Alma Mater Studiorum · Università di Bologna

**SCUOLA DI SCIENZE**
**Corso di Laurea in Informatica Magistrale**

# UN ALGORITMO GENETICO PARALLELO PER LA K-COLORABILITÀ

**Tesi di Laurea in Algoritmi Avanzati**

**Relatore:**
**Chiar.mo Prof.**
**MORENO MARZOLLA**

**Presentata da:**
**PASQUALE CAUTELA**

**Sessione III**
**Anno Accademico 2012/2013**

Un grazie sincero al Prof. Moreno Marzolla
per la massima disponibilità accordatami
e per la grande passione con la quale
svolge il proprio lavoro.

*A Laura e Bruno...*

# Indice

**1 Grafi** **3**

1.1 Teoria dei Grafi . . . . . 3

1.2 $K$-Colorabilità . . . . . 4

1.3 Grafi nel mondo reale . . . . . 6

**2 Algoritmi Genetici** **9**

2.1 Definizione . . . . . 9

2.1.1 Selezione . . . . . 10

2.1.2 Crossover . . . . . 11

2.1.3 Mutazione . . . . . 11

2.2 Algoritmi Genetici Paralleli (PGA) . . . . . 12

2.3 Campo di utilizzo . . . . . 14

**3 Il Calcolo Parallelo** **15**

3.1 Utilità e applicazioni . . . . . 16

3.2 Architetture . . . . . 17

3.3 Calcolo della complessità . . . . . 19

**4 Algoritmo sequenziale** **23**

4.1 Rappresentazione del Problema . . . . . 23

4.2 Fitness . . . . . 24

4.3 Selezione . . . . . 25

4.4 Crossover . . . . . 25

4.5 Mutazione . . . . . 25

**5 Algoritmo parallelo** **27**
5.1 Sezione da parallelizzare . . . . . 27
5.2 Parallelizzazione della popolazione . . . . . 28
5.3 Parallelizzazione del singolo individuo . . . . . 31
5.4 Parallelizzazione complessiva . . . . . 33
5.5 Validazione . . . . . 33

**6 MPI** **35**
6.1 Stuttura del programma e principali funzioni . . . . . 36
6.2 Gruppi e Comunicatori . . . . . 39

**7 BlueGene/Q** **41**
7.1 Architettura . . . . . 41
7.2 Operazioni preliminari . . . . . 42

**8 Fase di Sperimentazione** **45**
8.1 Legge di Amdahl . . . . . 45
8.2 Configurazione 1 . . . . . 46
8.3 Configurazione 2 . . . . . 48
8.4 Configurazione 3 . . . . . 49
8.5 Analisi delle prestazioni . . . . . 51

**Conclusioni** **55**

**Bibliografia** **57**

# Elenco delle figure


1.1 Esempio di grafo orientato. . . . . 4
1.2 Esempio di grafo completo non orientato. . . . . 4
1.3 Esempio di grafo colorato con due colori. . . . . 6

2.1 Assegnazione delle probabilità nella selezione proporzionale. . 11
2.2 Esempio di crossover ad un punto. . . . . 11
2.3 Esempio di mutazione. . . . . 12
2.4 Struttura algoritmo genetico nel modello globale. . . . . 13

3.1 Grafico del numero di transistor per chip dagli anni '70.
Fonte: www.gotw.ca/publications/concurrency-ddj.htm . . . . 16
3.2 Struttura di un'architettura a memoria condivisa . . . . . 19
3.3 Struttura di un'architettura a memoria distribuita . . . . . 19
3.4 Grafico dello speedup . . . . . 20
3.5 Grafico dell'efficienza . . . . . 21

4.1 Esempio di rappresentazione con 5 nodi e 3 colori . . . . . 24

5.1 Analisi dei tempi di esecuzione dell'algoritmo con Gprof . . . . 28
5.2 Distribuzione di una popolazione di 32 individui su 4 processori 29
5.3 Distribuzione di una matrice 8x8 su 4 processori . . . . . 32
5.4 Parallelizzazione con 16 processori e 4 master . . . . . 33

6.1 Struttura generale di un programma MPI [14] . . . . . 38
6.2 Principali funzioni MPI [14] . . . . . 38

6.3 Esempio di comunicatori con 16 processori . . . . . 39

7.1 Architettura del Fermi BlueGene . . . . . 42
7.2 Fermi BlueGene . . . . . 43

8.1 Speedup parallelizzazione popolazione su *test256.col* . . . . . 47
8.2 Speedup parallelizzazione popolazione su *test512.col* . . . . . 47
8.3 Speedup parallelizzazione singolo individuo su *test256.col* . . . 48
8.4 Speedup parallelizzazione singolo individuo su *test512.col* . . . 49
8.5 Speedup parallelizzazione combinata . . . . . 50
8.6 Speedup parallelizzazione combinata . . . . . 50
8.7 Bontà dei risultati su *test256.col* . . . . . 53
8.8 Bontà dei risultati su *test512.col* . . . . . 53

# Elenco delle tabelle

1.1 Esempi di grafi nel mondo reale . . . . . 6

5.1 Risultati della validazione . . . . . 34

8.1 Schema della parallelizzazione definitiva . . . . . 52

# Sommario

Il calcolo parallelo rappresenta un ambito di ricerca estremamente interessante nel quale però risulta difficile riuscire ad ottenere risultati soddisfacenti a causa di diversi fattori che spiegheremo in seguito nel capitolo 3. Per tali ragioni, spesso, l'applicazione delle conoscenze in questo ambito viene riservata a problemi di tipo accademico. Nel nostro lavoro ci proponiamo di implementare un algoritmo genetico per risolvere un problema classico dell'informatica ossia la colorazione di un grafo con il minor numero di colori. Nei capitoli 1 e 2 descriveremo le principali nozioni della teoria dei grafi e descriveremo in generale un algoritmo genetico. Nel capitolo 3 approfondiremo il calcolo parallelo analizzando le principali architetture parallele. I capitoli 4 e 5 costituiscono il punto centrale di questo lavoro nel quale presenteremo l'algoritmo sequenziale dal quale si è partiti e l'implementazione della sua versione parallela utilizzando le librerie MPI (capitolo 6) analizzando anche tutte le scelte strutturali che sono state necessarie. Il lavoro si conclude con il capitolo 8 nel quale vi è la presentazione dei risultati ottenuti mediante sperimentazioni eseguite sul supercoumputer *BlueGene/Q Fermi* del quale invece si parlerà brevemente nel capitolo 7.

# Capitolo 1

# Grafi

In questo primo capitolo tratteremo uno degli argomenti principali su cui si basa l'intero lavoro: i grafi [1, 2]. Spiegheremo cosa sono i grafi, ne daremo una descrizione delle principali caratteristiche, enunceremo alcuni teoremi che ci sono stati utili in alcune nostre scelte ed infine presenteremo il problema che ci prefissiamo di risolvere.

## 1.1 Teoria dei Grafi

Iniziamo con il definire un *grafo* $G$ come una coppia $G = (V, E)$ dove

- $V$ è un insieme finito di $n$ elementi detti *nodi* o *vertici* del grafo.
- $E$ è un insieme finito di $m$ coppie di elementi di $V$; ciascuna di queste coppie è detta *arco* del grafo.

Un grafo è detto *orientato* se i suoi archi lo sono ossia se ciascuno dei suoi archi ha una direzione e le coppie $(i, j)$ e $(j, i)$ sono considerate distinte.
Un grafo è detto *non orientato* se, invece, i suoi archi non hanno alcuna direzione e le coppie $(i, j)$ e $(j, i)$ sono considerate la stessa coppia.
Si definisce *cammino* una sequenza di nodi $v_0, v_1, ..., v_k$ tali che

$$(v_i, v_{i+1}) \in E \ \forall \ i = \{0, 1, \cdots, k-1\}$$

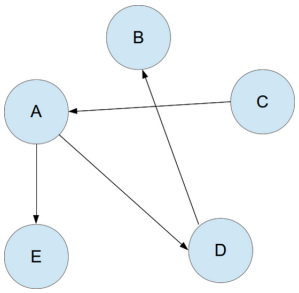


Figura 1.1: Esempio di grafo orientato.

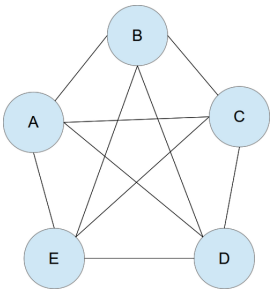


Figura 1.2: Esempio di grafo completo non orientato.

Un grafo si dice *connesso* se $\forall\ v_i, v_j \in V$, dove $i \neq j$, esiste almeno un cammino $v_i, \cdots, v_j$ (vedi figura 1.1).

Un grafo si dice *completo* se esiste un arco che collega ogni coppia di nodi (vedi figura 1.2).

Nel nostro lavoro utilizzeremo solo grafi non orientati e connessi ma non necessariamente completi.

Definiamo ora il *grado* di un nodo come il numero di archi incidenti in tale nodo e il *grado* di un grafo come il massimo dei gradi dei suoi nodi. Tale definizione ci sarà molto utile in alcune scelte in fase di implementazione dell'algoritmo oggetto di questo lavoro.

## 1.2 $K$-Colorabilità

Descriviamo in questa sezione il problema della *k-colorabilità di un grafo*. Si tratta di un problema molto noto nell'ambito informatico che appartiene

alla categoria dei problemi *NP-Completi* [5, 6].

Una *k-colorazione* dei vertici di un grafo $G$ consiste nell'assegnazione di $k$ colori $(1, 2, 3, ..., k)$ ai vertici di $G$ in modo tale che a due vertici adiacenti non sia assegnato lo stesso colore. Diremo *k-colorabile* ogni grafo $G$ che ammette una $k$-colorazione.
Un problema di colorazione di un grafo è un particolare tipo di k-colorabilità: si tratta di trovare il minimo $k$ con il quale è possibile colorare il grafo. Tale k è detto *numero cromatico* del grafo e viene di solito denotato con $\chi(G)$. Se $k = 1, 2, 3...$ e $P(G, k)$ è il numero di possibili soluzioni per colorare il grafo G con k colori allora

$$\chi(G) = min(k : P(G, k) > 0)$$

Esistono in generale diverse tecniche di risoluzione di un problema di colorazione di un grafo. Si può, ad esempio, definire un numero di colori fisso e assegnare a ciascun vertice un colore che non genera conflitti; in questo caso quindi potrebbero restare vertici non colorati e quindi il problema diventerebbe massimizzare il numero di vertici che si è riusciti a colorare. Un'altra tecnica invece prevede di scegliere un numero di colori sufficientemente alto e di provare a colorare il grafo; se si giunge ad una soluzione valida, si decrementa di uno il numero di colori e si ripete la procedura fino a che non si giunge ad un numero di colori che non permette di trovare una soluzione [7]. Questo approccio è esattamente quello che noi utilizzeremo nel nostro lavoro. La scelta del numero di colori dal quale partire è stata dettata dal seguente teorema

**Teorema 1.2.1.** *Se ogni vertice di un grafo G ha grado minore o uguale ad un numero d, allora il grafo è (d+1)-colorabile.*

Questo teorema ci dice quindi che il numero cromatico di un grafo è sicuramente minore o uguale al massimo dei gradi dei suoi vertici incrementato di uno e per tale ragione il nostro algoritmo inizierà a colorare il grafo con un numero di colori proprio uguale a tale valore. Il fatto che il teorema fornisca

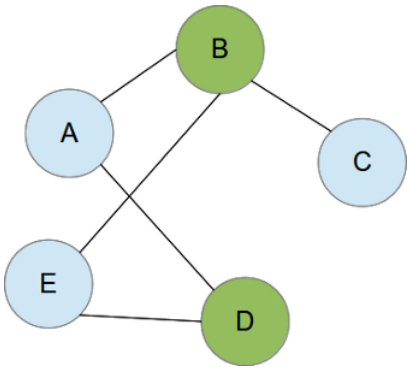


Figura 1.3: Esempio di grafo colorato con due colori.

un *upper bound* del numero cromatico non esclude che esso possa essere un numero anche molto più basso.

## 1.3 Grafi nel mondo reale

Utilizzando la struttura di un grafo è possibile rappresentare innumerevoli situazioni non solo riguardanti l'ambito matematico ma anche situazioni del mondo reale. Nella seguente tabella ne vediamo alcuni esempi:

| *Situazione* | *Nodi* | *Archi* |
|---|---|---|
| *Internet* | *Pagine* | *Links* |
| *Reti Stradali* | *Incroci* | *Strade* |
| *Incontri sportivi* | *Squadre* | *Incontri* |
| *Facebook* | *Persone* | *Amicizie* |
| *Giochi* | *Posizioni* | *Mosse* |

Tabella 1.1: Esempi di grafi nel mondo reale

È facile intuire quindi che il problema della colorazione di un grafo può risultare utile per risolvere numerosi problemi pratici. Per esempio sono risolvibili con questo approccio problemi di scheduling, di colorazione di mappe,

di assegnazione di frequenze radio e giochi (il famoso *sudoku* può essere visto come un problema di colorazione di un grafo di 81 nodi con 9 colori).

# Capitolo 2

# Algoritmi Genetici

## 2.1 Definizione

Gli *algoritmi genetici* sono delle procedure complesse che, imitando i processi dell'evoluzione naturale, si propongono di risolvere problemi di ricerca e di ottimizzazione. In particolare gli algoritmi genetici si basano sulle teorie presentate nel 1859 da Charles Darwin ossia sul principio che gli individui più forti hanno più possibilità di sopravvivere e di trasmettere ai loro successori le caratteristiche che li distinguono dagli individui più deboli.
Furono introdotti per la prima volta da John Holland nei primi anni degli anni '70 e da allora sono stati abbondantemente trattati in letteratura [9].
Le idee che stanno alla base degli algoritmi genetici sono quelle quindi della selezione naturale, della riproduzione sessuale e della mutazione casuale.
Cerchiamo ora di spiegare il funzionamento di un algoritmo genetico nella pratica. L'algoritmo genetico prevede innanzitutto la creazione di una *popolazione* casuale con un certo numero di individui ognuno dei quali rappresenta un candidato ad essere una soluzione del problema; a ciascuno di questi individui viene assegnato un valore mediante una funzione di *fitness* appositamente creata e tale valore rappresenta la bontà di quel candidato. A questo punto si passa a selezionare secondo diversi criteri (che vedremo a breve) i migliori individui, a farli accoppiare mediante la fase di *crossover* ed

infine ad operare una piccola mutazione casuale su ciascun individuo. Alla fine di queste fasi si arriva ad ottenere una nuova popolazione di individui che rappresenta la nuova *generazione*.
Ciascuna generazione tende a conservare le caratteristiche dei migliori individui di quella precedente e questo fa in modo che l'algoritmo genetico, durante la ricerca delle soluzioni, si concentri su uno spazio di ricerca ristretto. L'algoritmo continua fino a quando non si giunge ad una generazione in cui è presente un individuo che è soluzione del problema oppure fino a quando non sono state generate un numero di generazioni uguale a quello prefissato (in questo caso si avrà una approssimazione della soluzione).
In generale per rappresentare ciascun individuo della popolazione si utilizza una stringa di bit.
Analizziamo più nel dettaglio le tre fasi appena introdotte.

### 2.1.1 Selezione

Come già accennato in questa fase l'obiettivo primario è quello di selezionare gli individui migliori e di fare in modo che essi abbiano maggiori possibilità di "sopravvivere" nella generazione successiva rispetto a quelli con un fitness più basso.
Esistono diverse tecniche di selezione e le più utilizzate sono:

- *selezione proporzionale* che prevede l'assegnazione di una probabilità di sopravvivenza a ciascun individuo proporzionale al valore di fitness dell'individuo stesso (vedi figura 2.1);

- *selezione a torneo* che prevede l'estrazione casuale di due individui dalla popolazione, dei quali quello con il fitness maggiore "vincerà la sfida" e sopravviverà, mentre l'altro sarà dichiarato perdente.

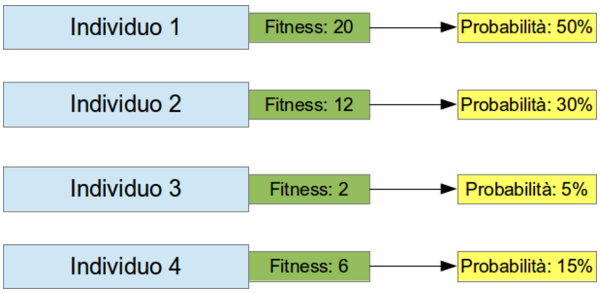


Figura 2.1: Assegnazione delle probabilità nella selezione proporzionale.

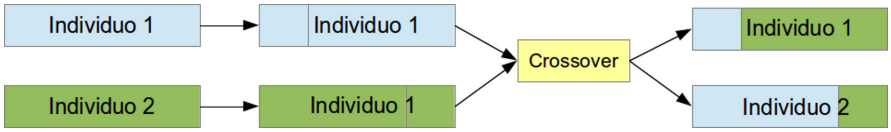


Figura 2.2: Esempio di crossover ad un punto.

### 2.1.2 Crossover

Questa è la fase che simula la riproduzione e consiste nel prendere due individui dalla popolazione e generarne uno nuovo che abbia alcune delle caratteristiche di entrambi quelli di partenza. Dal punto di vista pratico, dal momento che come già detto, ciascun individuo è rappresentato da una stringa di bit, il crossover si effettua scegliendo casualmente un punto (esistono anche crossover a due o più punti) della stringa e fondendo la prima parte del primo individuo con la seconda del secondo e viceversa. Possiamo osservarne un esempio nella figura 2.2.

### 2.1.3 Mutazione

Si tratta dell'ultima fase di un algoritmo genetico. È quella che rappresenta i cambiamenti operati dalla natura generati in modo quasi del tutto imprevedibile. L'operatore di mutazione, infatti, al punto di vista pratico non fa altro che apportare una piccola modifica alla stringa che rappresenta ciascun individuo (vedi figura 2.3). È importante sottolineare che la mutazione

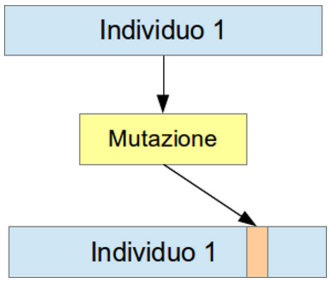


Figura 2.3: Esempio di mutazione.

può anche essere saltata e questo viene determinato da un certo *coefficiente di mutazione*.

## 2.2 Algoritmi Genetici Paralleli (PGA)

Dal momento che, come già detto, gli algoritmi genetici cercano una soluzione del problema esplorando solo una parte ristretta dello spazio di ricerca, è possibile che essi non trovino la soluzione esatta ma solo una soluzione approssimata oppure che essi restituiscano come soluzione un valore che invece rappresenta solo un massimo locale e non globale. Ovviamente, un algoritmo genetico è tanto più affidabile quanto più grandi sono il numero di generazioni da calcolare e soprattutto il numero di individui presenti nella popolazione iniziale; quindi per ottenere risultati soddisfacenti è necessario utilizzare una popolazione sufficientemente grande. Questo però comporta uno sforzo computazionale notevole che spesso diventa inaccettabile. É per tale ragione che anche su questo argomento si è molto discusso in letteratura [7] e sono stati proposti numerosi algoritmi genetici capaci di lavorare su più processori in parallelo. Andiamo ad analizzare alcune delle principali tecniche di parallelizzazione di un algoritmo genetico.

- *Modello globale.* Questa tecnica prevede la creazione di una sola popolazione di individui e l'esecuzione degli operatori genetici (selezione, crossover e mutazione) in sequenziale su un singolo processore. Dal momento che, in generale, la fase più pesante dal punto di vista com-

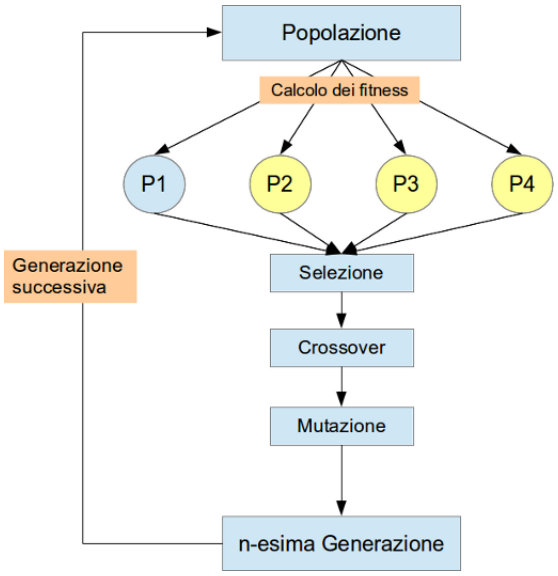


Figura 2.4: Struttura algoritmo genetico nel modello globale.

putazionale di un algoritmo genetico è costituita dal calcolo dei fitness degli individui, la tecnica del modello globale prevede di parallelizzare esattamente questa fase. Si supponga di avere a disposizione $p$ processori; il processore che avvia l'algoritmo genetico è detto *master* e si occupa di creare la popolazione di individui, eseguire le operazioni genetiche e nel momento in cui deve calcolare il fitness degli individui distribuisce il carico sugli altri *p-1* processori detti *slaves* (vedi figura 2.4).
Nel nostro lavoro ci atterremo ad una variante di questo modello.

- *Modello multi-popolazione.* In questa tecnica la popolazione dell'intero algoritmo genetico è divisa in sotto-popolazioni ciascuna delle quali si trova su di un singolo processore; ogni processore quindi applica gli operatori genetici sulla propria sotto-popolazione e solo in alcuni casi i processori si scambiano alcuni individui. La frequenza di questo scambio determina quanto ampio debba essere lo spazio di ricerca della soluzione. In generale questa tecnica, rispetto a quella illustrata precedentemente, fa si che soprattutto nella fase iniziale le parti dello spazio

di ricerca su cui si va ad esplorare siano diversificate in maniera molto più marcata.

## 2.3 Campo di utilizzo

Gli algoritmi genetici trovano la loro collocazione naturale all'interno di una disciplina molto più ampia: l'intelligenza artificiale [10]. Questa si occupa in generale dello studio dei processi mentali e di come riuscire a riprodurli mediante l'utilizzo di un computer; si pone come obiettivo principale quello di riuscire a fare in modo che le macchine riescano ad apprendere e ad imparare come un vero essere umano. I risultati ottenuti sono davvero interessanti e i campi di applicazione vanno dalla realizzazione di robot programmati per imparare, a videogiochi che grazie a questa tecnica riescono a generare nuovi schemi e situazioni di gioco, a macchine che riescono a giocare migliorando le proprie prestazioni ad ogni partita fino a diventare più forti di un essere umano e molti altri ancora. Ad oggi non esiste ancora una definizione univoca di intelligenza artificiale per via di contrapposizioni ideologiche tra diverse correnti di studiosi della disciplina. L'esempio classico didattico utilizzato quando si parla di algoritmi genetici è quello del *commesso viaggiatore*; questo è dovuto al fatto che nonostante sia un problema semplicissimo da inquadrare, appartiene alla categoria dei problemi NP-Completi ossia i problemi per i quali non è conosciuto un algoritmo che riesca a risolverli in tempo polinomiale ma solo in tempi che crescono esponenzialmente con la dimensione del problema (in questo caso con il numero delle città) [4]. Proprio per questa caratteristica, l'utilizzo di algoritmi genetici che, come già visto, esplorano solo una parte ristretta dello spazio delle soluzioni ed affrontano il problema in maniera non deterministica può portare ad una buona approssimazione della soluzione in tempi accettabili. Lo stesso discorso vale anche per il problema che affrontiamo nel nostro lavoro, anch'esso un problema NP-Completo: la $k$-colorazione di un grafo [8].

# Capitolo 3

# Il Calcolo Parallelo

In questa sezione introdurremo le principali nozioni del calcolo parallelo, vedremo dove esso viene applicato maggiormente e descriveremo le possibili architetture di una macchina parallela. Il calcolo parallelo è una evoluzione del calcolo sequenziale ed in generale è possibile definirlo come l'utilizzo contemporaneo di più processori (o computer) per risolvere un unico problema; per far ciò è necessario che il problema venga diviso in sotto-problemi, che ciascuno di questi sia assegnato ad un processore diverso e che i processori riescano a comunicare e a scambiarsi informazioni. In generale, quindi, i processori eseguiranno le medesime istruzioni ma su sotto-problemi differenti. È possibile utilizzare il calcolo parallelo per due diversi principali scopi:

- risolvere un problema di maggiori dimensioni impiegando lo stesso tempo (*scale-up*)
- risolvere lo stesso problema impiegando un tempo minore (*speed-up*)

Un ruolo fondamentale nello sviluppo nel tempo del calcolo parallelo è occupato dalla legge di Moore; lo scienziato Gordon Moore infatti, nel 1965, ipotizzò quella che poi nel 1975 sarebbe diventata la famosa legge che prevedeva che ogni due anni, a parità di dimensioni, il numero di transistor per chip sarebbe raddoppiato. In realtà il periodo di due anni si è poi stabilizzato sui diciotto mesi, tuttavia questa legge è riuscita a mantenere intatta la

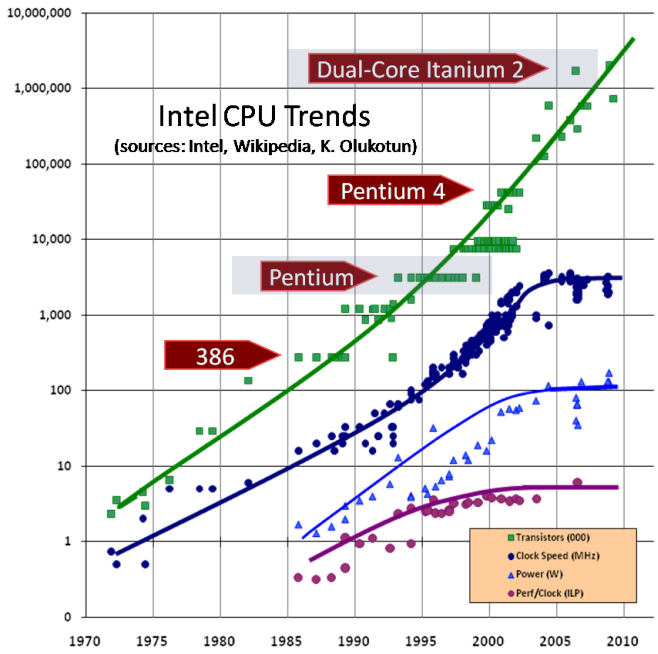


Figura 3.1: Grafico del numero di transistor per chip dagli anni '70.
Fonte: www.gotw.ca/publications/concurrency-ddj.htm

propria validità per oltre quarant'anni. È facile pensare che vi sono dei limiti per i quali questa legge è destinata a scomparire: la tecnologia utilizzata nel processo di produzione del silicio, la dimensione finita della materia (e quindi dei transistor) e la difficoltà nella gestione del surriscaldamento di transistor posti a distanza infinitesima tra loro.
Nella figura 3.1 è possibile visualizzare l'incremento del numero di transistor e della velocità del clock da quando Moore fece le prime ipotesi fino ai giorni nostri [11]. Come già spiegato, questo trend sembra destinato ad interrompersi.

## 3.1 Utilità e applicazioni

Il calcolo parallelo viene utilizzato principalmente per i cosiddetti *Grand Challenge Problems* ossia problemi di simulazione molto complessi; fanno

parte di questa categoria le simulazioni di reazioni nucleari, modellazioni del clima e previsioni del tempo, simulazioni di attività sismiche, lo studio della genetica e alcune applicazioni in campo militare. Nel caso delle previsioni del tempo, ad esempio, è necessario eseguire una grande mole di calcoli per simulare il comportamento atmosferico; tali calcoli, se eseguiti sequenzialmente, impiegherebbero una quantità di tempo che renderebbe inutile il lavoro di previsione ed è per tale ragione che accelerare i tempi mediante l'utilizzo del calcolo parallelo diventa essenziale. Il calcolo parallelo viene utilizzato anche in numerose applicazioni commerciali che richiedono l'elaborazione di enormi quantità di dati e l'esecuzione di algoritmi molto complessi come ad esempio i motori di ricerca, nell'ambito del data mining, programmi di grafica avanzata e di animazioni 3D eccetera. Nonostante la nascita del calcolo parallelo sia avvenuta nella seconda metà degli anni '80, esso si è sviluppato ed è stato utilizzato principalmente in ambito accademico e di ricerca. Questo è dovuto alla composizione di numerosi fattori:

- *Implementazione degli algoritmi.* Il calcolo parallelo implica la riscrittura (o per lo meno l'apporto di sostanziali modifiche) degli algoritmi sequenziali e questa è una fase molto delicata e difficoltosa a causa della complessità della programmazione parallela (spesso dipendente dall'architettura utilizzata). Esistono compilatori in grado di parallelizzare in modo automatico parti di codice sequenziale; si tratta però di sviluppi sperimentali che, almeno per il momento, non stanno dando risultati soddisfacenti.

- *Costo elevato.* I calcolatori paralleli, in generale, sono molto costosi e, componente ancora più importante, hanno delle elevate spese di gestione.

## 3.2 Architetture

Esistono numerose tipologie di architetture dei calcolatori ed in generale diversi approcci per classificarle [12]. La classificazione più utilizzata e di

maggior successo è quella nota come "Tassonomia di Flynn" dal nome dell'ingegnere informatico statunitense che la ideò; tale classificazione si basa sulla quantità di flussi di istruzioni che il sistema deve elaborare e sulla quantità di flussi di dati che deve esaminare e processare. Si distinguono quindi 4 principali categorie che andiamo ad analizzare brevemente.

- *SISD* (Single Instrucition Single Data). Si tratta di un'architettura sequenziale ed in particolare della classica *macchina di Von Neumann*; ciascun processore esegue uno stesso set di istruzioni (programma) su uno stesso flusso di dati.

- *SIMD* (Single Instruction Multiple Data). Ciascun processore esegue uno stesso set di istruzioni in maniera sincrona su più flussi di dati differenti.

- *MISD* (Multiple Instruction Single Data). Questa è una categoria teorica nel senso che non esistono macchine commerciali prodotte con questa tipologia di architettura; ciascun processore esegue differenti set di istruzioni su uno stesso flusso di dati.

- MIMD (Multiple Instruction Multiple Data). Ciascun processore può avere un programma diverso dagli altri processori ed eseguirlo su flussi di dati diversi.

Esiste un'ulteriore classificazione delle architetture dei calcolatori che si basa sulla modalità di gestione della memoria che queste utilizzano. Si differenziano le architetture

- *Shared-Memory*. I processori, pur lavorando in maniera indipendente dagli altri, accedono ad una stessa memoria e condividono le stesse risorse; ogni operazione di modifica di una cella della memoria fatta da un processore diventa valida anche per tutti gli altri. In generale viene utilizzata quando l'architettura è costituita da un numero limitato di processori. Vedi figura 3.2.

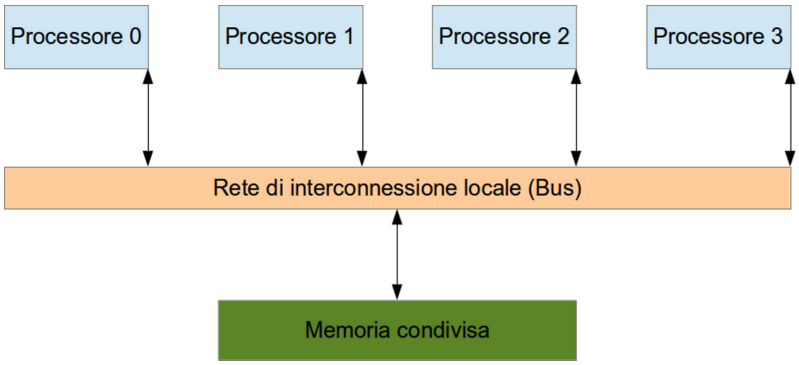


Figura 3.2: Struttura di un'architettura a memoria condivisa

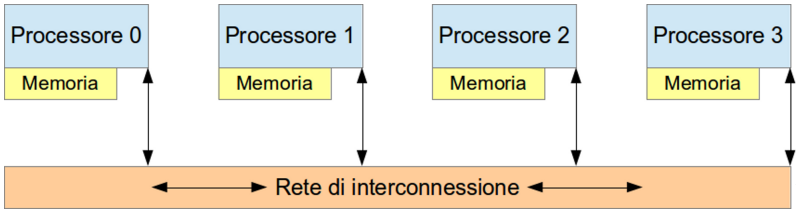


Figura 3.3: Struttura di un'architettura a memoria distribuita

- *Distributed Memory.* Ciascun processore ha una propria memoria locale alla quale nessun altro processore può accedere e i processori si scambiano dati ed informazioni utilizzando un'apposita infrastruttura di comunicazione la cui efficienza diventa un fattore determinante per la velocità di calcolo dell'intero sistema. Vedi figura 3.3.

Nella realtà esistono calcolatori che costituiscono una versione ibrida di quelli delle categorie appena analizzate dal momento che hanno un grande numero di nodi con una memoria condivisa e ciascun nodo è formato da un numero ristretto di unità di calcolo che dispongono di una propria memoria locale.

## 3.3 Calcolo della complessità

In questa sezione andiamo a definire le principali metriche di valutazione delle prestazioni di un algoritmo parallelo ossia lo *speedup* e l'*efficienza* [12]. Lo speedup è il rapporto tra il tempo $T_{seq}$ richiesto dall'algoritmo sequenziale

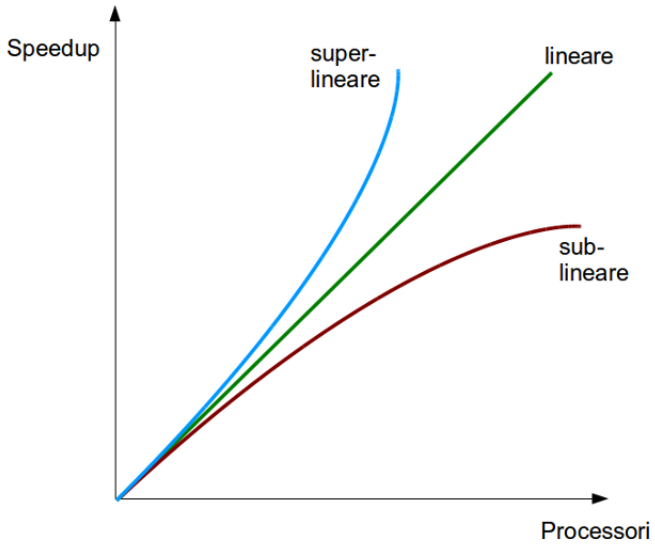


Figura 3.4: Grafico dello speedup

e quello $T_{par}(p)$ richiesto dalla sua versione parallela con un certo numero $p$ di processori [4]:

$$S(p) = \frac{T_{seq}}{T_{par}(p)}$$

É facile intuire che lo speedup ideale si ottiene quando il tempo richiesto dall'algoritmo parallelo è uguale al rapporto tra quello richiesto dalla sua versione sequenziale e il numero di processori utilizzato:

$$S(p)_{id} = \frac{T_{seq}}{T_{seq}/p} = p$$

In definitiva lo speedup ottimo deve essere uguale al numero di processori utilizzati e viene detto *speedup lineare* dal grafico che esso genera (vedi figura 3.4). Si tratta di un risultato quasi esclusivamente teorico e, in generale, lo speedup comune ottenuto da risultati sperimentali è il cosiddetto speedup sub-lineare ossia quello che al crescere del numero di processori cresce sempre più lentamente. Esistono casi in cui si riesce ad ottenere addirittura uno speedup super-lineare.

Un'altra metrica di valutazione delle prestazioni di un algoritmo parallelo è data dall'efficienza ossia dal rapporto tra lo speedup dell'algoritmo parallelo

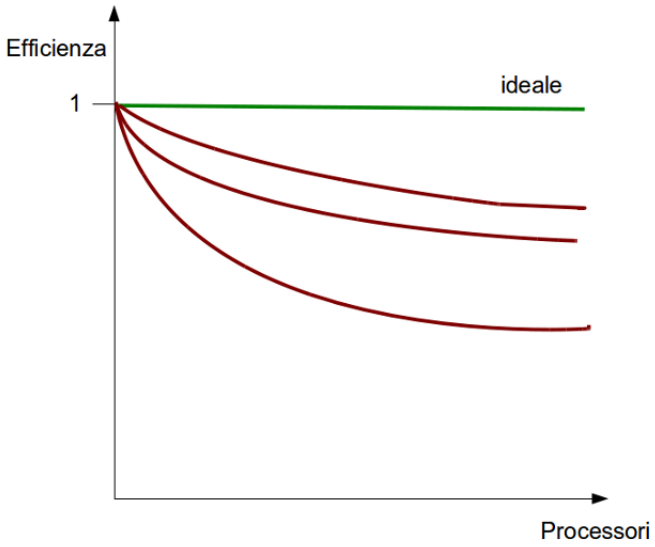


Figura 3.5: Grafico dell'efficienza

con un certo numero $p$ di processori ed il numero di processori stesso.

$$E(p) = \frac{S(p)}{p}$$

Dal momento che come abbiamo appena detto lo speedup ideale è uguale al numero di processori allora si deduce che l'efficienza ideale deve essere costante ed in particolare uguale a 1 (vedi figura 3.5).

$$E(p) = \frac{S(p)_{id}}{p} = \frac{p}{p} = 1$$

# Capitolo 4

# Algoritmo sequenziale

In questo capitolo parleremo dell'algoritmo sequenziale che ha costituito il punto di partenza per l'implementazione dell'algoritmo parallelo oggetto di questo lavoro. Si tratta di un algoritmo genetico [3], al quale sono state apportate alcune modifiche.

## 4.1 Rappresentazione del Problema

Innanzitutto analizziamo la fase di rappresentazione del problema. Come già accennato nei capitoli precedenti, per poter utilizzare un algoritmo genetico nella risoluzione di un problema è necessario riuscire a rappresentare ciascun individuo della popolazione con una stringa. Per quanto riguarda la questione della colorazione di un grafo, questa rappresentazione risulta abbastanza agevole: se si vuole colorare il grafo con $k$ colori basta rappresentare ciascun individuo con un array di lunghezza $n$, dove $n$ è il numero di nodi presenti nel grafo, ed assegnare a ciascuna cella (ogni cella rappresenta un nodo) un numero intero $c$ con $0 \leq c < k$ che coincide con l'identificatore del colore con il quale si vuole colorare il nodo. É possibile osservare un semplice esempio in figura 4.1.

Per rappresentare invece gli archi del grafo che connettono i nodi si è scelto di utilizzare una semplice matrice di adiacenza $M$ ossia una matrice $n \times n$

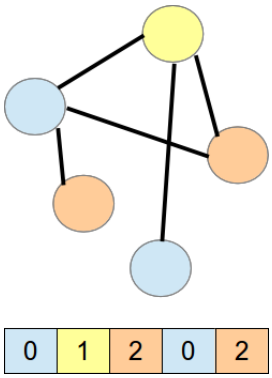


Figura 4.1: Esempio di rappresentazione con 5 nodi e 3 colori

con $n$ numero di nodi nel grafo, dove

$$M_{i,j} = \begin{cases} 1, & \text{se } \{i,j\} \in E \\ 0, & \text{se } \{i,j\} \notin E \end{cases}$$

## 4.2 Fitness

Il secondo passo fondamentale necessario per l'implementazione di un algoritmo genetico consiste nel trovare una funzione che possa rappresentare e calcolare in maniera adeguata il fitness di ciascun individuo della popolazione. Anche in questo caso, con il problema della colorazione di un grafo, è risultato abbastanza intuitivo scegliere come fitness il numero di conflitti che ciascun individuo presenta. Individui con fitness maggiore saranno considerati i peggiori mentre quelli con fitness minore i migliori. Se un individuo ha il proprio fitness uguale a 0 allora esso è soluzione del problema di partenza dal momento che nessun nodo ha lo stesso colore di un altro nodo con il quale è connesso da un arco.
Questa fase, in generale, risulta essere la più pesante dal punto di vista computazionale in un algoritmo genetico e questo vale in maniera ancora più evidente nel problema della colorazione di un grafo. Verificheremo quanto detto nei capitoli successivi.

## 4.3 Selezione

Nell'algoritmo che costituisce la base del nostro lavoro sono previste due distinte fasi di selezione delle quali però ne viene eseguite solo una per generazione. Denotiamo con $n$ il numero di individui della popolazione di partenza.

- *Selezione 1.* Viene eseguita quando il miglior fitness di tutti gli individui della popolazione risulta essere maggiore di 5 e consiste nel prendere in maniera casuale due coppie di individui dalla popolazione, confrontare i fitness degli individui di ciascuna coppia e selezionare i due individui *vincitori*.
- *Selezione 2.* Viene eseguita quando il miglior fitness di tutti gli individui della popolazione risulta essere minore o uguale di 5 ed è molto più semplice di quella precedente: consiste nel prendere l'individuo con fitness migliore e selezionarlo per due volte.

Indipendentemente da quale selezione venga eseguita il risultato della selezione sono due individui sui quali sarà effettuata la fase di crossover.

## 4.4 Crossover

La fase di crossover nel nostro algoritmo non presenta alcuna differenza sostanziale con quella generale presentata nei capitoli precedenti: i due individui selezionati nella fase appena conclusa, vengono *accoppiati* in un certo punto di crossover deciso in maniera casuale e danno origine a due nuovi individui con caratteristiche di entrambi quelli di partenza.

## 4.5 Mutazione

A questo punto dell'algoritmo, sugli individui risultanti dalla fase di crossover deve essere effettuata una mutazione. Anche qui, come per la fase della selezione, sono previste due distinte tipologie di mutazione.

- *Mutazione 1.* Viene eseguita quando il miglior fitness di tutti gli individui della popolazione risulta essere maggiore di 5 e consiste nel mutare tutti i colori che creano conflitti con dei colori che invece si è sicuri non ne creeranno. Questa tipologia di mutazione è un'altra delle fasi più pesanti dal punto di vista computazionale.

- *Mutazione 2.* Viene eseguita quando il miglior fitness di tutti gli individui della popolazione risulta essere minore o uguale a 5. Questo tipo di mutazione non fa altro che andare a mutare tutti i colori che creano conflitti con dei colori scelti in maniera random tra tutti quelli disponibili; potrebbe quindi succedere di cambiare un colore con un altro colore che comunque genera dei conflitti.

Il *coefficiente di mutazione*, in entrambi i casi, è fissato a 0.7. La mutazione costituisce l'ultima fase della parte *genetica* dell'algoritmo; tale fase viene ripetuta più volte per ciascuna generazione ed in particolare il numero di volte viene stabilito in base al miglior fitness di tutti gli individui. In definitiva, se il miglior fitness risulta essere minore o uguale a 5 si effettuano la prima tipologia di selezione, il crossover e la prima tipologia di mutazione per $n$ volte per ciascuna generazione. Alla fine di ogni generazione quindi, si avrà una popolazione composta da $(3 \times n)$ individui dei quali resteranno solo gli $n$ con i migliori fitness. Se invece, il miglior fitness risulta essere maggiore a 5 si eseguono la seconda tipologia di selezione, il crossover e la seconda tipologia di mutazione una sola volta ottenendo di conseguenza una popolazione di $(n+2)$ individui dei quali resteranno solo gli $n$ con i fitness migliori.

# Capitolo 5

# Algoritmo parallelo

Dopo aver descritto la struttura dell'algoritmo che si è deciso di parallelizzare, andiamo a presentare la sua versione parallela frutto di questo lavoro.

## 5.1 Sezione da parallelizzare

La prima questione da affrontare quando si decide di parallelizzare un algoritmo è individuare qual'è la sezione computazionalmente più pesante. Abbiamo più volte affermato, nei capitoli precedenti, che nel caso di un algoritmo genetico la maggior parte del tempo di esecuzione viene speso quasi sempre per il calcolo del fitness di ciascun individuo. Prima di procedere quindi, con la riscrittura della parte di codice candidata, si è deciso di verificare la veridicità di quanto appena affermato utilizzando come strumento di analisi *GProf* [15]; si tratta di un tool per sistemi Unix di analisi e di valutazione delle prestazioni di programmi. In particolare Gprof è un profiler che fornisce statistiche avanzate sui tempi di esecuzione di ciascuna routine del programma. Per utilizzare Gprof è necessario compilare il codice C con alcuni parametri ed eseguirlo normalmente; al termine dell'esecuzione viene generato un file che contiene tutte le informazioni riguardo ai tempi di esecuzione. Mostriamo in figura 5.1 il risultato dell'esecuzione del nostro algoritmo

implementato in modalità parallela ma eseguito in questa occasione in modalità sequenziale; come si può osservare il risultato è esattamente quello che ci si aspettava e oltre il 64% del tempo di esecuzione è stato impiegato per il calcolo del fitness di ciascun individuo.

Each sample counts as 0.01 seconds.

| % time | cumulative seconds | self seconds | calls | self us/call | total us/call | name |
|---|---|---|---|---|---|---|
| 64.11 | 22.98 | 22.98 | 213520 | 107.61 | 107.61 | calculateFitnessIndRoot |
| 34.40 | 35.31 | 12.33 | 58635 | 210.27 | 212.36 | mutate6 |
| 0.39 | 35.45 | 0.14 | | | | main |
| 0.34 | 35.57 | 0.12 | 41920 | 2.86 | 2.89 | selection1 |
| 0.25 | 35.66 | 0.09 | 14889315 | 0.01 | 0.01 | calculateRandom |
| 0.20 | 35.73 | 0.07 | 5555555 | 0.01 | 0.01 | sumArray |
| 0.14 | 35.78 | 0.05 | 41920 | 1.19 | 1.20 | crossover |
| 0.14 | 35.83 | 0.05 | 1097 | 45.61 | 45.61 | matrixToArray |
| 0.08 | 35.86 | 0.03 | 98 | 306.35 | 306.44 | mutate2 |
| 0.03 | 35.87 | 0.01 | 2194 | 4.56 | 4.56 | bestFitness |
| 0.00 | 35.87 | 0.00 | 83840 | 0.00 | 0.00 | calculateDoubleRandom |
| 0.00 | 35.87 | 0.00 | 1097 | 0.00 | 0.00 | makeMatrix |
| 0.00 | 35.87 | 0.00 | 1097 | 0.00 | 0.00 | trovato |
| 0.00 | 35.87 | 0.00 | 49 | 0.00 | 0.00 | bestEntity |
| 0.00 | 35.87 | 0.00 | 49 | 0.00 | 0.00 | selection2 |
| 0.00 | 35.87 | 0.00 | 31 | 0.00 | 61.94 | createPopulation |
| 0.00 | 35.87 | 0.00 | 2 | 0.00 | 0.00 | maxDegree |
| 0.00 | 35.87 | 0.00 | 1 | 0.00 | 0.00 | initializeTotal |
| 0.00 | 35.87 | 0.00 | 1 | 0.00 | 0.00 | initializeTotal1 |

Figura 5.1: Analisi dei tempi di esecuzione dell'algoritmo con Gprof

## 5.2 Parallelizzazione della popolazione

Dopo un'attenta analisi della funzione del calcolo del fitness abbiamo notato che i punti su cui si poteva intervenire per dividere il carico di lavoro erano essenzialmente due. Partiamo da quello più semplice dal punto di visto concettuale ossia quello della distribuzione della popolazione. In pratica si tratta semplicemente di dividere la popolazione per un certo numero di processori in modo che ciascuno calcoli in maniera indipendente il fitness degli individui che gli sono stati assegnati. Per quanto questa parallelizzazione possa risultare efficiente per un numero piccolo di processori, è facile intuire che non può rappresentare la nostra soluzione finale al problema dal

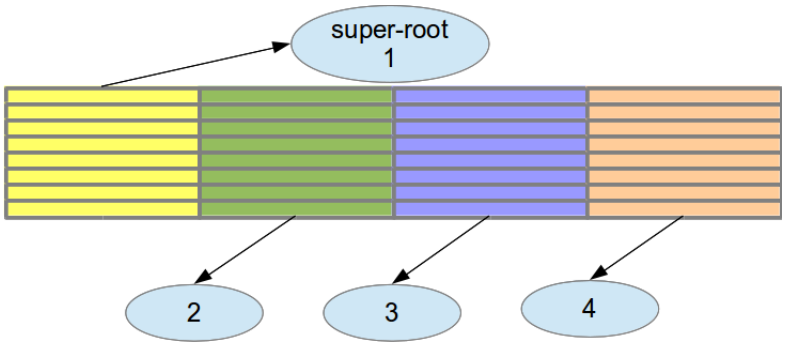


Figura 5.2: Distribuzione di una popolazione di 32 individui su 4 processori

momento che ha una scarsa scalabilità. Infatti, il numero di individui facenti parte della popolazione di partenza in un algoritmo genetico è un numero fisso deciso fin da subito dallo sviluppatore ed in generale, a causa dei soliti problemi tempistici, si tratta di un numero non troppo grande (di solito si utilizzano tra i 40 e i 100 individui). Di conseguenza questo significa che il numero di processori su cui è possibile distribuire la popolazione deve rimanere un numero ristretto.
Analizziamo un pò più nel dettaglio come avviene la parallelizzazione appena descritta. Chiamiamo *super-root* il processore che genera la popolazione iniziale; esso, giunto al punto di calcolare il fitness, distribuisce la popolazione sugli altri processori (compreso se stesso) e ciascuno di questi calcolerà il fitness degli individui inviando i risultati ottenuti nuovamente al *super-root* che a questo punto avrà il fitness di tutti gli individui della popolazione (vedi figura 5.2).

```
part = numIndividui / numProcessori;
// Scatter popolazione
MPI_Scatter(popolazione, part);
```

```
\\ ciascun processore calcola il fitness dei propri individui
calculateFitness();
```

```
part = numIndividui / numProcessori;
// Gather popolazione
MPI_Gather(fitnessLocale, part);
```

## 5.3 Parallelizzazione del singolo individuo

La seconda tipologia di parallelizzazione risulta essere molto più vicina alle nostre esigenze dal momento che è possibile utilizzare un numero di processori anche molto grande se si prende come input del problema un grafo di grandi dimensioni. Si tratta di fare in modo che i processori cooperino per il calcolo del fitness di ciascun individuo dividendosi questa volta, invece che la popolazione, la matrice di adiacenza da cui calcolare i conflitti di ciascun nodo del grafo.
Sia $M$ di dimensioni $n \times n$ la matrice di adiacenza di un grafo $G$ non orientato e sia $p$ il numero di processori che devono cooperare al calcolo del fitness di un certo individuo. In sequenziale, il singolo processore, avrebbe dovuto scorrere i nodi dell'individuo e andare a verificare (scorrendo la matrice di adiacenza) i conflitti di ciascun nodo. Si noti che, trattandosi di un grafo non orientato, la matrice di adiacenza è sempre simmetrica. L'idea della parallelizzazione è quella di far analizzare a ciascun processore solo una parte delle $\frac{n \times (n-1)}{2}$ celle totali della matrice. Per cercare di bilanciare il carico del lavoro sui processori si è deciso di assegnare a ciascun processore $\frac{n}{p}$ righe ed in particolare si assegnano tali righe a coppie prendendone una dall'alto della matrice ed una dal basso. In questo modo, supponendo di avere $n$ righe e $p = \frac{n}{2}$ processori, il generico processore $i$ si occuperà della riga $i$ e della riga $(n-i)$. In figura 5.3 vediamo l'esempio della divisione di una matrice $8 \times 8$ su 4 processori. Come si può osservare, ciascun processore andrà a controllare 7 celle delle matrice ed il carico di lavoro sarà equamente distribuito. Questa divisione ha però un limite: nel caso in cui il numero di righe $r$ (e quindi il numero di nodi del grafo) non fosse esattamente divisibile per il doppio del numero di processori, allora le righe in eccesso saranno gestite dal processore master; per fare il modo che il carico sia perfettamente bilanciato deve quindi risultare

$$r = 2p \times k$$

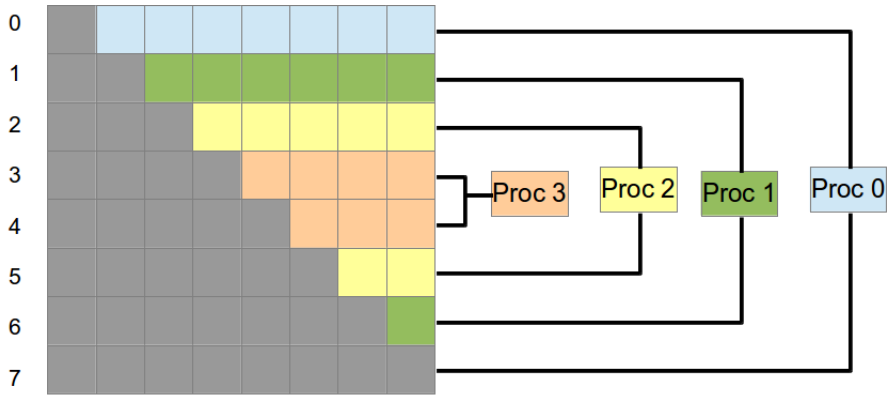


Figura 5.3: Distribuzione di una matrice 8x8 su 4 processori

```
MPI_Bcast(individuo, numNodes);
// ciascun processore calcola la propria parte
int conflittiParziali = calculateFitnessIndSub(individuo, numNodes);
// i conflitti trovati da ciascun processore vengono sommati
MPI_Reduce(&conflittiParziali);
MPI_Barrier(MPI_COMM_WORLD);
```

## 5.4 Parallelizzazione complessiva

La versione definitiva del nostro algoritmo implementa entrambe le tipologie di parallelizzazione presentate nelle sezioni precedenti. In definitiva si è deciso di utilizzare una popolazione con 40 individui e di dividere tale popolazione per un certo numero ristretto di master; ciascuno di questi master gestisce alcuni processori con i quali andrà a calcolare il fitness degli individui che gli sono stati assegnati (vedi figura 5.4).

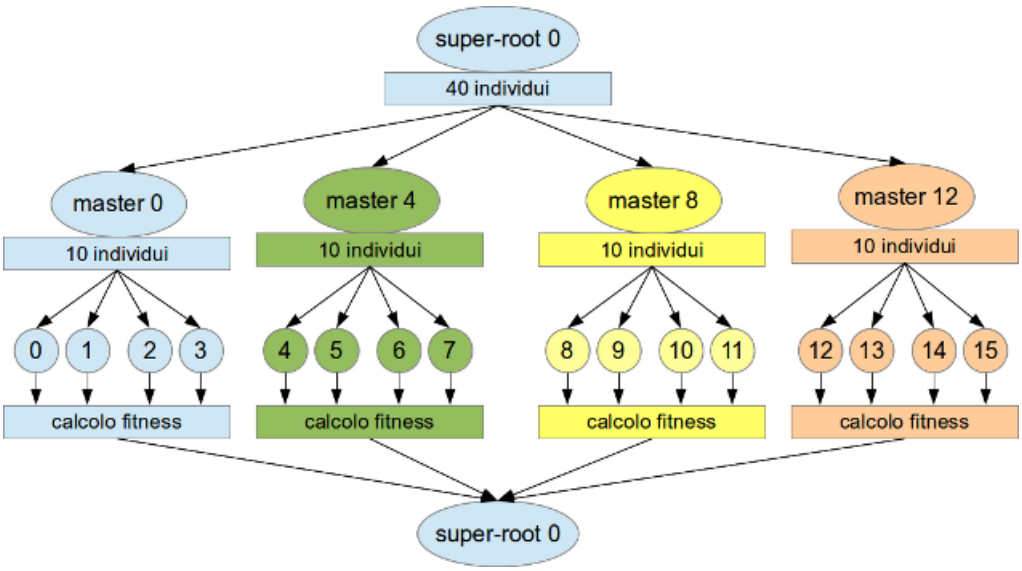


Figura 5.4: Parallelizzazione con 16 processori e 4 master

## 5.5 Validazione

Per validare il nostro algoritmo lo abbiamo eseguito prendendo in input alcuni grafi dei quali è noto il numero cromatico. Tali grafi fanno parte della collezione DIMACS (centro di matematica discreta e informatica teorica) [14], e sono stati utilizzati in numerosi altri lavori [3, 7]. Noi ne abbiamo selezionati 20 su ognuno dei quali sono stati eseguiti 5 run con una popolazione di 80 individui e per un totale di 5000 generazioni. I test sono stati tutti effettuati su una macchina con processore $intel^{®}\ Core^{tm}\ i7-2670QM$

che dispone di 4 core fisici. Di seguito la tabella con il numero cromatico atteso per ciascun grafo ed il risultato ottenuto (il minimo dei numeri cromatici ottenuti). Com'è possibile notare, in qualche caso non si è ottenuta la soluzione esatta e questo è dovuto alla natura euristica dell'algoritmo genetico; probabilmente con un numero di individui maggiore o con un numero di generazioni più alto si sarebbe ottenuto il risultato atteso.

| *File* | *Nodi* | *Archi* | $\chi(G)$ *atteso* | $\chi(G)$ |
|---|---|---|---|---|
| *anna.col* | 138 | 986 | 11 | 11 |
| *david.col* | 87 | 812 | 11 | 11 |
| *homer.col* | 561 | 3258 | 13 | 13 |
| *huck.col* | 74 | 602 | 11 | 11 |
| *jean.col* | 80 | 508 | 10 | 10 |
| *games*120.*col* | 120 | 1276 | 9 | 9 |
| *miles*250.*col* | 128 | 774 | 8 | 8 |
| *miles*500.*col* | 128 | 2340 | 20 | 20 |
| *miles*750.*col* | 128 | 4226 | 31 | 31 |
| *miles*1000.*col* | 128 | 6432 | 43 | 43 |
| *miles*1500.*col* | 128 | 10369 | 73 | 73 |
| *queen*5_5.*col* | 25 | 320 | 5 | 5 |
| *queen*6_6.*col* | 36 | 580 | 7 | 7 |
| *queen*7_7.*col* | 49 | 952 | 7 | 8 |
| *queen*8_8.*col* | 64 | 1456 | 9 | 10 |
| *mulsol.i*.1.*col* | 197 | 3925 | 49 | 50 |
| *mulsol.i*.2.*col* | 188 | 3885 | 31 | 31 |
| *mulsol.i*.3.*col* | 184 | 3916 | 31 | 31 |
| *mulsol.i*.4.*col* | 185 | 3496 | 31 | 31 |
| *mulsol.i*.5.*col* | 186 | 3973 | 31 | 31 |

Tabella 5.1: Risultati della validazione

# Capitolo 6

# MPI

Come osservato nei capitoli precedenti, i processori che fanno parte di un sistema a memoria distribuita hanno necessariamente bisogno di comunicare utilizzando un'apposita infrastruttura di comunicazione sulla quale viaggeranno i messaggi da un processore all'altro; tale infrastruttura può essere differente per ciascuna architettura ma tutte hanno bisogno di un metodo per definire i messaggi, la loro tipologia, il loro contenuto, informazioni come mittente e destinatari e così via. Per far ciò, si utilizza un protocollo standard di comunicazione chiamato *Message Passing Interface* (MPI) sviluppato espressamente per il calcolo parallelo. Non si tratta di una libreria di per sè ma soltanto di una specifica per librerie scritte nei linguaggi C/C++ e Fortran; tra le implementazioni gratuite segnaliamo OpenMPI [13].
L'obiettivo di MPI è fornire un paradigma standard per la programmazione parallela che sia pratico, portabile, efficiente e flessibile [14]. In particolare la portabilità costituisce una delle caratteristiche principali dal momento che il codice scritto con lo standard MPI non deve tener conto della tipologia della macchina su cui dovrà essere eseguito e di conseguenza può essere utilizzato su macchine anche molto differenti tra loro.

## 6.1 Stuttura del programma e principali funzioni

Analizziamo ora, con l'ausilio della figura 6.1, la struttura di un tipico programma che segue le specifiche MPI. La prima cosa da fare per poter utilizzare le funzioni MPI è quella di includere l'header file MPI con la seguente riga di codice

```
include "mpi.h"
```

A questo punto deve esser scritto tutto il codice del programma che deve essere eseguito in modo sequenziale e che quindi non necessità di nessuna funzione MPI. Sottolineiamo che tutto il codice viene eseguito da tutti i processori avviati (a meno di restrizioni definite espressamente dal programmatore ad esempio con blocchi *ITE*). Quando invece, si giunge alla parte che si è deciso di parallelizzare, bisogna inizializzare l'ambiente MPI, scrivere il codice parallelo con le opportune chiamate delle funzioni MPI, chiudere l'ambiente MPI e continuare con la normale scrittura del programma. L'inizializzazione e la chiusura dell'ambiente viene definita dalle seguenti due righe di codice al di fuori delle quali nessuna funzione MPI può essere richiamata:

```
MPI_Init(&argc, &argv);
...
MPI_Finalize();
```

Concentriamo ora la nostra attenzione sul blocco di istruzioni relative al codice da parallelizzare ed in particolare, restringiamo l'analisi delle funzioni MPI a quelle che realmente abbiamo utilizzato nell'implementazione del nostro algoritmo parallelo:

- `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`

  Salva nella variabile *rank* l'id del processore che ha chiamato la funzione.

- `MPI_Comm_size(MPI_COMM_WORLD, &size);`

  Salva nella variabile *size* il numero di processori attivi.

- `MPI_Barrier(MPI_COMM_WORLD);`

  Si tratta di un'operazione di sincronizzazione. Blocca i processori che hanno chiamato la funzione fino a quando anche tutti gli altri avranno effettuato la stessa chiamata.

- `MPI_Wtime();`

  Serve a calcolare i tempi di esecuzione del programma.

- `MPI_Bcast (&buffer,count,datatype,root,comm);`

  Con questa funzione il processore *root* invia un messaggio a tutti gli altri processori (vedi figura 6.2).

- `MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,`
  `recvcnt,recvtype,root,comm);`

  Si occupa di distribuire alcuni dati presenti sul processore *root* su tutti gli altri processori (vedi figura 6.2). Abbiamo utilizzato questa funzione nel nostro algoritmo per dividere la popolazione tra i processori master.

- `MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,`
  `recvcount,recvtype,root,comm)`

  Si occupa di unire alcuni dati presenti su diversi processori su un unico processore *root* (vedi figura 6.2). Abbiamo utilizzato questa funzione nel nostro algoritmo per unire i risultati del calcolo del fitness fatto in parallelo.

- `MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm);`

  Si tratta di un'operazione di computazione collettiva; in particolare applica l'operazione indicata come argomento sui dati presenti sui processori coinvolti e mette il risultato nella memoria del processore *root* (vedi figura 6.2).

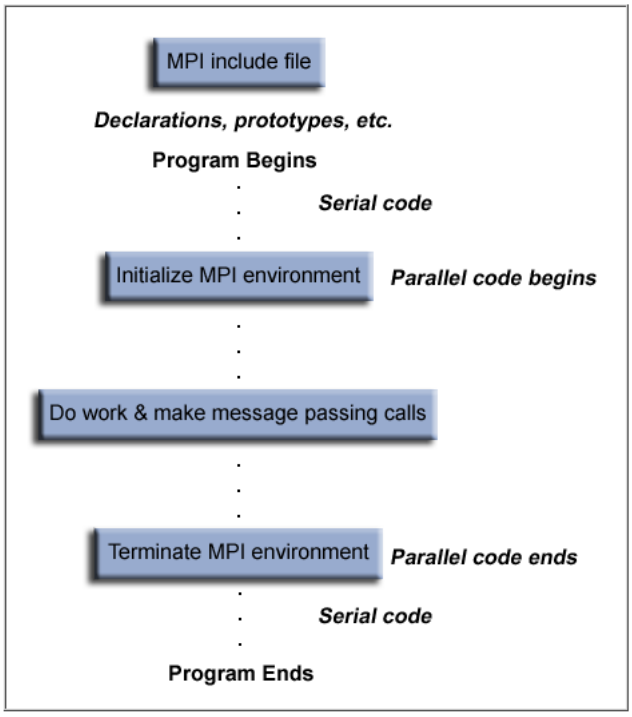


Figura 6.1: Struttura generale di un programma MPI [14]

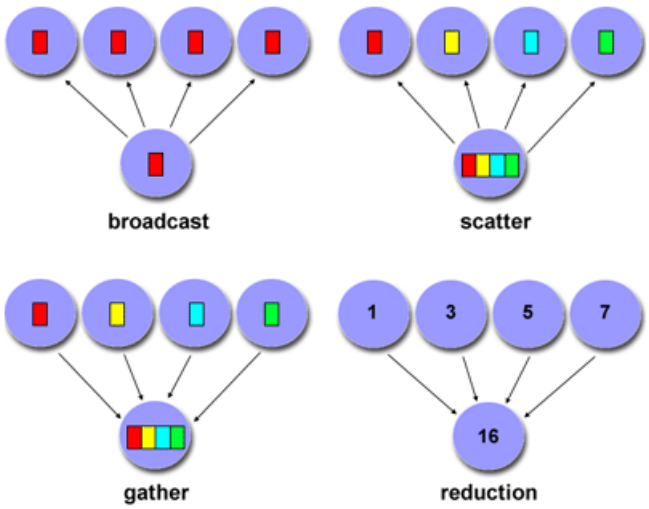


Figura 6.2: Principali funzioni MPI [14]

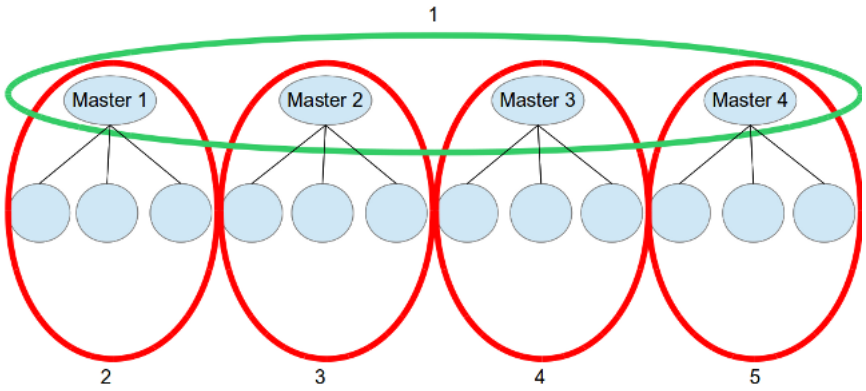


Figura 6.3: Esempio di comunicatori con 16 processori

## 6.2 Gruppi e Comunicatori

Molte delle funzioni MPI hanno bisogno di prendere come argomento un *comunicatore*. I comunicatori costituiscono una delle entità più importanti descritte dalle specifiche MPI e consistono in gruppi di processori all'interno dei quali essi possono scambiarsi messaggi. Il comunicatore più ampio presente in ogni programma MPI è *MPI_COMM_WORLD* che comprende tutti i processori avviati dall'utente.
Nel nostro programma è stato necessario definire altri comunicatori per far comunicare tra loro sotto-gruppi di processori; in particolare si è definito un comunicatore per i soli master e uno per ciascun master con i processori che esso deve gestire (vedi figura 6.3).

# Capitolo 7

# BlueGene/Q

A differenza dei test svolti per l'operazione di validazione, quelli riguardanti la fase sperimentale di questo lavoro non sono stati eseguiti su una macchina con un numero ristretto di core bensì su un sistema si calcolo a memoria distribuita basato su architettura *IBM BlueGene/Q* denominato *Fermi*. Si tratta di un calcolatore molto potente classificatosi al settimo posto tra i migliori 500 supercomputer del mondo nel 2012 e al dodicesimo posto nel 2013; si trova a Casalecchio di Reno (Bo), nella sede del Cineca (un consorzio formato da 69 università italiane e 3 enti tra i quali il Ministero dell'università e della ricerca altrimenti chiamato *MIUR*).

## 7.1 Architettura

Descriviamo brevemente l'architettura del *BlueGene/Q*. Il *Fermi* è composto da 10.240 nodi di calcolo ognuno dei quali dispone di 16 core che lavorano ad una frequenza di 1.6 GHz per un totale di 163.840 unità di calcolo. Ogni nodo dispone di 16 Gbyte di Ram DD3 che viene allocata in maniera statica in base ai core che si è deciso di utilizzare per ciascun nodo. I core del *BlueGene/Q* non sono particolarmente potenti ma il punto di forza di tutta l'architettura è costituito dalla infrastruttura di comunicazione interna 5D toroidale ad altissima velocità. Possiamo osservare uno schema dell'archi-

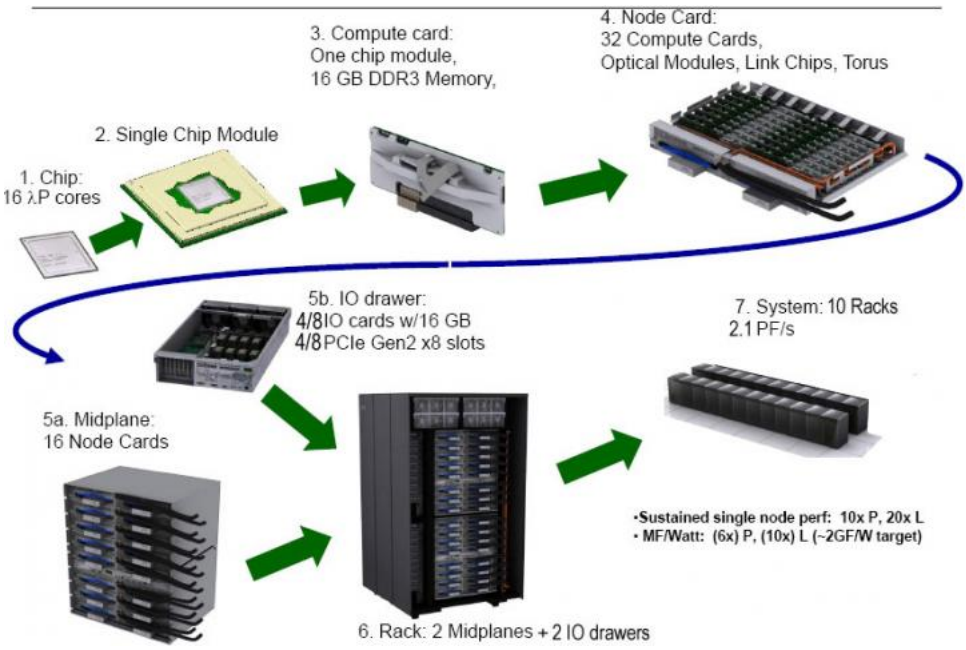


Figura 7.1: Architettura del Fermi BlueGene

tettura del calcolatore nella figura 7.1: i nodi di calcolo descritti sopra sono organizzati a blocchi da 32 detti *node card* che a loro volta sono organizzati a blocchi da 16 detti *midplane*. Questi ultimi sono collocati a coppie all'interno di un *rack* ed il sistema finale è costituito da 10 *rack*. In figura 7.2 è possibile vederne una foto reale.

## 7.2 Operazioni preliminari

Una volta ottenuto un account su *Fermi*, il primo passo per poter andare a svolgere la fase di sperimentazione è stato quello di copiare i file sorgenti e i file di input sulla macchina e di compilare il codice. A questo punto per mandare in esecuzione il programma è stato necessario scrivere dei file contenenti tutti i parametri necessari per l'esecuzione dei quali ne citiamo alcuni.

Figura 7.2: Fermi BlueGene

- Nome degli output. Si tratta di definire un nome identificativo per i file nei quali verranno salvati lo *standard output* del programma e lo *standard error*.

- Tempo limite superato il quale il programma sarà arrestato. Questo parametro è molto importante dal momento che determina in quale delle diverse code di attesa verrà collocato il job.

- Numero di nodi di calcolo da utilizzare. Questo parametro ha un limite inferiore di 64 nodi.

- Identificativo del proprio account.

- Indirizzo e-mail sul quale si desidera ricevere le notifiche riguardanti i propri job.

Come appena specificato, il numero minimo di nodi utilizzabili per l'esecuzione di un job è 64 e considerando che ciascun nodo dispone di 16 core il numero minimo di core risulta essere 1024. Questo può rappresentare un notevole spreco di risorse se, per esempio, si ha bisogno di eseguire l'algoritmo

utilizzando un numero ridotto di processori (ad esempio 1, 2, 4, 8); per evitare lo spreco di calcolo si è deciso di eseguire contemporaneamente più istanze del programma. Ad esempio, supponendo di voler eseguire 8 run con un solo processore, invece di allocare per 8 volte 1024 processori ed utilizzarne uno si sono eseguiti gli 8 run contemporaneamente con il risultato di un utilizzo di 1024 processori invece che di 8192. Questa modalità, detta a *sotto-blocchi* è stata utilizzata per tutte le configurazioni dove il numero di processori reali da utilizzare era sufficientemente basso.

# Capitolo 8

# Fase di Sperimentazione

La sperimentazione ha previsto tre differenti tipologie di parallelizzazione dell'algoritmo. Si è eseguito ciascun esperimento su due grafi distinti:

- *test256.col*. Nodi: 256, Archi: 1800, numero cromatico: 7.
- *test512.col*. Nodi: 512, Archi: 4000, numero cromatico: 8.

In tutti gli esperimenti il numero cromatico trovato è coinciso con quello atteso.

## 8.1 Legge di Amdahl

Prima di procedere alla presentazione dei risultati è doveroso capire quali sono quelli che è lecito aspettarsi dalla fase di test. Esiste una equazione detta *legge di Amdahl* con la quale è possibile calcolare il massimo speedup ottenibile da una parallelizzazione conoscendo la frazione $\alpha$ di tempo impiegata nella parte sequenziale e di conseguenza la percentuale $(1-\alpha)$ di tempo impiegata nella parte parallelizzata. L'equazione è la seguente:

$$S(p) = \frac{1}{\alpha + \dfrac{(1-\alpha)}{p}}$$

Per calcolare il parametro $\alpha$, come già spiegato nella sezione 5.1, si sono eseguiti una serie di test utilizzando il tool di profiling *GProf* ottenendo come risultato:

- *test256.col*: $\alpha = 0.36 \Rightarrow (1 - \alpha) = 0.64$
- *test512.col*: $\alpha = 0.34 \Rightarrow (1 - \alpha) = 0.66$

Andando a sostituire all'interno dell'*equazione di Amdahl* i valori ottenuti (analizziamo il caso del primo grafo) e facendo tendere a $+\infty$ il numero di processori è possibile calcolare il massimo speedup ottenibile:

$$S_{max} = \frac{1}{0.36 + \frac{(1 - 0.64)}{+\infty}} = \frac{1}{0.36} = 2.78$$

## 8.2 Configurazione 1

La prima configurazione di test prevede di mantenere sempre il numero di master uguale al numero di processori. Tali test quindi, prevedono la parallelizzazione della popolazione nelle modalità descritte nella sezione 5.2 ossia la popolazione viene distribuita equamente su tutti i processori ed ognuno di questi si occupa degli individui che gli sono stati assegnati. Per cercare di utilizzare un numero più alto di processori si è deciso di effettuare questi test con una popolazione di 80 individui mentre il numero di generazioni varia in base al grafo preso in input: 500 generazioni per *test256.col* e 350 generazioni per *test512.col*. I test sono stati effettuati a partire da un singolo processore fino a 16 processori. Nelle figure 8.1 e 8.2 vediamo lo speedup massimo teorico e lo speedup reale ottenuto distribuendo la popolazione equamente su tutti i processori.

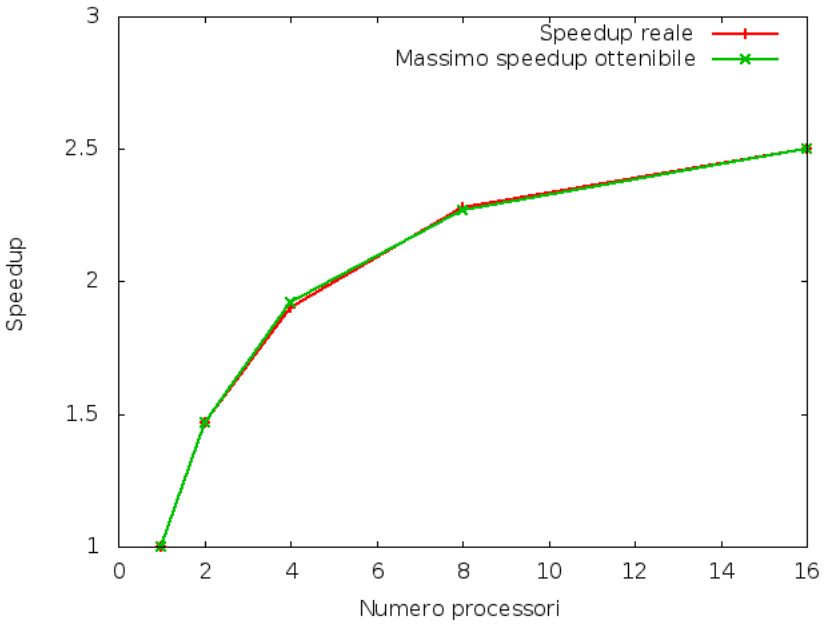


Figura 8.1: Speedup parallelizzazione popolazione su *test256.col*

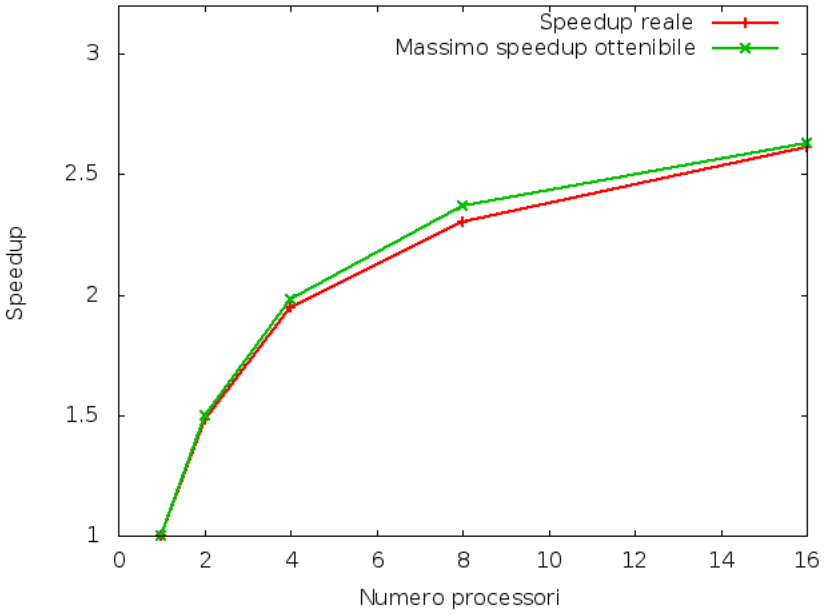


Figura 8.2: Speedup parallelizzazione popolazione su *test512.col*

## 8.3 Configurazione 2

La seconda configurazione invece, prevede di mantenere un solo master in tutti i test facendo così in modo che ogni processore lavori su tutti gli individui. Come già spiegato nella sezione 5.3 ogni processore calcola i conflitti parziali di ciascun individuo scorrendo solo la propria parte della matrice di adiacenza ed invia il risultato ottenuto all'unico master che ne fa la somma. In questo caso è stato possibile fare dei test più approfonditi dal momento che ci si è potuti spingere fino ad un numero di processori uguale al numero di nodi del grafo in input: 256 processori in *test256.col* e 512 processori in *test512.col*; è stata utilizzata una popolazione di 40 individui per 1000 generazioni. Nelle figure 8.2 e 8.4 vediamo lo speedup massimo teorico e lo speedup reale ottenuto mantenendo il numero di master a 1 e di conseguenza facendo in modo che ciascun processore calcolasse i conflitti parziali di tutti gli individui.

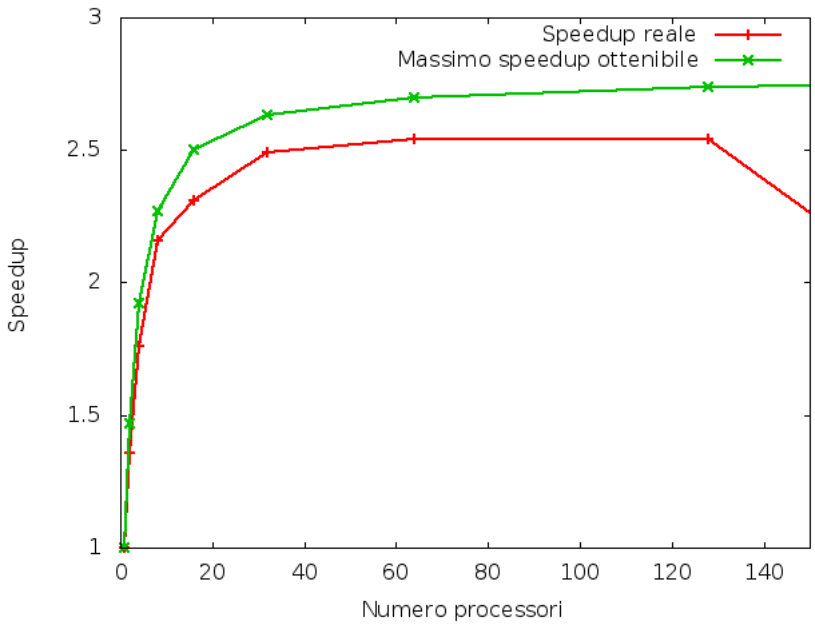


Figura 8.3: Speedup parallelizzazione singolo individuo su *test256.col*

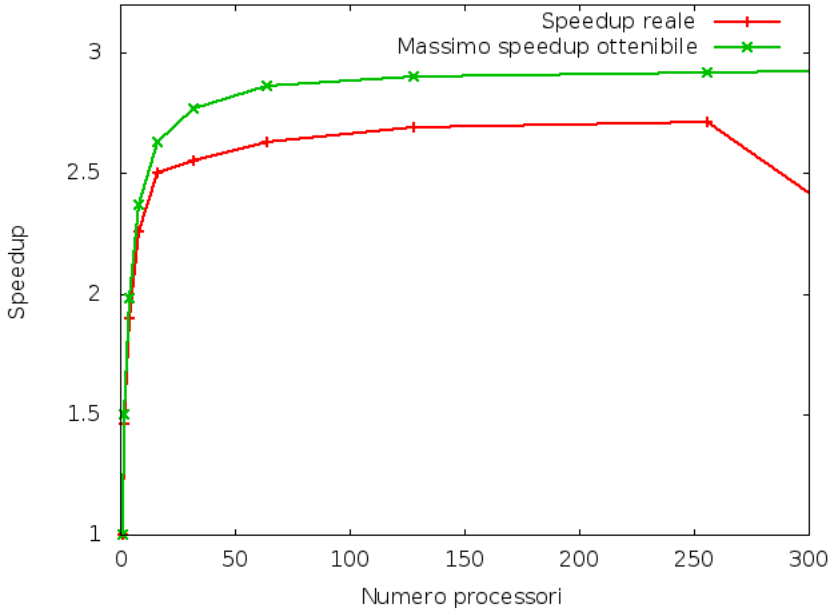


Figura 8.4: Speedup parallelizzazione singolo individuo su *test512.col*

## 8.4 Configurazione 3

L'ultima configurazione consiste nell'applicazione di quanto teorizzato nella sezione 5.4 ossia inizialmente si mantiene il numero di master uguale al numero di processori in modo da distribuire la popolazione; dopo un certo numero di test si mantiene invariato il numero di master e si aumenta il numero di processori in modo tale da parallelizzare i singoli individui sui sottogruppi gestiti dai master. L'idea è quella di riuscire ad utilizzare il maggior numero di processori mantenendo uno speedup vicino a quello massimo ottenibile. In questi esperimenti abbiamo utilizzato una popolazione di 40 individui per un numero di generazioni diverso in base al grafo in input: 1000 generazioni per *test256.col* e 500 generazioni per *test512.col*. Mostriamo nelle figure 8.5 e 8.6 il grafico dello speedup massimo ottenibile e di quello reale ottenuto distribuendo inizialmente la popolazione sui master ed in seguito facendo in modo che ciascun processore calcolasse i conflitti parziali di tutti gli individui assegnati al proprio master.

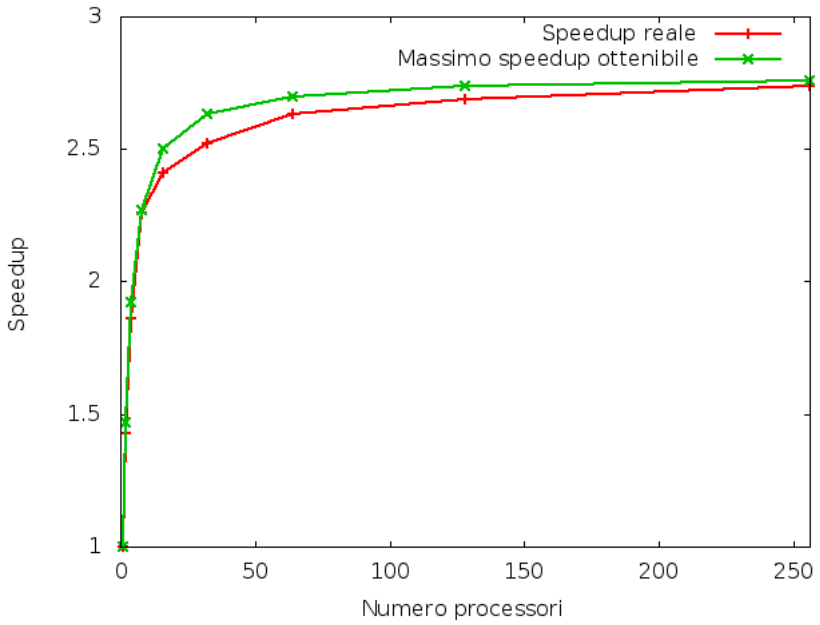


Figura 8.5: Speedup parallelizzazione combinata

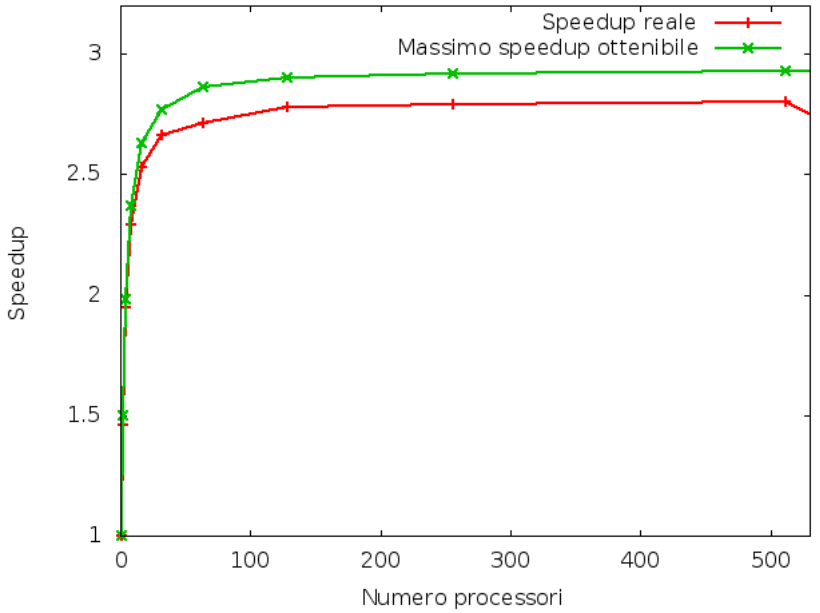


Figura 8.6: Speedup parallelizzazione combinata

## 8.5 Analisi delle prestazioni

In questa sezione analizziamo i risultati ottenuti e presentati nei grafici delle sezioni precedenti.
Per quanto riguarda la parallelizzazione che prevede la distribuzione della popolazione sui processori (soprattutto nel caso del grafo con 256 nodi) la curva praticamente si sovrappone a quella dello speedup ottimale e questo sta a significare che la parallelizzazione viene effettuata in maniera efficiente; purtroppo però, come già spiegato, questo tipo di parallelizzazione ha un limite molto forte nel numero di processori utilizzabili e di conseguenza è porta un vantaggio che è possibile sfruttare solo per un numero ristretto di processori.
I risultati ottenuti dai test effettuati facendo calcolare a ciascun processor solo parte del fitness di tutti gli individui ha portato a dei risultati leggermente meno positivi infatti le curve dei risultati reali si discostano in maniera più evidente da quelle degli speedup ottimali. Addirittura è possibile notare che se si utilizza un numero di processori pari alla metà del numero di nodi del grafo in input si ha un crollo della curva e si perde ogni vantaggio dovuto alla parallelizzazione: se si hanno 256 nodi allora i vantaggi restano apprezzabili fino all'utilizzo di 128 processori e, analogamente, se si hanno 512 nodi allora i vantaggi restano apprezzabili fino all'utilizzo di 256 processori. Tuttavia questo tipo di parallelizzazione presenta un importante vantaggio: è possibile apprezzare la riduzione dei tempi di esecuzione anche con un numero di processori discretamente elevato.
Passiamo ora alla parallelizzazione completa ossia quella che prevede l'utilizzo di entrambe le modalità di parallelizzazione appena descritte: distribuzione della popolazione e calcolo del fitness parziale di ciascun processore. In particolare si è deciso di parallelizzare seguendo il seguente schema:

| *Processori* | *Master* | *Tipologia* |
|---|---|---|
| 1 | 1 | *Sequenziale* |
| 2 | 2 | *Popolazione* |
| 4 | 4 | *Popolazione* |
| 8 | 8 | *Popolazione* |
| 16 | 8 | *Individuo* |
| 32 | 8 | *Individuo* |
| 64 | 8 | *Individuo* |
| 128 | 8 | *Individuo* |
| 256 | 8 | *Individuo* |
| 512 | 8 | *Individuo* |
| 1024 | 8 | *Individuo* |

Tabella 8.1: Schema della parallelizzazione definitiva

I risultati hanno dimostrato che realmente questa doppia modalità di parallelizzazione avvicina gli speedup ottenuti a quelli teorici e permette l'utilizzo di un numero di processori doppio rispetto a quello permesso dalla sola parallelizzazione del singolo individuo (i processori cooperano per calcolare il fitness di uno stesso individuo). Infatti questa volta il crollo della curva vi è stato con un numero maggiore rispetto al numero dei nodi del grafo in input. Concludiamo questo capitolo mostrando nelle figure 8.7 e 8.8 i grafici della bontà dei risultati ottenuti.

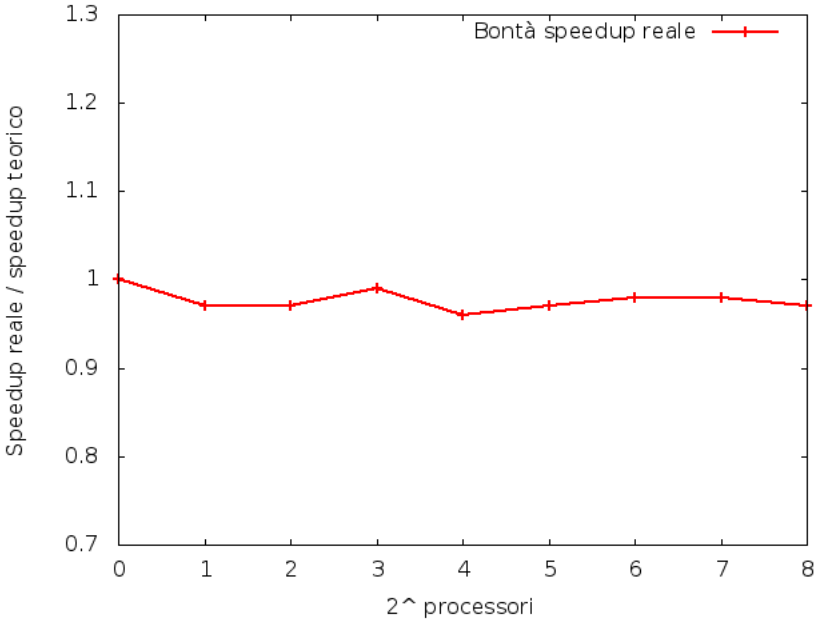


Figura 8.7: Bontà dei risultati su *test256.col*

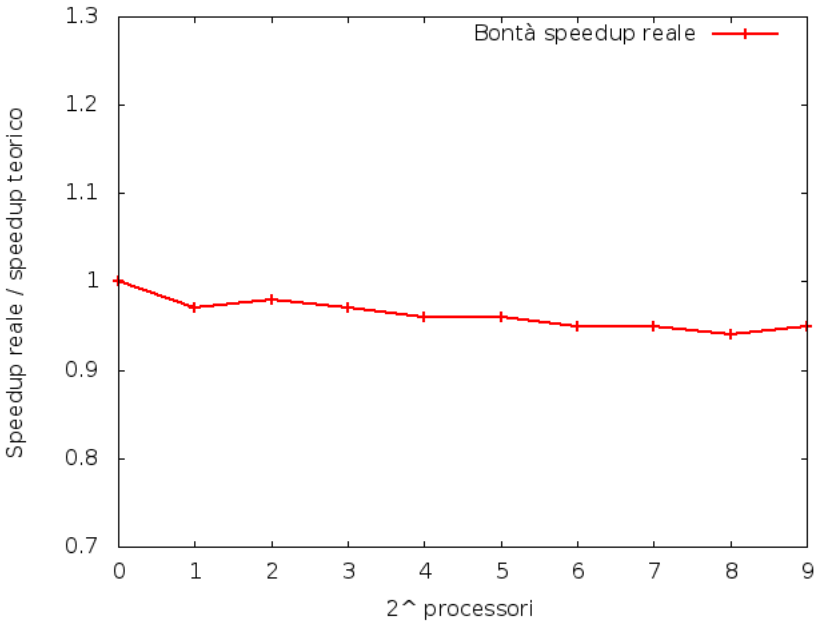


Figura 8.8: Bontà dei risultati su *test512.col*

# Conclusioni

Obiettivo di questo lavoro è stato quello di parallelizzare un algoritmo genetico per risolvere il problema della $k$-colorabilità. Dopo aver brevemente introdotto i principali concetti riguardanti la teoria dei grafi e gli algoritmi genetici abbiamo analizzato l'algoritmo sequenziale che ha costituito la base del nostro lavoro cercando di capire dove ed in che modo intervenire per riuscire a parallelizzarlo in modo opportuno ed efficace. Sono state considerate tre modalità di parallelizzazione e per ciascuna di queste, dopo averle descritte dal punto di vista teorico, sono stati effettuati numerosi test di valutazione dai quali si sono ottenuti risultati molto positivi; infatti si è riusciti ad ottenere dei valori sufficientemente prossimi a quelli che, secondo la *legge di Amdahl* rappresentano i valori massimi ottenibili. In questo lavoro ci si è occupati di parallelizzare solo la funzione del calcolo dei fitness degli individui che, come visto, occupa circa il 65% del tempo necessario all'esecuzione dell'intero programma. Vi è un'altra funzione, quella che riguarda la fase genetica della mutazione, che risulta occupare circa il 34% del tempo totale e di conseguenza, se implementata con la stessa efficacia, la parallelizzazione di questa fase potrebbe costituire un passo enorme verso speedup quasi lineari. Tutta la fase di testing è stata effettuata sul *BlueGene/Q Fermi* situato presso la sede del Cineca e per tale ragione si è potuto eseguire il programma su un numero elevato di processori (fino a 1024).

# Bibliografia


[1] R. Diestel, *Graph theory*, Springer-Verlag, 2005

[2] Tero Harju, *Lecture Notes on Graph Theory*, Department of Mathematics University of Turku, Turku, 1994-2011

[3] Musa M. Hindi and Roman V. Yampolskiy, *Genetic algorithm applied to the graph coloring problem*, Speed School of Engineering, Lousville Kentucky, 2012

[4] Alan A. Bertossi, *Algoritmi Paralleli*, Pitagora Editrice, Bologna, 2009

[5] Giorgio Ausiello, Fabrizio d'Amore, Giorgio Gambosi, *Linguaggi-Modelli-Complessità*, Franco Angeli, Roma, 2002

[6] Garey, M. R. and Johnson, D. S., *Computers and Interactability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[7] Reza Abbasian, Malek Mouhoub, *A hierarchical parallel genetic approach for the graph coloring problem*, Springer, New York, 2013

[8] Richard M. Karp, Reducibility Among Combinatorial Problems in R.E. Complexity of Computer Computations, New York, Plenum, 1972

[9] David E. Goldberg, *Genetic Algorithms in Search Optimization and Machine Learning*, Addison-Wesley, Reading-Massachusetts, 1989

[10] Stuart Jonathan Russell, Peter Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, New Jersey, 2010

[11] www.gotw.ca/publications/concurrency-ddj.htm

[12] C. Xavier, S. S. Iyengar, *Introduction to parallel algorithms*, Series Editor, United States of America, 1998

[13] www.open-mpi.org/

[14] https://computing.llnl.gov/tutorials/mpi/

[15] www.cs.utah.edu/dept/old/texinfo/as/gprof.html

[16] http://dimacs.rutgers.edu/

# Ringraziamenti

Rieccoci... Nuova laurea e nuovi doverosi ringraziamenti.
Sono stati due anni entusiasmanti e ricchi di motivazioni, con alti e bassi ma sicuramente positivi. Mi sento estremamente orgoglioso di quanto fatto e dei risultati ottenuti e, essendo assolutamente consapevole dell'enorme impegno che tutto ciò ha comportato, vorrei prima di tutto ringraziare per una volta me stesso, la mia costanza, la mia caparbietà e la mia totale incapacità di fermarmi di fronte alle difficoltà.
Grandi qualità, certo, ma tutte probabilmente inutili senza il supporto della persona che è stata al mio fianco tutti i giorni sostenendomi incondizionatamente. Non credo avrei mai potuto desiderare di meglio. Per questo e tutto quello che verrà, grazie Laura.
Questo importante traguardo, in una buona parte, è merito dei miei genitori per anni di insegnamenti e di sacrifici. Non sto parlando di sacrifici economici, che per quanto indubbiamente siano stati ingenti, credo siano niente rispetto al sacrificio fatto per rinunciare alla mia costante presenza a casa. Qualunque cosa mi riserverà il futuro sarò loro per sempre grato per avermi fatto realizzare questo sogno liberandomi da numerose responsabilità.
L'università di Bologna mi ha regalato la possibilità di conoscere ragazzi eccezionali con molti dei quali si è instaurato un rapporto di sincera amicizia oltre che di grande collaborazione. É motivo di vanto per me e soprattutto è stato uno stimolo determinante l'esser stato affiancato dai migliori.
Ringrazio i miei parenti e i miei amici di sempre, che considero delle certezze nella mia vita e che spero continueranno a dimostrarmi, come hanno

fatto anche in questa occasione, il loro affetto e la loro stima nei miei confronti. Concludo ringraziando mio fratello a cui è dedicato questo lavoro e senza il quale sarei sicuramente stato una persona peggiore. Questa laurea è soprattutto sua.