effettuate |
| VOICECALLEv | Numero delle chiamate-voce effettuate |
| VOICECALLMin | Durata delle chiamate-voce effettuate |
| VOICECALLVal | Valore (in euro) delle chiamate-voce effettuate |

`Piano.Tariffario  Piano tariffario con due modalità`

Dal Paragrafo 5.1 abbiamo visto, e lo confermano anche in questo caso, che gli algoritmi suddividono nello stesso modo i dati, evitiamo perciò di riportare grafici e risultati che possono essere facilmente estratti da una rapida esecuzione in R, ma ci limitiamo a fornire i tempi di esecuzione degli algoritmi.

Riportiamo ora i tempi di esecuzione calcolati con la funzione `system.time()` dei diversi algoritmi e calcoliamo gli indici di *speedup* ed *efficienza* delle versioni parallele:

**1 Thread**

```
   user  system elapsed
  0.256   0.004   0.288
```

**2 Thread**

```
   user  system elapsed
  0.176   0.004   0.133
```

*Speedup:*

$$S_2 = \frac{T_1}{T_2} = \frac{0.256}{0.176} = 1.454 \tag{5.1}$$

*Efficienza:*

$$E_2 = \frac{S_2}{2} = \frac{1.454}{2} = 0.727 \tag{5.2}$$

**4 Thread**

```
   user  system elapsed
  0.216   0.004   0.149
```

*Speedup:*

$$S_4 = \frac{T_1}{T_4} = \frac{0.256}{0.216} = 1.185 \tag{5.3}$$

*Efficienza:*

$$E_4 = \frac{S_4}{4} = \frac{1.185}{4} = 0.296 \tag{5.4}$$

Gli indici di *speedup* ed efficienza permettono, attraverso delle semplici operazioni, di capire se il nostro algoritmo sta ottenendo o meno i risultati sperati. Come

abbiamo visto nel Paragrafo 1.6.1, lo *speedup* permette di confrontare in modo diretto i tempi di esecuzione dell'algoritmo sequenziale e parallelo. Nel nostro caso, abbiamo ottenuto per entrambe le implementazioni valori maggiori di 1, ma minori al numero di processori $p$. Il caso ottimale sarebbe che il valore dello *speedup* sia uguale a $p$, mentre nel caso in cui questi indici superino il valore di $p$, siamo in presenza di effetti di *cache* e non di effettivi miglioramenti dell'algoritmo. Vediamo quindi che il nostro algoritmo ci permette di ridurre i tempi di calcolo anche se non è vicino al caso ideale.

### 5.2.2 Dataset su dati di sottoscrizione di un servizio telefonico

Questo dataset contiene informazioni su 58712 clienti che hanno sottoscritto un contratto con una compagnia telefonica. Alcuni di questi clienti hanno chiuso, per loro volontà, la relazione con la società telefonica, mentre altri risultano ancora clienti. Utilizzando questi dati, la società è interessata a studiare, per i clienti che hanno abbandonato il servizio, le caratteristiche di quelli che sono rimasti clienti per poco tempo rispetto a coloro che hanno abbandonato il servizio dopo un periodo più lungo. Sono disponibili inoltre alcune variabili che hanno caratterizzato l'inizio del rapporto tra il cliente e la società:

```
customer_id: identiticativo del cliente,
rate_plan:   il piano tariffario scelto dal cliente,
fee:         il costo mensile dell'abbonamento sottoscritto,
market:      la città dove il cliente vive,
channel:     il canale di vendita a cui il cliente si è rivolto per l'acquisto,
start_date:  data di inizio del contratto (anno-mese-giorno),
stop_date:   data di rescissione del contratto (anno-mese-giorno),
tenure:      differenza in giorni tra stop_date e start_date,
active:      stato di cliente attivo (active=1) o meno (active=0),
weekday:     giorno della settimana  in cui è stato attivato il contratto,
month:       mese in cui è stato attivato il contratto.
```

In base alle richieste del problema quindi andremo ad utilizzare la variabile `tenure` come variabile risposta e le altre (ad eccezione dell'identificativo del cliente) come variabili esplicative.

Riportiamo ora i tempi di esecuzione calcolati con la funzione `system.time()` dei diversi algoritmi e calcoliamo gli indici di *speedup* ed *efficienza* delle versioni parallele:

**1 Thread**

```
   user  system elapsed
  0.360   0.004   0.366
```

**2 Thread**

```
   user  system elapsed
  0.252   0.004   0.239
```

*Speedup:*

$$S_2 = \frac{T_1}{T_2} = \frac{0.360}{0.252} = 1.428 \tag{5.5}$$

*Efficienza:*

$$E_2 = \frac{S_2}{2} = \frac{1.428}{2} = 0.714 \tag{5.6}$$

**4 Thread**

```
   user  system elapsed
  0.244   0.016   0.272
```

*Speedup:*

$$S_4 = \frac{T_1}{T_4} = \frac{0.360}{0.244} = 1.475 \tag{5.7}$$

*Efficienza:*

$$E_4 = \frac{S_4}{4} = \frac{1.475}{4} = 0.368 \tag{5.8}$$

Dagli indici si può notare come vi sia un effettivo miglioramento delle prestazioni durante l'esecuzione, anche se questo miglioramento è lontano dal caso ideale di *speedup* lineare.

### 5.2.3 Osservazioni

Innanzitutto per avere un quadro completo dei risultati ottenuti andiamo a riportare i tempi di esecuzione e gli indici di prestazione che abbiamo calcolato. Nelle tabelle 5.1, 5.2 e 5.3 abbiamo riportato i tempi di esecuzioni dell'algoritmo utilizzando i tre dataset presi in considerazione per le analisi. Successivamente nelle tabelle 5.4, 5.5 e 5.6 abbiamo riportato gli indici di prestazione calcolati rispetto ai tempi di esecuzione medi.

| Dati CPU | user | system | elapsed |
|---|---|---|---|
| 1 Thread | 0.008 | 0.000 | 0.008 |
| 2 Thread | 0.012 | 0.000 | 0.008 |
| 4 Thread | 0.032 | 0.004 | 0.016 |

Tabella 5.1. Tempi di esecuzione sui dati relativi alle CPU

| Dati connessioni | user | system | elapsed |
|---|---|---|---|
| 1 Thread | 0.256 | 0.004 | 0.288 |
| 2 Thread | 0.176 | 0.004 | 0.133 |
| 4 Thread | 0.216 | 0.004 | 0.149 |

Tabella 5.2. Tempi di esecuzione sui dati relativi alle connessioni

| Dati servizi | user | system | elapsed |
|---|---|---|---|
| 1 Thread | 0.360 | 0.004 | 0.366 |
| 2 Thread | 0.252 | 0.004 | 0.239 |
| 4 Thread | 0.244 | 0.016 | 0.272 |

Tabella 5.3. Tempi di esecuzione sui dati relativi ai servizi

Dalle tabelle precedenti possiamo notare come quando andiamo ad utilizzare i dati sulle CPU non otteniamo miglioramenti con gli algoritmi paralleli, anzi otteniamo un discreto peggioramento. Ciò è dovuto al fatto che lavorando su un dataset molto piccolo il costo di creazione dei *thread*, suddivisione del lavoro e infine unione dei risultati non riesce a creare un vantaggio rispetto ad una esecuzione sequenziale. Negli altri due dataset invece possiamo osservare dei miglioramenti nei tempi di

| Dati CPU | *speedup* | efficienza |
|---|---|---|
| 1 Thread | - | - |
| 2 Thread | 0.667 | 0.333 |
| 4 Thread | 0.250 | 0.063 |

Tabella 5.4. Speedup ed efficienza calcolati sui dati relativi alle CPU

| Dati connessioni | *speedup* | efficienza |
|---|---|---|
| 1 Thread | - | - |
| 2 Thread | 1.454 | 0.727 |
| 4 Thread | 1.185 | 0.296 |

Tabella 5.5. Speedup ed efficienza calcolati sui dati relativi alle connessioni

esecuzione. Possiamo osservare una cosa particolare che accade nel dataset relativo ai dati sulle connessioni. Nella tabella 5.2 infatti, osserviamo che i tempi di esecuzione dell'algoritmo con 4 *thread* risultano maggiori rispetto a quello con 2 *thread*, anche se entrambi migliorano le prestazioni rispetto all'algoritmo sequenziale. Ciò è probabilmente dovuto al fatto che il dataset è notevolmente più grande rispetto al dataset sulle CPU (circa 9000 unità contro le sole 200 del primo) ma evidentemente ancora non abbastanza grande per trarre grossi benefici dall'utilizzo di molti *thread*.

L'ultimo dataset fa capire che la direzione in cui ci stiamo muovendo è corretta, nonostante il dataset non sia di dimensioni enormi (circa 55000 unità statistiche) vediamo un progressivo miglioramento all'aumentare del numero di *thread* e ciò si spinge a credere che la strada seguita sia effettivamente valida.

Un discorso a sé meritano i valori di *speedup* ed efficienza che sono stati calcolati. Tralasciando i valori calcolati sul primo dataset, vediamo come in riferimento al Paragrafo 1.6.1 i valori di *speedup* siano minori di $p$, ovvero del numero di *thread*

| Dati servizi | *speedup* | efficienza |
|---|---|---|
| 1 Thread | - | - |
| 2 Thread | 1.428 | 0.714 |
| 4 Thread | 1.475 | 0.368 |

Tabella 5.6. Speedup ed efficienza calcolati sui dati relativi ai servizi

utilizzati, ciò ci porta a dire che non ci troviamo di fronte ad uno *speedup* ideale ma che il miglioramento delle prestazioni c'è effettivamente stato.

I valori di efficienza, ci portano anch'essi alla conclusione di trovarci in un tipico caso reale con un miglioramento delle prestazioni anche se non il massimo miglioramento ottenibile, questo principalmente per il caso di 2 *thread*. Nel caso in cui si utilizzino 4 *thread* abbiamo un abbassamento di questo indice. La motivazione di questo calo di prestazione potrebbe essere attribuita alla possibilità di ottimizzare il codice maggiormente, eliminando cicli di calcolo che vengono eseguiti provando a seguire la strada della vettorizzazione del codice.

# Conclusioni

In questa tesi abbiamo avuto modo di vedere quali sono alcuni dei problemi che stanno emergendo nel mondo del *data mining*, sopratutto quelli legati alla sempre maggiore numerosità campionaria dei dataset da analizzare. I metodi attualmente disponibili per le analisi risultano comunque validi, ma i tempi di esecuzione, all'aumentare delle dimensioni dei dataset, iniziano ad essere un problema. Una possibile soluzione a questo problema comprende l'utilizzo del calcolo parallelo.

Le due strategie principali per sfruttare il calcolo parallelo sono: utilizzare l'algebra delle matrici per distribuire su più processori il costo computazionale dell'algoritmo oppure utilizzare il calcolo parallelo per ottenere un algoritmo parallelo che permette di suddividere il lavoro e svolgerlo in contemporanea su più processori o macchine.

Poiché molte implementazioni della prima strategia sono già comprese in molti software attualmente disponibili, abbiamo deciso di seguire la seconda strategia, il cui obiettivo è quello di cercare di ridurre i tempi di esecuzione degli algoritmi sfruttando queste nuove tecnologie, in particolare computer multi-processore o multi-core.

Uno degli obiettivi della tesi era quello di creare un algoritmo che permettesse di sfruttare l'hardware che più facilmente si può incontrare nella maggior parte degli ambienti di lavoro e di ricerca, ovvero i computer con processore multi-core. Nel Capitolo 4 abbiamo implementato l'algoritmo di crescita del CART, uno degli algoritmi più utilizzati nelle analisi di *data mining* che comunemente vengono svolte.

Abbiamo visto nel Paragrafo 4.3 che l'algoritmo prevede una iniziale crescita sequenziale dell'albero fino ad arrivare ad un determinato numero di nodi, una volta terminata questa prima crescita ogni nodo viene eseguito su un *thread* diverso facendo in modo che l'albero cresca in simultanea, riducendo in questo modo i tempi

di esecuzione.

Dalle esecuzioni su alcuni dataset diversi, siamo arrivati a concludere in un primo momento che le modifiche all'algoritmo permettevano di raggiungere una suddivisione analoga a quella dell'algoritmo sequenziale, dandoci la conferma che le operazioni condotte all'interno dell'algoritmo fossero esatte. Abbiamo infatti visto che fino ad una certo punto i nodi degli alberi venivano suddivisi nello stesso modo, utilizzando le stesse variabili. Una particolarità del nostro algoritmo parallelo è la possibilità di andare maggiormente in profondità rispetto all'algoritmo sequenziale. Questo perché una volta che l'algoritmo sequenziale viene interrotto e inizia la regione parallela ogni nodo verrà considerato come se fosse un nodo radice di un nuovo albero, non avendo più traccia del numero di suddivisioni effettuate in precedenza, l'algoritmo potrà scendere più in profondità nella suddivisione delle osservazioni. Questa particolarità, nel caso non risulti utile, può tranquillamente essere non sfruttata attraverso una opportuna parametrizzazione della funzione di `R`.

Successivamente, utilizzando altri due dataset di dimensioni maggiori, abbiamo accertato, confrontando tempi di esecuzione e calcolando indici di prestazione, che l'algoritmo parallelo permetteva effettivamente di ottenere dei vantaggi in termini di tempi di esecuzione.

Su entrambi i dataset abbiamo ottenuto uno *speedup* di circa 1.4 sia per le esecuzioni utilizzando due *thread*, sia per le esecuzioni sfruttando quattro *thread*. Se lo *speedup* ottenuto dall'algoritmo ha dato riscontro positivo al lavoro svolto, lo stesso non si può dire riguardo l'indice di efficienza calcolato su entrambi i dataset.

Nel momento in cui eseguiamo l'algoritmo con due *thread* otteniamo un valore di efficienza vicino ad 1 (massima efficienza ottenibile) ma, nel momento in cui eseguiamo l'algoritmo con quattro *thread* questo indice di efficienza cala anche se resta comunque positivo.

Ciò potrebbe essere dovuto ad una mancata ottimizzazione del codice, in particolare nella parte dedicata alla differente modalità di riempimento tra R e C e la non ottima gestione della memoria che l'algoritmo esegue. Come vedremo nel paragrafo successivo, un possibile sviluppo della tesi consiste nello studio di operazioni di vettorizzazione in alcuni punti del codice per migliorare l'efficienza totale all'aumentare del numero di *thread* utilizzati.

# Sviluppi futuri

L'idea, alla base di questo lavoro, è stata quella di ottenere algoritmi che sfruttassero le tecnologie attuali per migliorare le loro prestazioni e diminuissero i tempi di esecuzione. L'algoritmo descritto e implementato in questa tesi di laurea è aperto a tutta una serie di sviluppi futuri che consentano di migliorarlo ed aumentarne le performance.

Una prima direzione, in cui si potrebbe andare a lavorare, è quella di ottimizzare il codice in C per la parallelizzazione e l'unione dei risultati. Questa strada richiede che vi sia da parte del programmatore una conoscenza molto approfondita del linguaggio di programmazione C e che sia perfettamente a conoscenza di come ogni istruzione utilizzi la memoria per poter ridurre al minimo gli accessi ad essa, e ridurre il tempo necessario per la comunicazione. La gestione della memoria infatti non è stata ottimizzata al meglio, una delle questioni su cui intervenire è quella della duplicazione delle variabili che devono essere passate alla parte ricorsiva. Operazione questa non banale ma che porterebbe ad un utilizzo di memoria inferiore rispetto all'implementazione attuale. Legata a questa forma di ottimizzazione vi è anche la parte dedicata alla generalizzazione dell'algoritmo. Essendo partiti dall'algoritmo sequenziale già implementato, le modifiche necessarie per questa generalizzazione non saranno particolarmente invasive.

Altra strada che potrebbe essere percorsa riguarda più una modifica nella logica dell'algoritmo. Infatti, l'algoritmo è stato sviluppato per poter sfruttare al meglio i sistemi multiprocessore che equipaggiano la maggior parte dei computer attuali. Un'estensione di questo concetto è quello di andare a sfruttare dei cluster di computer. Come visto nel Capitolo 1, l'utilizzo di sistemi a memoria distribuita da un lato aumenta la difficoltà nella programmazione ma dall'altro introduce una serie di vantaggi quali, il possibile aumento a dismisura della memoria e la scalabilità

dell'algoritmo.

Gli sviluppi futuri potrebbero portare inevitabilmente all'applicazione di questa logica a tutti quegli algoritmi che in qualche modo permettono la suddivisione dei dati al loro interno e lo sviluppo simultaneo di più percorsi all'interno dell'algoritmo.

Infine, ricordiamo che nel momento in cui gli algoritmi fossero perfettamente ottimizzati e generalizzati per poter lavorare in maniera efficace ed efficiente su gran parte delle macchine, la costruzione di un pacchetto R da rendere disponibile alla comunità potrebbe essere la giusta conclusione del lavoro.

# Appendice A

# Codice R e C

## A.1 Codice R

### A.1.1 Funzione `treeParallel()`

```
treeParallel <-
function(formula, data, weights, subset,
         na.action = na.pass, control = tree.control(nobs, ...),
         method = "recursive.partition",
         split = c("deviance", "gini"),
         model = FALSE, x = FALSE, y = TRUE, wts = TRUE, ...)

{
# Se il model è un dataframe, gli argomenti formula e data sono ignorati
# model è utilizzato per definire il modello
    if (is.data.frame(model))
    {
# se il modello è già un data frame lo assegno a m
     m <- model
     model <- FALSE
    }
    else
    {
```

```
 # Ritorna una chiamata nella quale tutti gli argomenti specifici
 # sono specificati attraverso il nome intero
 # Tutti gli argomenti passati vengono inseriti in m
 # match.call returns a call in which all of the specified arguments
 # are specified by their full names
        m <- match.call(expand.dots = FALSE)
 # Poniamo a NULL una serie di parametri
        m$method <- m$model <- m$control <- m$... <- m$x <- m$y <-
           m$wts <- m$split <- NULL

 # m[[1L]] diventa un nome -> model.frame.default
        m[[1L]] <- as.name("model.frame.default")

 # eval.parent valuta una funzione R in uno specifico ambiente
        m <- eval.parent(m)

 # Se il metodo richiesto è model.frame e non recursive.partition
 # ritorno il modello m altrimenti continuo l'esecuzione
        if(method == "model.frame")
              return(m)
    }

# Esegue il confronto con una serie di valori di una tabella
# In pratica legge il contenuto e vede se è tra le opzioni previste
# match.arg matches arg against a table of candidate values as
# specified by choices, where NULL means to take the first one.
    split <- match.arg(split)

# Estrae i termini del model.frame e li mette in un vettore Terms
    Terms <- attr(m, "terms")

# Controllo delle interazioni
# Se tra i termini letti esiste almeno un termine "order" e maggiore di 1
```

```
    if(any(attr(Terms, "order") > 1))
        stop("trees cannot handle interaction terms")

# Estraiamo la var risposta dal modello, se la risposta è multipla
# non può essere manipolata da un albero
    Y <- model.extract(m, "response")
    if(is.matrix(Y) && ncol(Y) > 1L)
        stop("trees cannot handle multiple responses")

# Estraggo i livelli della var risposta
    ylevels <- levels(Y)

# Estraggo i pesi dal modello
    w <- model.extract(m, "weights")

# Se i pesi non sono nulli, replico 1 per il numero di righe di m
    if(!length(w))
        w <- rep(1, nrow(m))
# yna vettore di valori true/false che indicano se un valore è nullo
# se uno dei valori è true, assegno 0 o 1 come segue.
    if(any(yna <- is.na(Y)))
    {
        Y[yna] <- 1
        w[yna] <- 0
    }
# GESTIONE DELL'OFFESET
# Estraggo l'attributo offset da "Terms"
    offset <- attr(Terms, "offset")
    if(!is.null(offset))
    {
        if(length(ylevels))
            stop("offset not implemented for classification trees")
        offset <- m[[offset]]
```

```
        Y <- Y - offset
    }
# Trasformo il modello in una matrice per contenere le info dell'albero
    X <- tree.matrix(m)
# Estraggo i livelli di X
    xlevels <- attr(X, "column.levels")
    if(is.null(xlevels))
    {
        xlevels <- rep(list(NULL), ncol(X))
        names(xlevels) <- dimnames(X)[[2L]]
    }
# Estraggo il numero di osservazioni della var risposta
    nobs <- length(Y)
    if(nobs == 0L)
       stop("no observations from which to fit a model")

    if(!is.null(control$nobs) && control$nobs < nobs)
    {
        stop("control$nobs < number of observations in data")
    }

# Richiamo la funzione in C "BDRgrow1My" nel file "grow.c"
# Alla funzione passo una serie di parametri ottenuti precedentemente.

    fit <- .C("BDRgrow1My",
              as.double(X),
              as.double(unclass(Y)),
              as.double(w),
              as.integer(c(sapply(xlevels, length), length(ylevels))),
              as.integer(rep(1, nobs)),
              as.integer(nobs),
              as.integer(ncol(X)),
              node = integer(control$nmax),
```

```
              var = integer(control$nmax),
              cutleft = character(control$nmax),
              cutright = character(control$nmax),
              n = double(control$nmax),
              dev = double(control$nmax),
              yval = double(control$nmax),
              yprob = double(max(control$nmax * length(ylevels), 1)),
              as.integer(control$minsize),
              as.integer(control$mincut),
              as.double(max(0, control$mindev)),
              nnode = as.integer(0L),
              where = integer(nobs),
              as.integer(control$nmax),
              as.integer(split=="gini"),
              as.integer(sapply(m, is.ordered)),
              NAOK = TRUE)
#---------------------------------------------------------------

# Numero di nodi dell'albero
    n <- fit$nnode

# Inizio a costruire il data frame con i vari elementi passati ad R
# dalla funzione in C

# In fit ci saranno: "var", "n", "dev", "yval"
# Prendo tutti i valori delle osservazioni
    frame <- data.frame(fit[c("var", "n", "dev", "yval")])[1L:n,  ]

    frame$var <- factor(frame$var, 0:length(xlevels),
                        c("<leaf>", names(xlevels)))
    frame$splits <-
     array(unlist(fit[c("cutleft", "cutright")]),
      c(control$nmax, 2),
```

```
      list(character(0L), c("cutleft", "cutright")))[1L:n, , drop = FALSE]

    if(length(ylevels))
    {
      frame$yval <- factor(frame$yval, 1L:length(ylevels), ylevels)
      class(frame$yval) <- class(Y)
      frame$yprob <-
        t(array(fit$yprob, c(length(ylevels), control$nmax),
                list(ylevels, character(0L)))[, 1L:n, drop = FALSE])
    }
    row.names(frame) <- fit$node[1L:n]
    fit <- list(frame = frame, where = fit$where, terms = Terms,
                call = match.call())
    attr(fit$where, "names") <- row.names(m)

    if(n > 1L)
      class(fit) <- "tree"
    else
      class(fit) <- c("singlenode", "tree")
    attr(fit, "xlevels") <- xlevels
    if(length(ylevels)) attr(fit, "ylevels") <- ylevels
    if(is.logical(model) && model) fit$model <- m
    if(x) fit$x <- X
    if(y) fit$y <- Y
    if(wts) fit$weights <- w

    # Restituisco fit
    fit
}
```

## A.2 Codice C

### A.2.1 Funzione `dividi livello()`

Questa funzione esegue la crescita non ricorsiva, limitandosi a fare una sola suddivisione del nodo che gli viene passato senza continuare a suddividere ricorsivamente.

```
void dividi_livello(int inode)
{
    int     i, iv, j, k, shift, shifted = False;
    double  bval, tmp;

    // Se il numero del nodo passato e' maggiore del numero
    // di nodi max -> err.
    if (inode >= nmax) error(_("tree is too big"));

    // Viene richiamata la funzione fillin_node
    fillin_node(inode);

    //printf("SE n[inode]=%f < minsize=%d, faccio return da
    // divide_node\n",n[inode],minsize);
    // SE entro in questo if vuol dire che sono in una foglia e posso
    // tornare indietro e analizzare un altro nodo

    if ( n[inode] < minsize )
    {
return;
    }

    // Calcolo di devtarget
    if (Gini)
    {
       bval = 0.0;
      for (k = 0; k < nc; k++)
```

```
        {
tmp = yprob[inode*nc + k];
bval += tmp *tmp;
}
     bval = n[inode] * (1 - bval);
     bval *= 2.0;
     Printf("gini = %g\n", bval);
     devtarget = bval;
    }
    else
    {
     bval = dev[inode];
     devtarget = dev[inode] - mindev*dev[0];
    }

    // Se la devianza scende sotto una soglia target interrompo la
    // crescita, sono in una foglia
    if(devtarget <= (1e-6)*dev[0])
{
//printf("La devianza è scesa sotto la soglia target!!
//Ritorno da dividi(%d)\n",inode);
return;
}

    // Poiche' la devianza non e' ancora scesa sotto la soglia obiettivo,
    // continuo
    // ESEGUO o split_disc o split_cont
    // Per trovare il valore migliore per lo split a seconda se la
    // variabile e' continua o discreta

    // Dentro a node[inode] c'è il numero del nodo che poi appare anche
    // nell'albero in R
    Printf("\n--evaluating node %d(%d) size %g\n", inode,(int)node[inode],
```

```
    // n[inode]);

    for (iv = 0; iv < nvar; iv++)
        if (levels[iv])
{
              split_disc(inode, iv, &bval);
}
        else
{
              split_cont(inode, iv, &bval);
}

    Printf("..best value is %g\n", bval);

    // Qui non ho ancora suddiviso le osservazioni, ho però trovato il
    // punto di suddivisione

    // Se non entro in questo if vuol dire che sono arrivato ad un nodo
    // foglia e torno su di un livello
    // nella ricorsione
    if (bval < devtarget)
    {
      //printf("..splitting\n");
      if ( node[inode] >=  1073741824 )
      {
         error(_("maximum depth reached\n")); //max profondita' albero
         return;
      }

      //printf("SE inode<nnode-1 (%d<%d)\n",inode,nnode-1);
      // Se il nodo che sto dividendo non e' l'ultimo
      if (inode < nnode-1)
      {
```

```
        shifted = nnode;
        for (i= nnode-1; i > inode; i--) shift_up_node(i, nmax-nnode);
        nnode = inode + 1;
            /*Printf("..shifted up\n");*/
       }
       else
       {
           shifted = False;
//printf("shifted=FALSE\n");
       }

     /* write left as nnode */
//printf("Continuiamo con il nodo a sinistra di %d, inode=%d\n",
//(int)node[inode], inode);
     for (j = 0; j < nobs; j++)
        {
         if (ttw[j] == 0) where[j] = nnode;
         if (ttw[j] == NALEVEL) where[j] += NALEVEL;
     }

node[nnode++] = 2 * node[inode];

     /* write right as nnode */
//printf("Continuiamo con il nodo a destra di %d, inode=%d\n",
// (int)node[inode], inode);
     for (j = 0; j < nobs; j++)
         if (where[j] == inode) where[j] = nnode;
node[nnode++] = 2 * node[inode] + 1;


     if (shifted)
        {
         shift = nnode - inode -1;
```

```
          for (i = inode+1; i < shifted; i++)
            shift_down_node(i+shift, nmax-shifted-shift);
          offset += shift;
          nnode = shifted + shift;
             /*Printf("..shifted down\n");*/
}
    }
}
```

### A.2.2 Funzione `suddividi dati()`

```
void suddividi_dati()
{
int i,j,k;
int nRighe[NUM_THREADS]={0,0,0,0};
int indicenodo[NUM_THREADS]={2,3,5,6}; //indice del vettore non è
// numero del nodo
int TotVar = nvar+1;

// Matrici di appoggio (OTTIMIZZANDO IL CODICE SI ARRIVA AD ELIMINARE
// QUESTA CREAZIONE
// STATICA RENDENDO DINAMICA LA CREAZIONE DI PUNTATORI A PUNTATORI)
double mat1[(int)n[2]][nvar], mat2[(int)n[3]][nvar],
mat3[(int)n[5]][nvar],mat4[(int)n[6]][nvar];

// ALLOCHIAMO I VETTORI DI PUNTATORI DA PASSARE ALLA PARTE RICORSIVA
// 1. XR
for(i=0;i<NUM_THREADS;i++)
{
XR[i] = (double *)calloc(n[indicenodo[i]]*nvar, sizeof(double));
}
// 2. yR
for(i=0;i<NUM_THREADS;i++)
```

```
{
yR[i] = (double *)calloc(n[indicenodo[i]], sizeof(double));
}
// 3. wR
for(i=0;i<NUM_THREADS;i++)
{
wR[i] = (double *)calloc(n[indicenodo[i]], sizeof(double));
}
// 4. levelsR
for(i=0;i<NUM_THREADS;i++)
{
levelsR[i] = (int *)calloc(TotVar, sizeof(int));
}

// 8. nodeR
for(i=0;i<NUM_THREADS;i++)
{
nodeR[i] = (int *)calloc(nmax, sizeof(int));
}
// 9. varR
for(i=0;i<NUM_THREADS;i++)
{
varR[i] = (int *)calloc(nmax, sizeof(int));
}

// 10. CUTLEFT
cutleftR = calloc(NUM_THREADS, sizeof(char)*MAX_CHAR);
// 11. CUTRIGHT
cutrightR = calloc(NUM_THREADS, sizeof(char)*MAX_CHAR);

// 12. nR
for(i=0;i<NUM_THREADS;i++)
{
```

```
nR[i] = (double *)calloc(nmax, sizeof(double));
}
// 13. devR
for(i=0;i<NUM_THREADS;i++)
{
devR[i] = (double *)calloc(nmax, sizeof(double));
}
// 14. yvalR
for(i=0;i<NUM_THREADS;i++)
{
yvalR[i] = (double *)calloc(nmax, sizeof(double));
}

// 20. whereR
for(i=0;i<NUM_THREADS;i++)
{
whereR[i] = (int *)calloc(n[indicenodo[i]], sizeof(int));
}

// 23. orderedR
for(i=0;i<NUM_THREADS;i++)
{
orderedR[i] = (int *)calloc(TotVar, sizeof(int));
}

// 24. junkR
for(i=0;i<NUM_THREADS;i++)
{
junkR[i] = (int *)calloc(n[indicenodo[i]], sizeof(int));
}


//-------------------------------------------------------
```

```
// RIEMPIO LE VARIABILI DA PASSARE A PARTE RICORSIVA
for(j=0;j<nobs;j++)
{
switch(where[j]) //In base a quale dei 4 gruppi appartengono
{
case 2:
{
for(i=0;i<nvar;i++)
mat1[nRighe[0]][i] = matX[j][i];
yR[0][nRighe[0]] = y[j];
wR[0][nRighe[0]] = w[j];
nobsR[0]=n[2];
nRighe[0]++;
break;
}
case 3:
{
for(i=0;i<nvar;i++)
mat2[nRighe[1]][i] = matX[j][i];
yR[1][nRighe[1]] = y[j];
wR[1][nRighe[1]] = w[j];
nobsR[1]=n[3];
nRighe[1]++;
break;
}
case 5:
{
for(i=0;i<nvar;i++)
mat3[nRighe[2]][i] = matX[j][i];
yR[2][nRighe[2]] = y[j];
wR[2][nRighe[2]] = w[j];
nobsR[2]=n[5];
nRighe[2]++;
```

```
break;
}
case 6:
{
for(i=0;i<nvar;i++)
mat4[nRighe[3]][i] = matX[j][i];
yR[3][nRighe[3]] = y[j];
wR[3][nRighe[3]] = w[j];
nobsR[3]=n[6];
nRighe[3]++;
break;
}
}
}

// Setto tutte le variabili da passare a "BDRgrow1R ricorsiva" ma
// che possono essere inizializzate con valori di default

for(k=0;k<NUM_THREADS;k++)
{
for(i=0;i<TotVar;i++)
{
levelsR[k][i] = 0; //levelsR
}
for(i=0;i<NUM_THREADS;i++)
{
nobsR[i]=n[indicenodo[i]]; //nobsR
}
nvarR[k]=nvar;
for(i=0;i<nmax;i++)
{
nodeR[k][i] = 0; //nodeR
}
```

```
for(i=0;i<nmax;i++)
{
varR[k][i] = 0; //varR
}
cutleftR[k] = calloc(nmax, sizeof(char)*MAX_CHAR);
cutrightR[k] = calloc(nmax, sizeof(char)*MAX_CHAR);
for(i=0;i<nmax;i++)
{
nR[k][i] = 0; //nR
}
for(i=0;i<nmax;i++)
{
devR[k][i] = 0; //devR
}
for(i=0;i<nmax;i++)
{
yvalR[k][i] = 0;
}
yprobR[k]=0; //yprobR
minsizeR[k]=minsize; //minsizeR
mincutR[k]=mincut; //mincutR

mindevR[k]=mindev; //mindevR
nnodeR[k]=0; //nnodeR
for(i=0;i<nobsR[k];i++)
{
whereR[k][i] = 0; //whereR
}
nmaxR[k] = nmax; //nmaxR
GiniR[k] = 0; //GiniR
for(i=0;i<TotVar;i++)
{
orderedR[k][i] = 0; //orderedR
```

```
}
for(i=0;i<nobsR[k];i++)
{
junkR[k][i] = 1; //junkR
}
numSuddivisioniR[k]=0; //numSuddivisioniR

}

// PASSO DA MATRICI DI X A VETTORI X
int tot;

for(k=0;k<NUM_THREADS;k++)
{
tot=0;
for(j=0;j<nvar;j++)
{
for(i=0;i<n[indicenodo[k]];i++)
{
switch(k)
{
case 0: // Nodo:4 Indice:2
{
XR[k][tot]=mat1[i][j];
tot++;
break;
}
case 1: // Nodo:5 Indice:3
{
XR[k][tot]=mat2[i][j];
tot++;
break;
}
```

```
case 2: // Nodo:6 Indice:5
{
XR[k][tot]=mat3[i][j];
tot++;
break;
}
case 3: // Nodo:7 Indice:6
{
XR[k][tot]=mat4[i][j];
tot++;
break;
}
}

}
}
}
}
```

### A.2.3 Funzione `create complete tree()`

```
create_complete_tree()
{
int found=0;
int i_r=0;
int j_r=0;
int k_r=0;

int node_backup[nmax];

for (int b=0;b<nmax;b++)
{node_backup[b]=0;}
```

```
while (node [i_r]!=0)
{node_backup[i_r]=node[i_r];i_r++;}

i_r=0;

while (i_r<NUM_THREADS)
{

if (node[k_r]==nodeR[i_r][j_r])
{
found=1;
int s_k=k_r;
while ((node[s_k])!=0)
{
node[s_k+nnodeR[i_r]]=node[s_k+1];
var[s_k+nnodeR[i_r]]=var[s_k+1];
dev[s_k+nnodeR[i_r]]=dev[s_k+1];
yval[s_k+nnodeR[i_r]]=yval[s_k+1];
n[s_k+nnodeR[i_r]]=n[s_k+1];
cutleft[s_k+nnodeR[i_r]]=cutleft[s_k+1];
cutright[s_k+nnodeR[i_r]]=cutright[s_k+1];
s_k++;
}
while (nodeR[i_r][j_r]!=0)
{
node[k_r]=nodeR[i_r][j_r];
var[k_r]=varR[i_r][j_r];
dev[k_r]=devR[i_r][j_r];
yval[k_r]=yvalR[i_r][j_r];
n[k_r]=nR[i_r][j_r];
cutleft[k_r]=cutleftR[i_r][j_r];
cutright[k_r]=cutrightR[i_r][j_r];
k_r++;
```

```
j_r++;
}
i_r++;
j_r=0;
}
else
{
  k_r++;
}

}

k_r=0;
for(int j=0;j<NUM_THREADS;j++)
{
for(int i=0;i<nobsR[j];i++)
{
whereR[j][i]=nodeR[j][whereR[j][i]];
while (node[k_r++]!=whereR[j][i]){};
whereR[j][i]=k_r;
k_r=0;
}
}

int s_z=0;
int i_n=0;
int i_m = 0;

for(int j=NUM_THREADS-1;j>=0;j--)
{
while (nodeR[j][0]!=node_backup[s_z]) {s_z++;};

while (i_n<nobs)
```

```
{
if (where[i_n]==s_z)
{
where[i_n]=whereR[j][i_m];
i_m++;
}
i_n++;
}

i_n=0;
i_m=0;
s_z=0;
}

nnode = nnode - NUM_THREADS;
for(int j=0;j<NUM_THREADS;j++)
{
nnode=nnode+nnodeR[j];
}
}
```

### A.2.4 Funzione `BDRgrow1My()`

```
void
BDRgrow1My(double *pX, double *pY, double *pw, Sint *plevels, Sint *junk1,
 Sint *pnobs, Sint *pncol, Sint *pnode, Sint *pvar, char **pcutleft,
 char **pcutright, double *pn, double *pdev, double *pyval,
 double *pyprob, Sint *pminsize, Sint *pmincut, double *pmindev,
 Sint *pnnode, Sint *pwhere, Sint *pnmax, Sint *stype, Sint *pordered)
{
    // Assegno il contenuto delle variabili passate alle variabili globali
    // inizializzate all'inizio del file
    X = pX; y = pY; w = pw; dev = pdev; yval = pyval; yprob = pyprob;
```

```
    nobs = *pnobs; nvar = *pncol;
    levels = plevels; node = pnode; var = pvar; n = pn; mindev = *pmindev;
    minsize = *pminsize; mincut = *pmincut; nmax = *pnmax;
    nnode = *pnnode;
    where = pwhere; cutleft = pcutleft; cutright = pcutright;
    ordered= pordered; Gini = *stype;

 // i e nl sono variabili private della funzione "BDRgrow1"
    int i, nl;

    // nc variabile globale
    nc = levels[nvar];

    // ALLOCO LA MEMORIA DELLE VARIABILI CHE DEVONO ESSERE PASSATE ALLA
    // PARTE RICORSIVA
    // I PUNTATORI SONO DIVENTATI PUNTATORI A PUNTATORI, LE VARIABILI SONO
    // DIVENTATE PUNTATORI

mindevR = calloc(NUM_THREADS, sizeof(double)); //1
XR = calloc(NUM_THREADS, sizeof(double)); //2
yR = calloc(NUM_THREADS, sizeof(double)); //3
wR = calloc(NUM_THREADS, sizeof(double)); //4
nR = calloc(NUM_THREADS, sizeof(double)); //5
devR = calloc(NUM_THREADS, sizeof(double)); //6
yvalR = calloc(NUM_THREADS, sizeof(double)); //7
yprobR = calloc(NUM_THREADS, sizeof(double)); //8
mindevR = calloc(NUM_THREADS, sizeof(double)); //9

levelsR = calloc(NUM_THREADS, sizeof(int)); //10
nodeR = calloc(NUM_THREADS, sizeof(int)); //11
varR = calloc(NUM_THREADS, sizeof(int)); //12
whereR = calloc(NUM_THREADS, sizeof(int)); //13
orderedR = calloc(NUM_THREADS, sizeof(int)); //14
```

```
nobsR = (int *)calloc(NUM_THREADS, sizeof(int)); //15
nvarR = (int *)calloc(NUM_THREADS, sizeof(int)); //16
nmaxR = (int *)calloc(NUM_THREADS, sizeof(int)); //17
minsizeR = (int *)calloc(NUM_THREADS, sizeof(int)); //18
mincutR = (int *)calloc(NUM_THREADS, sizeof(int)); //19
nnodeR = (int *)calloc(NUM_THREADS, sizeof(int)); //20
GiniR = (int *)calloc(NUM_THREADS, sizeof(int)); //21

cutleftR = calloc(NUM_THREADS, sizeof(char)*MAX_CHAR); //22
cutrightR = calloc(NUM_THREADS, sizeof(char)*MAX_CHAR); //23

junkR = calloc(NUM_THREADS,sizeof(double));
numSuddivisioniR = (int *)calloc(NUM_THREADS, sizeof(int));

    // Qualche stampa di controllo
    Printf("nnode: %d\n", nnode);    // Numero di nodi iniziali
    Printf("nvar: %d\n", nvar);      // Numero di variabili esplicative

    for(i = 0; i <= nvar; i++)
            Printf("%d ", (int)levels[i]); // Stampo i livelli iniziali
    Printf("\n");

    /* allocate scratch storage */
    nl = 0;
    for(i = 0; i <= nvar; i++)
       if (levels[i] > nl) nl = levels[i];
    maxnl = max(nl, 10);

    // Allocazione delle altre variabili globali non passate da R

    twhere = (int *) S_alloc(nobs, sizeof(int));
    ttw = (int *) S_alloc(nobs, sizeof(int));
```

```
tvar = (double *) S_alloc(nobs, sizeof(double));
ind = (int *) S_alloc(nl, sizeof(int));
w1 = (double *) S_alloc(nobs, sizeof(double));
cnt = (double *) S_alloc(nl, sizeof(double));
cprob = (double*) S_alloc(nl, sizeof(double));
scprob = (double*) S_alloc(nl, sizeof(double));
indl = (int*) S_alloc(nl, sizeof(int));
if (nc > 0)
{
 yp = (double *) S_alloc(nc, sizeof(double));
 tab = (double*) S_alloc(nl*(1+nc), sizeof(double));
 indr = (int*) S_alloc(nl, sizeof(int));
 ty = (int *) S_alloc(nobs, sizeof(int));
}
else
{
    tyc = (double *) S_alloc(nobs, sizeof(double));
    ys = (double *) S_alloc(nl, sizeof(double));
}

// Inizializzazione di due variabili globali
exists = nnode;
//printf("exists: %d \n",exists);
offset = 0;

// INIZIO CICLI PER LA CRESCITA DELL'ALBERO

// DIVIDO AL PRIMO LIVELLO, OTTENGO 2 RAMI (DX e SX)
// Se non abbiamo nessun nodo o un solo nodo (siamo all'inizio)
if (exists <= 1)
{
   // Creo vettore di zeri, indica che tutte le osservazioni sono nel
   // nodo 0.
```

```
for(i = 0; i < nobs; i++) where[i] = 0;
       // Pongo numero di nodi = 1, ho il nodo(0) che contiene tutte le
       // osservazioni
nnode = 1;
       // Pongo primo elemento del vettore "node" a 1
node[0] = 1;
       // Chiamo la funzione per suddividere il primo nodo
numSuddivisioni++;
dividi_livello(0);
    }
    else
    {
   /* Adjust from S indexing */
   //Decremento il vettore where di i per ogni osservazione
        for(i = 0; i < nobs; i++) where[i]--;

        // Ciclo finche' non arrivo al numero di nodi esistenti
        for(i = 0; i < exists; i++)
// se l'elemento del vettore var in posizione [i+offset] non e' nullo
if (!var[i+offset])
          {
/* Printf("trying node %d at offset %d, nnode %d\n", i,
 offset, nnode);*/
//printf("Adesso eseguo divide_node(%d)\n", i+offset);
             dividi_livello(i + offset);
         }
    }
// IN CASO DI 4 THREAD DEVO SUDDIVIDE ANCORA IN DUE ALBERI PRIMA DI
// FARE LA PARTE RICORSIVA
if(NUM_THREADS==4)
{
// INIZIO SUDDIVISIONE AL SECONDO LIVELLO
numSuddivisioni++;
```

```
dividi_livello(1); // Passo inode, posizione del nodo
//nel vettore nodi[1,2,3], voglio cosi dividere nodo 2

numSuddivisioni++;
dividi_livello(4);

// HO SUDDIVISO I PRIMI DUE LIVELLI

// Completo le variabili che sono aggiornate in fillin node

fillin_node(2);
fillin_node(3);
fillin_node(5);
fillin_node(6);
}
else  // CASO CON 2 THREAD, AL MOMENTO NON CONSIDERO TUTTI I
// POSSIBILI NUM DI THREAD
{
fillin_node(1);
fillin_node(2);
}
//-----------------------------------------------------------------
// SUDDIVISIONE DATI ORIGINARI IN PIU' ELEMENTI DA PASSARE AD OGNI
// FUNZIONE RICORSIVA

// TRASFORMO VETTORE X IN MATRICE
int j,v,k;
j=0,v=0;

for(i=0;i<nvar;i++)
{
for(k=0;k<nobs;k++)
{
```

```
matX[k][j]=X[v];
v++;
}
j++;
}

// DEVO SUDDIVIDERE I DATI X IN 4 o 2 SOTTOMATRICI IN BASE A COME SONO
// RAGGRUPPATI I DATI
// SUDDIVIDO ANCHE I VETTORI Y

if(NUM_THREADS==4)
suddividi_dati_4_thread();
else  // CASO CON SOLO 2 THREAD
suddividi_dati_2_thread();

//-----------------------------------------------------------------------
    // Ogni ramo verrà assegnato a un thread il quale al suo interno potrà
    // richiamare la funzione ricorsiva divide_node.
int q=0;
omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(q)
    {
//#pragma omp single // DIRETTIVA CHE SERVIVA PER LANCIARE
// PROGRAMMA IN MODO SEQUENZIALE
// Richiamo la funzione BDRgrow1 del file per la ricorsione
#pragma omp for schedule(static,1)
for(i=0; i<NUM_THREADS; i++)
{
q=omp_get_thread_num();
//printf("Esecuzione del thread q=%d: i=%d\n", q,i);
BDRgrow1_R(XR, yR, wR, levelsR, junkR,
 nobsR, nvarR, nodeR, varR, cutleftR,
 cutrightR, nR, devR, yvalR,
```

```
 yprobR, minsizeR, mincutR, mindevR,
 nnodeR, whereR, nmaxR, GiniR, orderedR, i);
}
    }
  for (i=0;i<NUM_THREADS;)
  {
j=0;
  while (nodeR[i][j] != 0)
  {
int max_pow=pow(2.0,(int)my_log((double)(nodeR[i][j]+0.1)
 , 2.0));
nodeR[i][j]=max_pow*NUM_THREADS - max_pow + nodeR[i][j] +
 max_pow*i;
j++;
  }
   i++;
  }
// RINOMINO CORRETTAMENTE I NODI E UNISCO RISULTATI
create_complete_tree();

*pnnode = nnode;
    Printf("\nFinished!\n");
}
```

# Bibliografia


Agati R. (2009). Calcolo parallelo, architettura tradizionale per il calcolo parallelo e distribuito e possibile implementazione tramite l'uso di personal computer. `http://www.renatoagati.com/scuola/appunti/Calcolo_parallelo.pdf`.

Azzalini A., Scarpa B. (2004). *Analisi dei dati e data mining*. Springer, 1 edizione.

Baesso L. (2010-2011). *Sistemi multiprocessore e multicore*. Tesi per Master, Università degli studi di Padova.

Bonaccorso F. (2005). Openmp. `http://www.mat.uniroma1.it/centro-calcolo/HPC/materiale-corso/OpenMP-Intro1.pdf`.

Breiman L., Friedman J., Olshen R., Stone C. (1984). *Classification Regression Trees*. Wadsworth International Group.

Buyya R. (1999). *High Performance Cluster Computing: Programming and Applications, vol. 2*. Prentice Hall, 1 edizione.

Campelli C. M., Stagni A. (2010-2011). *Metodi numerici di calcolo parallelo per la soluzione di reti di reattori*. Tesi per Master, Politecnico di Milano.

Chergui J., Lavallée P.-F. (2012). Openmp multithreaded parallelization for shared-memory machines.

Dongarra J., Foster I., Fox G., Gropp W., Kennedy K., Torczon L., White A. (2003). *Sourcebook of parallel computing*. Morgan Kaufmann Publishers.

Ein-Dor P., Feldmesser J. (1987). Attributes of the performance of central processing units: a relative performance prediction model. *Comm. ACM.*, (30), 308–317.

Hunt E. B., Marin J., Stone P. J. (1966). Experiments in induction. *New York: Academic Press*.

Jin R., Yang G., Agrawal G. (2004). Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance. *IEEE transaction in knowledge and data engineering*, **16**(10).

Kumbaro O., Ravera G., Stringhini G. (2007). *Tutorial su OpenMP*. `ftp://ftp.giorgioravera.it/pub/OpenMP.pdf`.

Matloff N. (2011). *The Art of R Programming (A Tour of Statistical Software Design)*. No Starch Press, Inc.

McCallum Q. E., Weston S. (2011). *Parallel R*. O'Reilly Media Inc., 1 edizione.

Meadows L., Bull M., Mattson T. (2009). *A "Hands-on" Introduction to OpenMP*. `http://openmp.org/mp-documents/omp-hands-on-SC08.pdf`.

Mehta M., Agrawal R., Rissanen J. (1996). Sliq: A fast scalable classifier for data mining. *IBM Almaden Research Center*, pp. 18–32.

Miller D., Mattson K. (2011). *OpenMP 3.1 API C/C++ Syntax Quick Reference Card*. `http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf`.

Org. O. (2002). *OpenMP C and C++ Application Program Interface*. `http://www.openmp.org/mp-documents/cspec20.pdf`.

Pas R. V. D. (2005). An introduction into openmp. `http://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf`.

Pimentel A. D., Vassiliadis S. (2004). *Computer Systems: Architectures, Modeling and Simulation*. Springer, 4 edizione.

Quinlan J. R. (1986). *Induction of decision trees. Machine Leaning, vol (1)*. Springer.

Quinlan J. R. (1987). Simplifying decision trees. *International Journal of Machine Studies*, (27), 221–234.

Shafer J., Mehta M., Agrawal R. (1996). Sprint: A scalable parallel classifier for data mining. *IBM Almaden Research Center*, pp. 544–555. Morgan Kaufmann.

Skillicorn D. B. (1999). Strategies for parallel data mining. *IEEE Concurrency*.

Xiao H. (2010). Towards parallel and distributed computing in large-scale data mining: A survey. Technical University of Munich.