ALGORITMI GENETICI PARALLELI PER LA SOLUZIONE DEL TSP

R. Baraglia, M. Bucci, D. Circelli, R. Perego
CNUCE, Istituto del Consiglio Nazionale delle Ricerche, Pisa, Italia

9 ottobre 1996

Sommario

Il lavoro descritto in questo rapporto riguarda lo studio, il progetto e l'implementazione su una macchina ad elevato parallelismo, un *multicomputer* nCUBE2 con 128 nodi di elaborazione, di *algoritmi genetici*. Il problema usato come caso di studio è il classico problema del commesso viaggiatore (TSP), che per la sua semplicità formale ben si presta ad essere usato come *benchmark* per la valutazione di euristiche. Allo scopo di effettuare uno studio preliminare sui diversi algoritmi genetici descritti in letteratura, sono stati dapprima implementati e valutati AG sequenziali standard. Tali implementazioni hanno anche permesso di scegliere gli operatori genetici più promettenti ed ottimizzare il loro funzionamento per il caso di studio trattato. Sono stati quindi implementati e valutati AG paralleli a *grana fine* e a *grana grossa* al variare di alcuni parametri. Per l'algoritmo genetico a *grana fine* si è adottata una tecnica di *mapping* della popolazione sui processori che ha permesso di rendere indipendente la dimensione della popolazione dal numero di nodi dell'elaboratore usato.

Indice

1 Algoritmi genetici

Gli algoritmi genetici appartengono alla classe degli **algoritmi di approssimazione** o **euristici** usati per risolvere problemi **NP** [1], tra cui anche quelli di ottimizzazione combinatoriale [10]. Gli algoritmi genetici sono algoritmi stocastici nei quali la ricerca delle soluzioni è condotta imitando il comportamento degli organismi viventi che hanno una riproduzione sessuata; si generano delle **popolazioni** di soluzioni e si selezionano i migliori individui, analogamente a quanto avviene in natura secondo la teoria evolutiva di Darwin. Inoltre vengono sfruttati i concetti di ereditarietà e sopravvivenza dell'individuo più adatto. In natura, per la maggior parte degli organismi viventi, l'evoluzione avviene attraverso due processi fondamentali: la selezione naturale e la riproduzione sessuale. La prima determina quali elementi di una popolazione sopravvivono per riprodursi, la seconda garantisce il rimescolamento e la ricombinazione dei geni dei loro discendenti. Quando spermatozoo e uovo si uniscono, i cromosomi corrispondenti si allineano, poi si incrociano e si separano scambiandosi materiale genetico. Questo rimescolamento consente una evoluzione molto più rapida di quella che si avrebbe se tutti i discendenti contenessero semplicemente una copia dei geni di un unico genitore, modificata di tanto in tanto per mutazione.

Per poter impiegare gli algoritmi genetici (nel seguito useremo l'abbreviazione AG) è necessario innanzitutto stabilire una codifica della soluzione del problema trattato in termini di stringhe binarie (dalla definizione originaria di Holland che vedremo nel prossimo Paragrafo); la ricerca di una buona soluzione diventa quindi la ricerca di particolari stringhe binarie [12].

Si parte da una popolazione iniziale di **individui**, ciascuno dei quali mantiene una soluzione codificata nel suo DNA (stringa binaria): valutando la bontà della soluzione fornita da ciascun *individuo* mediante l'applicazione di una **funzione costo**[2], si selezionano gli individui che si dovranno riprodurre. La riproduzione delle stringhe avviene, in analogia con gli esseri viventi a riproduzione sessuata, tramite **accoppiamento**, cioè tramite la fusione delle due stringhe appartenenti ai genitori. In pratica ciò si realizza stabilendo un punto in cui spezzare in due ciascuna delle stringhe, per accoppiare poi ciascuna metà con la metà complementare dell'altra stringa. I discendenti così prodotti non vanno a rimpiazzare i genitori, che, ricordiamolo, sono le stringhe che hanno fornito i migliori valori per la funzione costo; essi rimpiazzano invece le stringhe della precedente generazione che hanno dato valori meno buoni della funzione costo. In questo modo la popolazione globale non cambia ad ogni generazione. Quindi, al fine di evitare la produzione di popolazioni troppo uniformi (che esplorano una porzione ristretta dello spazio delle soluzioni), si produce su una piccola percentuale di individui una **mutazione** (per esempio negando il valore di un bit della stringa).

2 Funzionamento degli AG

Gli AG non sono legati alle particolari caratteristiche del problema a cui sono applicati. Essi forniscono ottimi locali e attualmente non è disponibile nessuna informazione teorica sull'errore della soluzione trovata rispetto all'ottimo globale.

Ogni parametro del problema da ottimizzare è codificato in una stringa di bit, che sono visti come un **gene**. L'insieme dei parametri è codificato in una stringa di bit, che sono visti come un **cromosoma**. L'algoritmo non ha nessuna conoscenza sul significato della stringa di bit.

Gli algoritmi genetici non lavorano a partire da un singolo punto nello spazio di ricerca, ma da una popolazione di individui. Negli algoritmi genetici ogni soluzione è vista come un individuo della popolazione, l'insieme di soluzioni all'istante t viene detto **popolazione**. Ad ogni individuo è assegnato un valore di **fitness**[3]. Quest'ultimo non deve riguardare solo un attributo di un particolare gene, piuttosto la combinazione totale dei geni.

[1] Si dicono **NP** quei problemi decisionali risolubili in tempo polinomiale da un algoritmo non deterministico.

[2] Con il termine *funzione costo* si indica una funzione che, data una soluzione ammissibile del problema, restituisce un valore che ne rappresenta la "bontà".

[3] La *fitness* indica quanto è buona la soluzione fornita dall'individuo. Tipicamente, coincide con il valore che si ottiene calcolando la funzione costo sulla soluzione.

Gli algoritmi genetici sono stati introdotti da Holland in [9]. L'algoritmo presentato da Holland è riportato in Figura 1. Con P_C si indica la probabilità di applicare l'operatore di crossover, con P_M la probabilità di mutare i geni dell'individuo trattato e con P_I la probabilità di invertire la sequenza dell'intero codice genetico di un individuo.

L'algoritmo di Holland funziona come segue. Sia $\beta(0)$ la popolazione iniziale generata casualmente e $\beta(t)$ la popolazione di m individui all'istante t, l'algoritmo iterativamente genera una nuova popolazione $\beta(t+1)$ a partire da $\beta(t)$ applicando alcuni operatori genetici. Questi operatori modificano alcuni individui nella popolazione $\beta(t)$ ottenendo così una popolazione $\beta(t+1)$ che contiene un maggior numero di individui che hanno un valore di $fitness$ migliore.

La condizione di terminazione può essere fissata da:

- un numero massimo di iterazioni, raggiunto il quale l'algoritmo si ferma;
- un controllo sul miglioramento dei valori di $fitness$ forniti dalle soluzioni correnti; quando si notano miglioramenti poco apprezzabili delle soluzioni, l'algoritmo è interrotto.

Nei paragrafi successivi è descritto ogni singolo passo dell'algoritmo.

3 Operatori degli AG

3.1 Selezione

L'operatore di selezione ha la funzione di scegliere gli individui ai quali devono essere applicati gli operatori genetici. La selezione degli individui può essere fatta casualmente, oppure per mezzo del valore di $fitness$. In quest'ultimo caso gli individui con valore di $fitness$ migliore hanno una probabilità più alta di essere selezionati per generare i figli.

Questa operazione di selezione degli individui con valore di $fitness$ più alto, può essere vista come una metafora della sopravvivenza naturale di Darwin. Infatti in natura il valore di $fitness$ di un individuo è determinato dall'abilità di superare tutti gli ostacoli dell'ambiente.

La fase di selezione è applicata sulla *popolazione corrente*. Le stringhe selezionate sono messe in una *popolazione intermedia*. Gli operatori genetici sono applicati alla popolazione intermedia, per creare la *popolazione successiva*. Ottenuta quest'ultima si dice allora che nell'esecuzione dell'algoritmo genetico è trascorsa una generazione.

3.2 Rimpiazzamento

Ci sono due modi principali per rimpiazzare un individuo nella popolazione:

- *rimpiazzamento generazionale*;
- *rimpiazzamento a stato continuo*.

Nel *rimpiazzamento generazionale* la popolazione corrente è rimpiazzata dalla successiva; in questo caso le dimensioni della popolazione corrente e della successiva devono essere uguali.

Nel *rimpiazzamento a stato continuo*, i nuovi individui generati rimpiazzano gli individui peggiori della popolazione corrente quando i figli hanno ottenuto un valore di $fitness$ migliore. La popolazione successiva è identica alla popolazione corrente eccetto per l'ultimo figlio generato. In letteratura questo modello è conosciuto come *modello statico di popolazione*.

Un'importante differenza tra i due metodi è che nel *rimpiazzamento a stato continuo* i nuovi figli sono immediatamente disponibili per la riproduzione, perciò gli AG hanno l'opportunità di esplorare subito gli individui più promettenti appena essi vengono creati. Questi due modelli di rimpiazzamento vengono impiegati coerentemente col modello di generazione adottato: il *rimpiazzamento generazionale* col *modello di generazione discreto* e il *rimpiazzamento a stato continuo* col *modello di generazione continuo*. I modelli di generazione sono descritti nel Paragrafo 4.

3.3 Crossover

L'operatore di *crossover* prende parti dei due genitori e crea un nuovo individuo (figlio) combinando le parti prese. In questo modo i figli contengono informazioni genetiche di entrambi i padri.

I punti di *crossover* (di solito due) sono scelti casualmente. I nuovi individui consisteranno di parti alternate delle stringhe dei genitori.

Nonostante sia casuale, questo scambio di informazione genetica rende gli algoritmi genetici molto versatili; infatti una parte buona di un genitore può rimpiazzare una parte meno buona dell'altro genitore, creando un figlio migliore dei genitori.

3.4 Inversione

L'operatore di inversione prende due punti casuali nel cromosoma e inverte la sequenza di geni tra questi due punti. L'importanza del cambiamento così effettuato dipende dalla particolare codifica scelta per rappresentare la soluzione.

3.5 Mutazione

L'operatore di mutazione è molto semplice. Esso cambia casualmente i valori di un gene in un individuo appena creato; nel caso della codifica binaria della soluzione, si nega il valore di un bit. È impiegato principalmente per introdurre delle varianti nelle soluzioni della popolazione corrente, che consentano di sfuggire dagli ottimi locali quando la maggioranza delle soluzioni converge verso questi ultimi.

4 Modello di generazione

Il modello di generazione descrive il modo secondo il quale la popolazione si evolve. Come risulta da [8], esistono due modelli di generazione: il **modello di generazione discreto**, e il **modello di generazione continuo**.

Nel *modello di generazione discreto* si tiene ogni generazione ben distinta dall'altra. Infatti la generazione successiva $\beta(t+1)$, formata dai figli, è generata dalla popolazione dei genitori $\beta(t)$.

In questo modo riusciamo a distinguere le popolazioni di genitori da quella dei figli. Se tutti i genitori hanno prodotto i loro figli, questi ultimi diventeranno i nuovi genitori della successiva generazione e così via.

Nel *modello di generazione continuo* la decisione sulla sopravvivenza di un figlio viene fatta immediatamente dopo la sua creazione, anzichè dopo la generazione di tutti gli altri figli. I figli diventano immediatamente disponibili come genitori. Quindi non è possibile distinguere nella generazione corrente tra genitori e figli.

5 Aspetti teorici degli AG

Gli aspetti teorici trattati nei seguenti paragrafi si basano sulla definizione originaria di AG dovuta a Holland [9]. Se si immagina di disporre nello spazio tutte le stringhe binarie di lunghezza n possibili, ponendo più vicine quelle che differiscono per il minor numero di bit, si possono individuare alcune *regioni* che forniscono risulati migliori di altre per la funzione costo.

Per esempio, su una stringa di 4 bit, si potrebbe trovare che la regione identificata da 1*0* (dove * sta per 0 o 1 indifferentemente) fornisce in genere buoni valori per la funzione costo.

Gli algoritmi genetici riescono a campionare stringhe appartenenti a varie regioni, e attraverso la selezione degli individui migliori, si concentrano attorno alle regioni più promettenti.

Il numero di stringhe appartenenti a regioni con un buon valore della funzione costo aumenta di generazione in generazione: la fusione di stringhe (che avviene tramite l'operatore di *crossover*) può anche trasferire un discendente da una regione all'altra. Ciò si riflette in un rallentamento dell'evoluzione verso una buona soluzione.

La probabilità che i discendenti di due stringhe si allontanino dalla regione dei genitori dipende dalla distanza tra gli 0 e gli 1 che definiscono quella regione. I discendenti di una stringa della regione 1**0 rischiano di allontanarsi da tale regione indipendentemente da dove si situi il punto di separazione, mentre i discendenti della regione 10** hanno migliori possibilità di rimanere accanto ai loro genitori. I blocchi di 0 e 1 adiacenti in una regione si chiamano **blocchi costitutivi compatti**.

L'importanza dei *blocchi costitutivi compatti* deriva dal fatto che è molto probabile che essi rimangano intatti durante il processo di fusione di stringhe e quindi che si ritrovino in generazioni future proporzionalmente alla bontà della funzione costo posseduta dalle stringhe che li trasportano.

Il meccanismo fin qui illustrato riesce a campionare le regioni proporzionalmente alla bontà della loro funzione costo, solo in presenza di regioni definite da blocchi compatti.

Negli organismi viventi esiste un processo chiamato inversione, il quale ridispone occasionalmente i geni in maniera tale che quelli che si trovano più distanti nei genitori si ritrovino più vicini nella prole. Ciò corrisponde a ridefinire un blocco costitutivo in modo che sia meno soggetto alla separazione durante la fusione di stringhe. Ciò consente di perdere un minor numero di discendenti, e quindi di concentrarsi adeguatamente sulle zone più promettenti dello spazio delle stringhe.

6 Algoritmi genetici paralleli

La maggiore disponibilità di elaboratori ad alto parallelismo ha reso possibile la definizione e l'utilizzo di AG paralleli con lo scopo di diminuire il tempo di esecuzione ottenendo la stessa qualità delle soluzioni.

Vari modelli di parallelizzazione sono elencati in [4], e sono suddivisi in modelli:

- **a grana fine**: si opera su una singola popolazione in cui ogni individuo è posizionato in un elemento della topologia della struttura di interconnessione usata. Gli operatori genetici vengono applicati solamente tra individui vicini;

- **a grana grossa**: varie sottopopolazioni si evolvono in parallelo, scambiandosi periodicamente gli individui che forniscono soluzioni migliori;

7 Modelli di parallelizzazione

7.1 Algoritmi genetici paralleli a grana fine

Questo modello sfrutta i concetti di **spazialità** e di **vicinanza**. Il primo termine prevede che la popolazione di individui sia distribuita secondo una topologia di interconnessione logica, mentre il secondo specifica che l'accoppiamento e la selezione avvengono solamente tra individui vicini nella struttura topologica utilizzata. Un esempio di vicinanza è riportato in Figura 2.

L'insieme dei vicini di un individuo è detto il suo **intorno** e determina tutti i suoi potenziali **partner** per la riproduzione. È importante notare che, secondo il modello di vicinanza tra individui utilizzato, uno stesso individuo può appartenere contemporaneamente a più intorni dei suoi vicini, e quindi può accoppiarsi indifferentemente con ciascuno di essi (Figura 3).

Le comunicazioni tra i nodi devono avvenire rispettando il modello logico di vicinanza, cioè devono intercorrere solamente tra nodi logicamente vicini.

Possiamo sintetizzare le proprietà principali del modello a grana fine nei seguenti punti:

- nessun controllo centralizzato;

- iterazioni solo tra gli individui *vicini*;

- indipendenza di ogni singolo individuo.

Un classico algoritmo a grana fine è quello descritto in [6], il cui pseudocodice è riportato in Figura 4.

```
Program Algoritmo_Genetico;
begin
t=0;
β (t) = INIZIALIZZA_POPOLAZIONE() ;
repeat

        for i = 1 to m do f(A_i) = COMPUTA_FITNESS(A_i);
        fitness_medio = CALCOLA_FITNESS_MEDIO(β (t));
        for k = 1 to m do
        begin

              A_k = SELEZIONA(β (t));
              /* Ogni individuo viene selezionato confrontando il suo valore di fitness con il
              fitness_medio */
              if (P_C > random(0,1)) then
              begin
                   A_i = SELEZIONA(β (t));
                   A_figlio = CROSSOVER( A_i,A_k);
                   if (P_M > random (0,1)) then MUTAZIONE (A_figlio);
                   if (P_I > random(0,1)) then INVERSIONE (A_figlio);
                   β (t + 1)=AGGIORNA_POPOLAZIONE (A_figlio);

              end

        end;


t=t+1;
until (condizione_di_terminazione);
end
```

Figura 1: Algoritmo genetico di Holland.

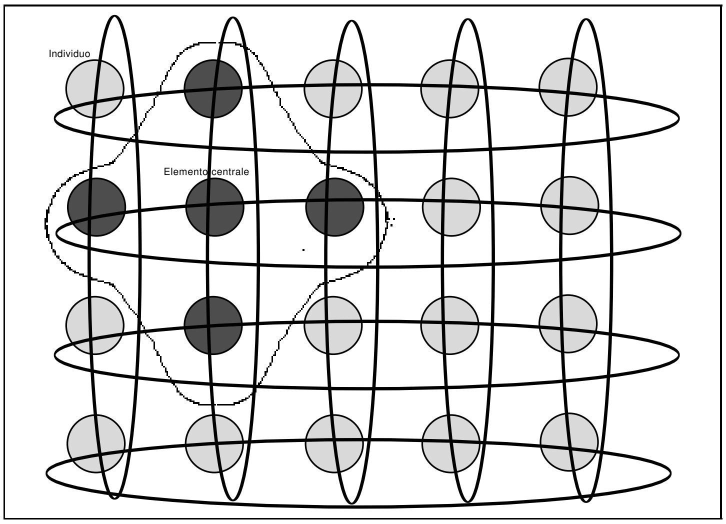

Figura 2: Struttura logica toroidale di connessione tra individui.

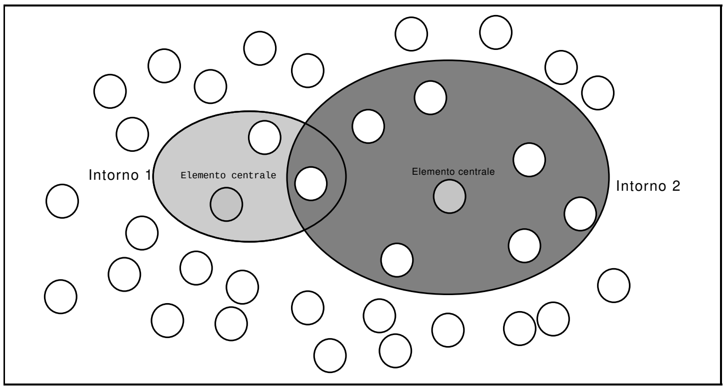

Figura 3: Esempio di sovrapposizione di intorni.

```
Program Grana_Fine;
begin
popolazione_iniziale=GENERA(numero_individui);
ASSEGNA_POSIZIONE_NELLA_GRIGLIA(popolazione_iniziale);
while condizione_di_terminazione do

    for cella = 1 to numero_celle do
    begin

        contenuto_cella=SELEZIONA(popolazione);
        /* Seleziona l'individuo che deve occupare la posizione cella della griglia, tra quelli
        assegnati a cella e al suo intorno */
        individuo_vicino=SELCASUALE(popolazione);
        CROSSOVER(individuo_vicino,contenuto_cella);
        /* Seleziona casualmente un individuo tra i vicini di cella ed effettua un crossover con
        la soluzione associata a cella secondo la probabilità di riproduzione. Assegna uno dei
        figli a contenuto_cella */
        MUTAZIONE(contenuto_cella);
        /* Muta l'individuo assegnato a contenuto_cella secondo la probabilità di mutazione */
        COMPUTA_FITNESS(contenuto_cella);
        /* Calcola il valore della fitness del nuovo individuo assegnato a cella */

    end;

end
```

Figura 4: Algoritmo a grana fine espresso in pseudocodice.

7.2 Algoritmi genetici paralleli a grana grossa

Nel modello a grana grossa la popolazione è divisa in un certo numero di sottoinsiemi chiamati **isole**, che determinano una partizione della popolazione complessiva in sottopopolazioni.

Il processo evolutivo avviene esclusivamente all'interno di tali isole. In questo modo, rispetto ad una popolazione panmitica senza struttura, si ha la possibilità di mantenere una maggiore diversità genetica all'interno della popolazione complessiva. Ciò si riflette in una migliore esplorazione dello spazio delle soluzioni: isole diverse esplorano zone distanti in parallelo. Per evitare la convergenza delle popolazioni delle isole verso ottimi locali, si effettua periodicamente una migrazione di alcuni individui di un'isola verso un'altra. In tal modo si rimescolano i geni delle popolazioni, e si evita una convergenza su un ottimo locale.

I due principali modelli di implementazione per l'algoritmo a grana grossa sono:

- il modello **a isole**;
- il modello **stepping stone**.

Mentre nel modello ad *isole* la migrazione avviene verso qualsiasi isola, nel modello *stepping stone* si definisce una vicinanza tra isole, cosicchè la migrazione possa avvenire esclusivamente tra isole vicine.

I risultati degli studi fin qui condotti mostrano come il numero di individui fatti migrare e l'intervallo di migrazione (intervallo di tempo tra due migrazioni successive) siano entrambi fattori critici: un alto numero di individui che migrano riducono il comportamento del modello a isole a quello di una popolazione panmitica, e quindi si perdono i vantaggi rispetto a quest'ultima. Un basso numero di migranti tendono a non rimescolare il patrimonio genetico, e quindi non permettono di oltrepassare gli ottimi locali trovati nelle singole isole.

8 Studio degli AG per il TSP

In questo Paragrafo presentiamo i risultati dello studio effettuato sull'applicazione degli AG al problema del commesso viaggiatore [2], che in seguito indicheremo con l'abbrevazione TSP.

Gli esperimenti sono stati eseguiti sull'elaboratore parallelo nCUBE2.

La prima fase dello studio ha riguardato l'implementazione e la prova di AG sequenziali, allo scopo di effettuare una preliminare valutazione sui diversi criteri di rimpiazzamento degli individui all'interno della popolazione, sui diversi valori del parametro di *crossover* e sul tipo di operatore di *crossover*. Sono stati implementati due operatori di *crossover* e tre tipi di rimpiazzamento dei genitori con i figli, ed è stata studiata l'influenza di questi operatori genetici sulla bontà della soluzione.

I test effettuati hanno mostrato il comportamento degli algoritmi, evidenziando quale scelta degli operatori genetici di rimpiazzamento e di *crossover* conduce ad una migliore convergenza dell'algoritmo. L'operatore di *crossover* che ha mostrato un migliore comportamento è stato utilizzato negli AG paralleli. Sono stati implementati, secondo il modello di programmazione SPMD (*Single Program Multiple Data*) [1], AG paralleli a *grana fine* e a *grana grossa*, e ne sono state misurate le prestazioni al variare di alcuni parametri. Il modello SPMD prevede che tutti i nodi eseguano lo stesso programma sequenziale su differenti dati del problema.

L'algoritmo genetico a *grana fine* adotta uno schema di *mapping* che consente di far variare in modo indipendente la dimensione della popolazione e il numero di nodi sui quali l'algoritmo viene eseguito.

8.1 Il problema del TSP

Il problema del commesso viaggiatore è definito come segue:
dato un insieme di n città e un modo di ottenere la distanza tra di esse, si trovi un cammino di lunghezza minima che passi per tutte le città ritornando alla città di partenza, passando per ogni città una sola volta.

Più formalmente il TSP si può definire nel seguente modo:
siano $C = (c_1, c_2, ..., c_n)$ le città e, date due città qualsiasi c_i c_j, sia $d(c_i, c_j)$ la distanza che le separa.

Il problema è trovare una permutazione $\pi\prime$ delle città

$$(c_{\pi'(1)}, c_{\pi'(2)}, ..., c_{\pi'(n)})$$

tale che

$$\sum_{i=1}^{n} d(c_{\pi'(i)}, c_{\pi'(i+1)}) \leq \sum_{i=1}^{n} d(c_{\pi^k(i)}, c_{\pi^k(i+1)}) \qquad \forall \pi^k \neq \pi' \tag{1}$$

Dove $(n + 1) \equiv 1$.

8.2 AG sequenziali

Il modello di generazione da noi adottato nell'implementazione degli AG sequenziali è il *modello di generazione discreto* [8] (vedi Paragrafo 4) in cui la popolazione di genitori viene tenuta distinta dalla popolazione dei figli. Quando i figli sono stati tutti generati, entrano a far parte della popolazione di genitori seguendo il criterio di rimpiazzamento adottato.

Gli AG sono stati implementati usando la rappresentazione del TSP detta a *percorso*: i percorsi sono rappresentati da una sequenza ordinata di città, e le città contigue sono quelle unite da un arco. Gli operatori genetici operano su un insieme ordinato di numeri, il cui campo di variabilità va da 0 al numero delle città del particolare problema rappresentato. La rappresentazione a *percorso*, pur avendo il vantaggio della chiarezza, richiede l'uso di operatori genetici modificati. Infatti, applicando il normale operatore di *crossover*, si corre il rischio di creare percorsi non legali (città ripetute e percorsi interrotti).

Si sono implementati due diversi operatori di *crossover* e tre modalità di rimpiazzamento dei padri con i figli. Sia negli AG sequenziali che in quelli paralleli, descritti nel Paragrafo 9, il criterio adottato per la selezione degli individui sui quali applicare l'operatore di *crossover* per produrre figli è casuale. Nei Paragrafi seguenti si descrivono gli operatori genetici e le modalità di rimpiazzamento adottate.

8.2.1 Operatore di mutazione

L'operatore di mutazione implementato per la rappresentazione a *percorso* è un operatore già usato in [8]. Esso non fa altro che scegliere due distinte città della stringa che definisce il *percorso* corrente, e scambiarle di posto. Per esempio, avendo un problema TSP a 6 città, definito dalla soluzione corrente

$$A = (123456)$$

ed applicando l'operatore di mutazione alle città 3 e 4, si ottiene

$$A' = (124356)$$

che è ancora una soluzione ammissibile del problema.

8.2.2 Operatore di crossover

L'operatore di *crossover* per la rappresentazione a *percorso* deve essere tale da non generare percorsi illegali. Infatti, applicando semplicemente la definizione originaria di Holland, si ottengono percorsi con città isolate, quindi non legali. Per esempio, sempre trattando un TSP a 6 città, supponiamo di avere i due genitori:

$$A = (432156)$$

e

$$B = (123465)$$

e scegliamo il punto di *crossover* a metà, cioè dopo la terza città. I figli generati sono quindi:

$$C = (432|465)$$

e

$$D = (123|156).$$

Risulta chiaro come entrambi i figli C e D rappresentino percorsi illegali. In D la città 4 non ha archi entranti o uscenti, e la città 1 ha due archi di troppo. In C si nota come la città 1 risulti isolata dalle altre, e come la città 4 abbia un arco di troppo sia in entrata che in uscita.

Nel seguito vengono esposte le soluzioni adottate per consentire la creazione di percorsi legali mediante l'applicazione dell'operatore di *crossover*.

8.2.3 Operatore di crossover ad un punto

L'operatore di *crossover ad un punto* [3], opera spezzando il codice genetico dei genitori in uno specifico punto, e ricombinando nei figli le informazioni contenute nelle due metà. Il crossover implementato evita la creazione di percorsi illegali controllando, per ogni città che deve essere inserita nel *percorso* del figlio, se questa non sia già stata inserita precedentemente dall'altro genitore. Se risulta già inserita, l'operatore di *crossover* passa alla prossima città del padre. La legalità dei percorsi dei genitori, rende sempre possibile trovare nel padre il numero di città necessarie a completare correttamente il *percorso* del figlio. Rifacendoci all'esempio precedente, il figlio C sarebbe stato creato dall'operatore di *crossover ad un punto* nel seguente modo. Dapprima il figlio C avrebbe ereditato il codice genetico del padre A fino al punto di *crossover*, cioè fino alla città 2 compresa

$$C = (432 * **);$$

quindi l'operatore avrebbe considerato l'altro genitore B dal punto di crossover in poi (cioè le città dalla 4 in poi). Ma, come si vede esaminando la stringa che definisce C, la città 4 risulta già presente nel figlio. Quindi l'operatore di *crossover* avrebbe continuato con la successiva città del padre B, che è 6. Questa non è presente nel figlio C, e può quindi in esso essere inserita ottenendo:

$$C = (4326 * *).$$

Successivamente C viene completato con le due città 5 e 1 del padre B.

$$C = (432651).$$

Il *percorso* ottenuto è valido.

8.2.4 Operatore di crossover a due punti

L'operatore di *crossover a due punti* [7] opera spezzando il codice genetico in tre parti, e ricombinandole nei figli in modo da formare percorsi legali. L'operatore funziona in modo simile al *crossover ad un punto*, controllando per ogni città che deve essere inserita nella stringa del figlio, se questa non sia già stata inserita precedentemente.

8.2.5 Modalità di rimpiazzamento

I criteri di rimpiazzamento stabiliscono le modalità con le quali i figli che vengono generati mediante l'applicazione dell'operatore di *crossover* entrano nella popolazione di soluzioni. Le soluzioni implementate sono tre, e verranno chiamate nel seguito R1, R2 e R3.

Le tre modalità funzionano nel seguente modo:

- modalità di rimpiazzamento R1: sostituisce gli individui della popolazione che hanno alti valori di *fitness* [4]. Dal momento che la popolazione di individui viene mantenuta ordinata in base al valore di *fitness*, R1 non fa altro che sostituire nella popolazione tutti gli individui situati nelle posizioni contraddistinte da un'alta *fitness* con i figli appena creati. In questo modo si possono introdurre nella popolazione anche dei figli che hanno una *fitness* più alta dei padri che stanno rimpiazzando. D'altronde proprio questo fatto introduce una maggiore variabilità nel codice genetico complessivo della popolazione, e ciò può risultare utile per esplorare un maggior numero di *regioni* dello spazio delle soluzioni (vedi Paragrafo 5);
- modalità di rimpiazzamento R2: ordina dapprima i figli in base al valore di *fitness*, per poi confrontare il figlio con minor valore di *fitness* con l'elemento della popolazione che ha il più alto valore di *fitness*. Se il figlio risulta avere una *fitness* migliore (più bassa), allora rimpiazza l'elemento della popolazione, e va avanti considerando il prossimo figlio e il successivo elemento della popolazione. In tal modo si rimpiazzano gli elementi peggiori della popolazione con i figli appena creati solamente se i figli hanno migliori valori di *fitness*. Si controlla così maggiormente l'andamento della popolazione di soluzioni lungo il susseguirsi delle generazioni, permettendo solo ai migliori di entrarne a far parte;
- modalità di rimpiazzamento R3: utilizza la *fitness media* della popolazione per decidere se un figlio deve rimpiazzare o meno un elemento della popolazione. Tra tutti i figli generati, vengono selezionati solo quelli che hanno un valore di *fitness* inferiore alla media (portano cioè soluzioni migliori della media della popolazione). Questi figli vanno a rimpiazzare solamente gli elementi della popolazione che hanno un valore di *fitness* superiore alla media.

9 AG paralleli

9.1 Modello a grana grossa

Il modello di generazione adottato nell'implementazione degli AG paralleli a *grana grossa* è il *modello di generazione discreto* (vedi Paragrafo 4), in cui i figli generati vengono inseriti nella popolazione successiva. Come risulta da [8], tale modello di generazione risulta essere il più adatto per gli AG a *grana grossa*. L'implementazione adotta il modello *stepping stone* [8] (vedi Paragrafo 7.2), con le sottopopolazioni connesse tramite una struttura ad anello. Le migrazioni avvengono in un solo senso e ad intervalli uguali per tutte le sottopopolazioni. Gli individui da trasferire alle altre sottopopolazioni sono scelti tra quelli che forniscono il miglior valore di *fitness*. Lo pseudocodice dell'algoritmo a *grana grossa* implementato è riportato in Figura 5.

La popolazione complessiva è stata fissata in N individui, da suddividere in base al numero di processori usati per l'esecuzione dell'algoritmo. Il numero di individui da migrare è una percentuale fissa della sottopopolazione. Le migrazioni, seguendo il criterio proposto in [5], avvengono periodicamente ad intervalli regolari per ciascuna sottopopolazione, dopo che si sono eseguite un certo numero di generazioni.

La modalità adottata per il rimpiazzamento degli elementi della sottopopolazione con gli individui migrati da altre sottopopolazioni è R1 allo scopo di immettere tutti i migranti all'interno della sottopopolazione. Essendo il numero di individui da migrare un parametro critico[5] negli AG a *grana grossa*, si sono rilevate le prestazioni dell'algoritmo al variare del parametro di migrazione.

[4] Ricordiamo qui che il problema trattato richiede di trovare il minimo percorso che unisce tra loro tutte le città. Un alto valore di *fitness* indica un percorso più lungo di altri, e quindi la soluzione corrispondente può essere scartata.

[5] Si intende che il parametro di migrazione può influenzare la convergenza degli AG verso l'ottimo.

```
Program GRANA_GROSSA;
begin
whoami(mionodo,M);
/*Determina l'identificatore del nodo e la dimensione dell'ipercubo */
num_processori = 2^M;
num_individui = TOTALE_INDIVIDUI div num_processori;
num_figli = parametro_crossover * num_individui;
num_mutazioni = parametro_mutazione * num_individui;
popolazione=GENERA(num_individui);
/* Inizializza la sottopopolazione di num_individui su ogni nodo*/
ORDINAMENTO(popolazione);
CALCOLA_FITNESS(popolazione);
for epoche = 1 to num_epoche do
begin

        for gen = 1 to num_gen do
        begin

                for i = 1 to (num_figli div 2) do
                begin

                        genitore1=SELEZIONA(popolazione);
                        genitore2=SELEZIONA(popolazione);
                        CROSSOVER(genitore1,genitore2,figlio1,figlio2);
                        /* Il crossover produce dai medesimi genitori due figli diversi */
                        AGGIORNA_POP_FIGLI(figlio1,figlio2,pop_figli);
                        /* i figli generati vengono messi nella popolazione dei figli*/

                end;
                for i = 1 to (num_mutazioni) do
                begin

                        figlio_da_mutare=SELEZIONA(pop_figli);
                        MUTAZIONE(figlio_da_mutare);

                end;
                CALCOLA_FITNESS(pop_figli);
                ORDINAMENTO(pop_figli);
                /*Ordina i figli in base al valore crescente di fitness*/
                RIMPIAZZAMENTO(popolazione,pop_figli);
                /*I figli rimpiazzano solo gli individui della popolazione con valore di fitness peggiore */

        end; /* fine ciclo di num_gen*/
        migranti=SELEZIONA_INDIVIDUI_MIGLIORI(popolazione);
        INVIA_AL_NODO(migranti);
        RICEVI_DAL_NODO(migranti);
        /*invia al nodo successivo e riceve dal nodo precedente*/
        RIMPIAZZA_MIGRANTI(popolazione,migranti);
        /*Gli individui ricevuti rimpiazzano gli individui peggiori della popolazione */

end; /* fine ciclo di num_epoche */
end
```

Figura 5: Pseudocodice dell'algoritmo genetico parallelo a *grana grossa*.

9.2 Modello a grana fine

Il modello di generazione adottato nell'implementazione degli AG paralleli a *grana fine* è il *modello di generazione continuo* (vedi Paragrafo 4), in cui i figli generati vengono inseriti immediatamente nella popolazione corrente. Come risulta da [8], tale modello risulta essere il più adatto per gli AG a *grana fine*.

Nel modello a *grana fine* la popolazione è strutturata secondo una topologia logica che definisce le possibilità di interazione di un individuo con il resto della popolazione: ogni individuo s è posto su un vertice $v(s)$ della topologia logica T, e può accoppiarsi solamente con gli individui posti nell'*intorno* costituito dai vertici direttamente connessi a $v(s)$ in T. Nel nostro caso la popolazione di 2^N individui ha una struttura logica ad *ipercubo* N-dimensionale. Sfruttando la definizione ricorsiva della topologia ad ipercubo si possono inoltre rendere indipendenti il numero di individui della popolazione ed il numero di processori fisici disponibili. Come si può vedere dall'esempio schematizzato in Figura 6, una popolazione di $2^3 = 8$ individui può essere allocata su un ipercubo di $2^2 = 4$ nodi semplicemente mascherando il primo (o l'ultimo) *bit* del codice **Grey** [13] usato per la numerazione dei vertici di un ipercubo. Il nodo fisico $X00$ conterrà gli individui 000 e 100 della topologia logica senza per questo violare in alcun modo le relazioni di vicinanza che determinano la strutturazione della popolazione. In altre parole, gli individui dell'intorno di un qualsiasi individuo s continueranno ad essere posti su vertici connessi a $v(s)$ anche nella topologia fisica.

Tale schema di *mapping* può essere chiaramente generalizzato: per determinare l'allocazione di una popolazione di 2^N individui su un ipercubo fisico di 2^M nodi con N qualsiasi e $M < N$ è sufficiente mascherare i primi (gli ultimi) $N - M$ *bit* della codifica binaria di ogni individuo.

Sfruttando questa proprietà della topologia ad ipercubo, è semplice disegnare un'implementazione parallela di un algoritmo genetico a *grana fine*, quale quella descritta dallo pseudocodice di Figura 7, in cui il numero di individui della popolazione ($TOTALE\_INDIVIDUI$) ed il numero di processori utilizzati ($num\_processori$) siano parametri indipendenti.

10 Risultati sperimentali

Ogni AG, sequenziale e parallelo, è stato applicato alla risoluzione di due problemi TSP di dimensioni pari a 48 città e 105 città. I problemi utilizzati per le prove sono noti come **GR48**, di soluzione ottima pari a 5046, e di **LIN105**, di soluzione ottima pari a 14379[6]. Per ogni AG sono state condotte 32 prove, utilizzando ogni volta popolazioni iniziali casuali e diverse. Si sono ottenuti così risultati sufficientemente generali da poter essere assunti come comportamento medio. Delle prove effettuate si sono presi in considerazione i seguenti risultati:

- la media delle soluzioni trovate dagli AG: $MED = \frac{\sum_{i=1}^{32} F_{E_i}}{32}$, dove con F_{E_i} si indica la *fitness* migliore dell'esecuzione E_i dell'algoritmo;
- la migliore delle soluzioni trovate: $MIG = min\{F_{E_i}, i = 1....32\}$;
- la peggiore delle soluzioni trovate: $PEG = max\{F_{E_i}, i = 1....32\}$.

10.1 Risultati degli AG sequenziali

Le prove sugli AG sequenziali sono state effettuate mantenendo invariati i seguenti valori:

- popolazione = 640 individui;
- parametro di Mutazione = 0.2, che corrisponde ad una applicazione dell'operatore di mutazione sul 20% della popolazione;

[6]Entrambi i problemi sono disponibili all'indirizzo: ftp://elib.zib-berlin.de/pub/mp-testdata/tsp/tsplib.html

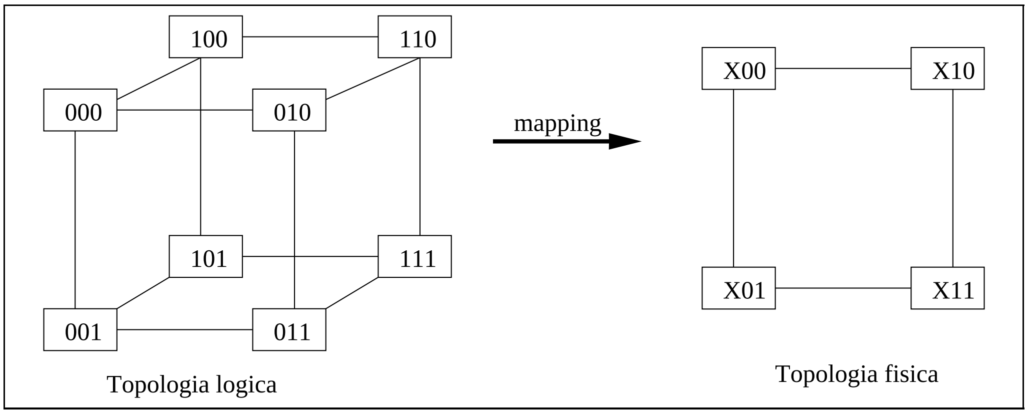

Figura 6: Esempio di applicazione dello schema di *mapping* per l'algoritmo genetico a *grana fine*.

```
Program Grana_Fine;
begin
whoami(mionodo,M);
/*Determina l'identificatore del nodo e la dimensione dell'ipercubo allocato */
TOTALE_INDIVIDUI = 2^N;
num_processori = 2^M;
num_individui = TOTALE_INDIVIDUI div num_processori;
popolazione=GENERA();
MAPPING(popolazione,num_individui,mionodo);
/*Determina su ogni nodo quali sono gli individui vicini allocati */
while condizione_di_terminazione do
begin

        INVIA_A_NODI_VICINI();
        RICEVI_DA_NODI_VICINI()();
        /* Scambio degli individui con i nodi fisicamente connessi */
        for cella = 1 to num_individui do
        begin

                contenuto_cella=SELEZIONA(popolazione);
                /* Seleziona un individuo tra quelli assegnati a mionodo */
                individuo_vicino=SELCASUALE(popolazione);
                /* Seleziona un individuo vicino tra gli individui che sono nell'intorno */
                CROSSOVER(individuo_vicino,contenuto_cella,figlio1,figlio2);
                /* Genera due figli applicando il crossover su individuo_vicino e contenuto_cella */
                figlio_da_mutare =SELFIGLIO(figlio1,figlio2);
                /*Seleziona uno dei due figli per applicare la mutazione*/
                MUTAZIONE(figlio_da_mutare);
                COMPUTA_FITNESS(figlio1, figlio2);
                /* Calcola i valori di fitness dei figli generati */
                contenuto_cella= RIMPIAZZAMENTO(contenuto_cella,figlio1,figlio2);
                /*Seleziona l'individuo che deve occupare la posizione cella: viene selezionato
                l'individuo con valore di fitness più basso*/

        end;


end;
end
```

Figura 7: Algoritmo genetico a *grana fine* espresso in pseudocodice.

- generazioni: 2000 per il TSP a 48 città e 3000 per il TSP a 105 città; ciò per consentire una migliore risoluzione del problema a 105 città.

Nella prima serie di esperimenti è stato studiato il comportamento degli AG al variare del tipo di crossover (ad un punto e a due punti) e della modalità di rimpiazzamento (R1, R2, R3), fissato il valore del parametro di *crossover*.

Nella seconda serie di esperimenti abbiamo studiato l'influenza del valore del parametro di *crossover* (che determina il numero di figli prodotti in ogni generazione) al variare del tipo di rimpiazzamento, fissato il tipo di crossover.

Nella terza serie di esperimenti è stata valutata l'influenza delle modalità di rimpiazzamento, fissato sia il tipo che il valore del parametro di crossover.

Nei grafici presentati nel seguito sono riportati i valori medi delle prove effettuate elaborando il problema TSP a 48 città all'aumentare del numero di generazioni. Le Tabelle riportano i valori medio (MED), migliore (MIG) e peggiore (PEG) delle prove effettuate sui problemi TSP a 105 e 48 città.

10.1.1 Tipo di crossover

I tipi di crossover adottati in questa serie di esperimenti sono due:

- crossover ad un punto;
- crossover a due punti.

In Figura 8 sono riportati i risultati medi delle prove effettuate. Il valore del parametro di *crossover* impiegato nelle prove è C=0.4, che corrisponde all'applicazione dell'operatore sul 40% della popolazione. Si può osservare che, in generale, il *crossover* a due punti converge ad una soluzione migliore del *crossover* ad un punto. Quest'ultimo presenta un comportamento iniziale migliore, ma converge poi a valori di *fitness* più alta. In particolare si può osservare che usando le modalità di rimpiazzamento R2 ed R3 il *crossover* a un punto fornisce soluzioni migliori del *crossover* a due punti fino a circa la generazione 500, mentre con la modalità di rimpiazzamento R1 il *crossover* ad un punto offre soluzioni migliori fino a circa la generazione 700 circa. Nelle Tabelle 1 e 2 sono riportati i risultati delle prove su 48 e su 105 città. Anche nel caso del problema a 105 città il *crossover* a due punti risulta fornire risultati migliori.

10.1.2 Parametro di crossover

In Figura 9 si possono osservare i risultati delle prove condotte sulle tre modalità di rimpiazzamento R1, R2, R3 al variare del parametro di *crossover*. La variazione del valore del parametro di *crossover* influisce in maniera simile sulle modalità di rimpiazzamento R2 e R3: in entrambi i casi all'aumentare del parametro di *crossover* si ottiene una maggior velocità di convergenza.

Nel caso della modalità di rimpiazzamento R1 si ha un comportamento diverso: la maggior velocità di convergenza si ha per il parametro di $C = 0.6$. Infatti, data la modalità con la quale si effettuano i rimpiazzamenti, con $C = 0.8$ si inseriscono nella popolazione in maggior quantità figli che hanno un valore di *fitness* peggiore dei padri. Dai dati riportati in Tabella 3, relativa al problema TSP a 48 città, si può osservare che per le modalità di rimpiazzamento R2 e R3 il miglior valor medio delle soluzioni si ottiene per $C = 0.4$, mentre per R1 si ha la soluzione media migliore per $C = 0.6$. Ciò è in accordo con quanto si osserva nei grafici di Figura 9: R1 ottiene soluzioni mediamente migliori per $C = 0.6$. In Tabella 4 sono riportati i dati relativi al problema a 105 città.

10.1.3 Modalità di rimpiazzamento

Per il confronto delle modalità di rimpiazzamento si è scelto il valore $C = 0.4$ del paramentro di *crossover*, cioè il valore che ha dimostrato fornire i migliori risultati con i rimpiazzamenti R2 e R3 e che mostra un buon comportamento anche con R1. In Figura 10 si può osservare con maggiore chiarezza il confronto tra le diverse modalità di rimpiazzamento. Le modalità R2 e R3, come già notato in precedenza, convergono con maggiore velocità rispetto a R1. Inoltre R2 risulta avere una convergenza più veloce di R3.

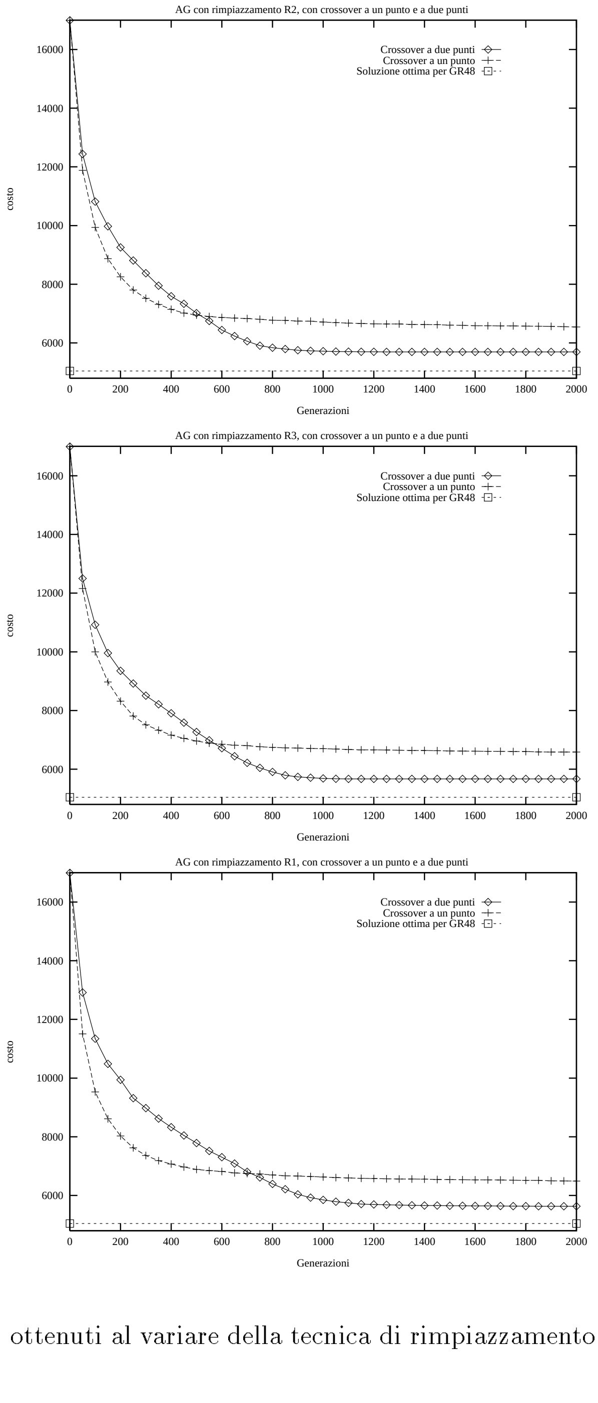

Figura 8: Valori di *fitness* ottenuti al variare della tecnica di rimpiazzamento (R1, R2, R3) e del tipo di *crossover*.

	crossover ad un punto			crossover a due punti		
	R1	*R2*	*R3*	*R1*	*R2*	*R3*
MED	6288	6547	6587	5632	5696	5669
MIG	5586	5717	5863	5315	5243	5178
PEG	8247	7623	7746	6079	6180	6140

Tabella 1: Valori di *fitness* ottenuti con l'algoritmo genetico sequenziale al variare della tecnica di ripiazzamento (R1, R2, R3) e del tipo di *crossover* per il TSP a 48 città.

	crossover ad un punto			crossover a due punti		
	R1	*R2*	*R3*	*R1*	*R2*	*R3*
MED	25812	27045	27523	22857	22982	21864
MIG	21513	24222	24703	20587	18982	18942
PEG	29964	29874	32938	26176	23904	25115

Tabella 2: Valori di *fitness* ottenuti con l'algoritmo genetico sequenziale al variare della tecnica di rimpiazzamento (R1, R2, R3) e del tipo di *crossover* per il TSP a 105 città.

		Valore di C			
		0.2	0.4	0.6	0.8
R1	**MED**	6255	5632	5585	5870
	MIG	5510	5315	5135	5305
	PEG	7828	6079	6231	6693
R2	**MED**	5902	5696	5735	5743
	MIG	5405	5243	5323	5410
	PEG	7122	6180	6225	6243
R3	**MED**	6251	5669	5722	5773
	MIG	5441	5178	5281	5200
	PEG	7354	6140	6370	6594

Tabella 3: Valori di *fitness* ottenuti con l'algoritmo genetico sequenziale variando il parametro di *crossover* per il TSP a 48 città.

		Valore di C			
		0.2	0.4	0.6	0.8
R1	**MED**	34992	22857	22010	26955
	MIG	29997	20587	19450	23450
	PEG	39710	26176	25318	30865
R2	**MED**	32486	22982	20661	21154
	MIG	23423	18982	19176	18405
	PEG	39273	23904	22653	25815
R3	**MED**	28246	21864	22444	22735
	MIG	34399	18942	19600	18825
	PEG	42050	25115	25522	25778

Tabella 4: Valori di *fitness* ottenuti con l'algoritmo genetico sequenziale variando il parametro di *crossover* per il TSP a 105 città.

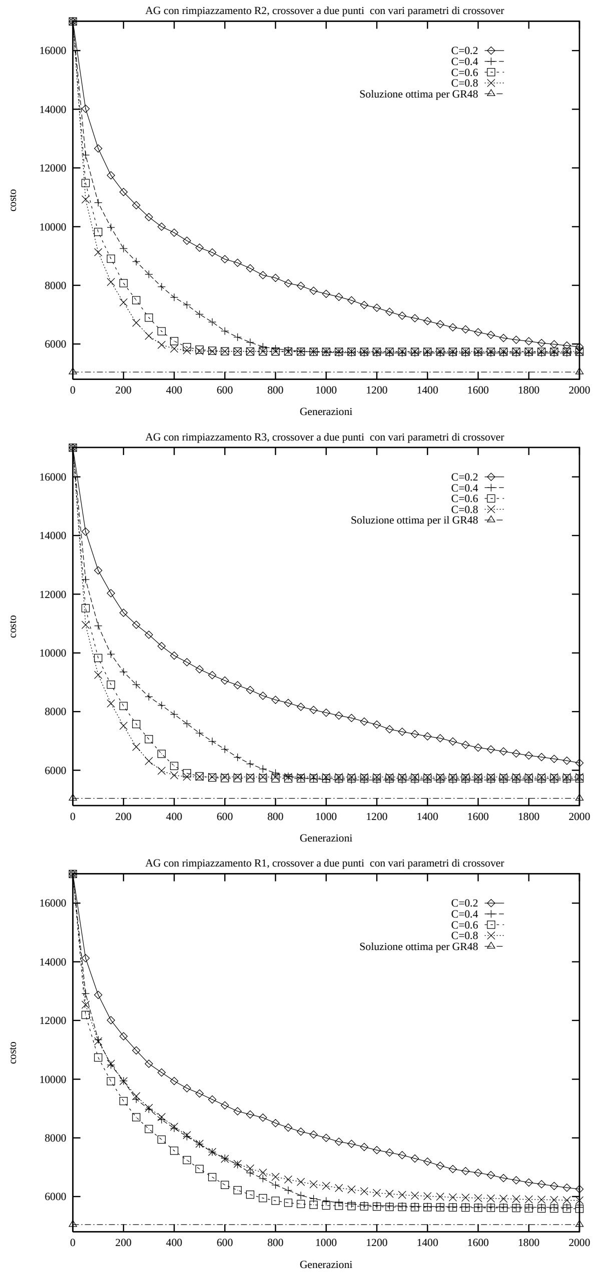

Figura 9: Valori di *fitness* ottenuti al variare della tecnica di rimpiazzamento adottata e del valore del parametro di *crossover*.

Esaminando però i valori di *fitness* raggiunti dai tre rimpiazzamenti dopo 2000 generazioni (Tabella 3) si nota che R1 arriva ad un valore di *fitness* migliore sia di R2 che di R3, e che R3 consegue migliori valori di *fitness* di R2. Pertanto le modalità di rimpiazzamento che esibiscono una convergenza più lenta mostrano di avvicinarsi maggiormente all'ottimo.

10.2 Risultati degli AG paralleli

10.2.1 Modello a grana grossa

Dai *test* effettuati nei precedenti Paragrafi si deduce che il *crossover* a due punti porta ad una migliore approssimazione della soluzione ottima; inoltre la modalità di rimpiazzamento R2 ha mostrato una maggiore velocità di convergenza verso buoni valori di *fitness*. L'algoritmo parallelo adotta quindi il *crossover* a due punti e la modalità di rimpiazzamento R2. Nelle Tabelle 5 e 6 sono riportati i risultati dell'algoritmo genetico parallelo a *grana grossa* per i problemi a 48 e a 105 città. Osservando anche la Figura 11, relativa al problema TSP a 48 città, si può notare che in generale si ha un peggioramento delle soluzioni media (MED), migliore (MIG) e peggiore (PEG) all'aumentare del numero di nodi. Ciò può essere spiegato osservando che la popolazione complessiva di 640 individui viene suddivisa in base al numero di nodi: su 4 nodi si hanno sottopopolazioni di 160 individui, mentre su 64 nodi la sottopopolazione è di soli 10 individui. Quindi al diminuire della sottopopolazione, si rileva un peggioramento della ricerca: sottopopolazioni con pochi individui non esplorano sufficientemente bene lo spazio delle soluzioni.

L'influenza del numero di individui da migrare è chiaramente osservabile notando come la soluzione media cambi al variare del parametro di migrazione M. Sul problema TSP a 48 città si nota che per un numero di nodi pari a N=4, 8, 16 il valore di migrazione che fornisce migliori soluzioni medie è M=0.1. Per un maggior numero di nodi invece il valore di M che fornisce migliori soluzioni medie è diverso: infatti per N=32 si ha M=0.3, e per N=64 si ha M=0.5. Quindi, al diminuire del numero di individui allocati su ogni nodo, un alto valore del parametro di migrazione riesce a migliorare le soluzioni trovate, rimescolando maggiormente il codice genetico delle sottopopolazioni.

Osservando i risultati riportati in Tabella 6, relativi al problema TSP a 105 città, si nota che la tendenza viene confermata: per N=4, 8 e 16 risulta M=0.3, mentre per N=32 e 64 risulta M=0.5.

In Tabella 7 sono riportati i tempi medi di esecuzione e gli indici di prestazione ed efficienza ottenuti elaborando l'algoritmo genetico parallelo all'aumentare il numero dei nodi usati. I tempi medi di esecuzione sono espressi in secondi. Dai dati riportati in Tabella 7 e dalla Figura 12 si può osservare che si ottiene uno *speedup* superlineare. La causa di questo comportamento è da attribuirsi all'algoritmo di *quicksort* utilizzato per l'ordinamento della popolazione.

10.2.2 Modello a grana fine

Come nel modello *a grana grossa*, anche nel modello *a grana fine* il *crossover* adottato è quello a due punti.

L'algoritmo è stato eseguito per 2000 generazioni sul problema TSP a 48 città, e 3000 generazioni sul problema TSP a 105 città. I valori di *fitness* raggiunti dall'algoritmo che adotta il modello *a grana fine* sul problema a 48 città sono riportati in Tabella 8.

Sul problema a 105 città è stato eseguito l'algoritmo genetico a *grana fine* variando la dimensione dell'ipercubo logico.

In Tabella 9 sono riportati i risultati di *fitness* conseguiti eseguendo l'algoritmo a *grana fine* variando le dimensioni della popolazione: si va dai 128 individui corrispondenti ad un ipercubo di dimensione 7, ai 1024 individui corrispondenti ad un ipercubo di dimensione 10. Come era logico aspettarsi, lo schema di *mapping* adottato permette di conseguire *fitness* che migliorano all'aumentare del numero di individui.

Inoltre, come si può leggere nelle Tabelle 10, 11, 12 e 13 relative ai parametri di prestazione dell'algoritmo a *grana fine* eseguito su una popolazione rispettivamente di 128, 256, 512 e 1024 individui, lo *speedup* migliora all'aumentare della popolazione. Fissata la dimensione logica della popolazione lo *speedup* ha un andamento che si avvicina alla linearità, come mostra la Figura 14 per la popolazione di

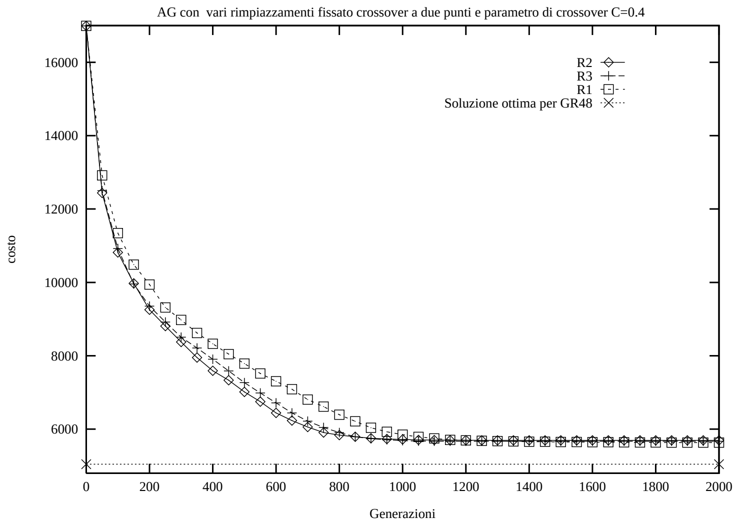

Figura 10: Valori di *fitness* ottenuti con l'algoritmo genetico sequenziale al variare della tecnica di rimpiazzamento (R1, R2, R3) con *crossover* a due punti, e parametro di *crossover* C=0.4.

		Numero di nodi					
		N=4	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
M=0.1	**MED**	5780	5786	5933	6080	6383	6995
	MIG	5438	5315	5521	5633	5880	6625
	PEG	6250	6387	6516	6648	8177	8175
M=0.3	**MED**	5807	5877	5969	6039	6383	6623
	MIG	5194	5258	5467	5470	5727	6198
	PEG	6288	6644	7030	6540	8250	7915
M=0.5	**MED**	5900	5866	5870	6067	6329	6617
	MIG	5419	5475	5483	5372	6017	6108
	PEG	6335	6550	7029	6540	8250	7615

Tabella 5: Valori di *fitness* ottenuti con l'algoritmo genetico a *grana grossa* al variare del parametro di migrazione per il TSP a 48 città.

		Numero di nodi					
		N=4	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
M=0.1	**MED**	23790	24538	27281	29364	32632	37872
	MIG	21370	21829	24532	26517	28966	36021
	PEG	26706	28381	37436	37008	37779	44048
M=0.3	**MED**	23560	24526	27063	29422	32542	35342
	MIG	20219	21120	23510	25783	30927	33015
	PEG	25899	29348	36795	39330	39131	41508
M=0.5	**MED**	23790	24670	26752	29137	32208	35067
	MIG	21272	21411	24088	23373	30617	31603
	PEG	26706	30875	36080	37779	36649	43494

Tabella 6: Valori di *fitness* ottenuti con l'algoritmo genetico a *grana grossa* al variare del parametro di migrazione per il TSP a 105 città.

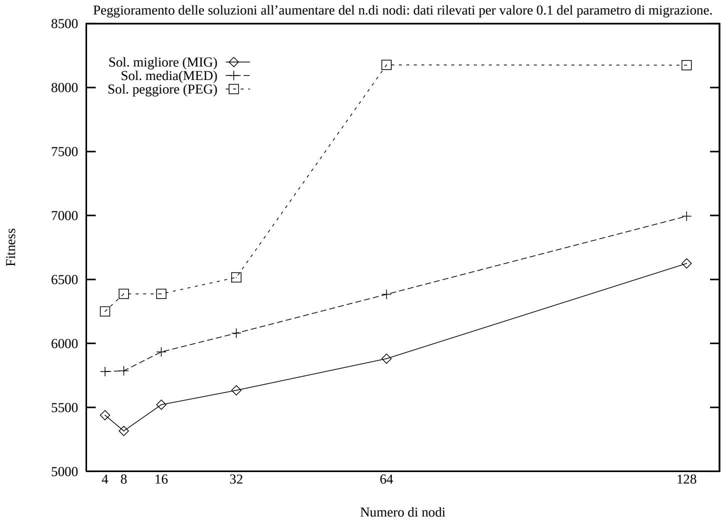

Figura 11: Valori di MED, MIG e PEG all'aumentare del numero di nodi per l'algoritmo genetico a *grana grossa* applicato al TSP a 48 città.

Numero di nodi	Tempo (in sec.)	Speedup	Efficienza
1	2600	1	1
4	417	6.23	1.565
8	181	14.36	1.795
16	85	50.58	1.911
32	40	65	2.03
64	20	130	2.03
128	10	260	2.03

Tabella 7: Valori di prestazione dell'algoritmo genetico a *grana grossa* per il problema TSP a 48 città.

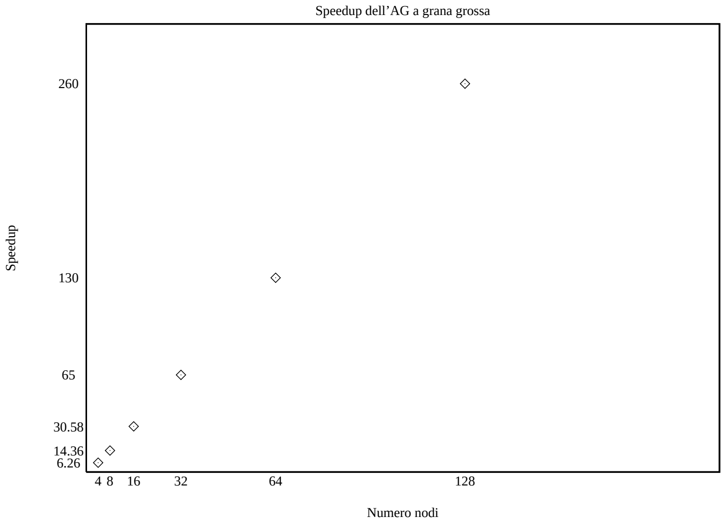

Figura 12: Valori di *speedup* dell'algoritmo genetico a *grana grossa* per il TSP a 48 città.

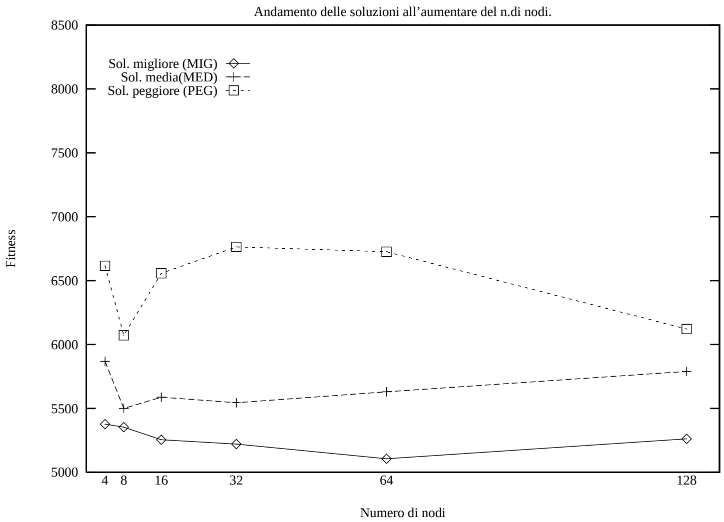

Figura 13: Valori di MED, MIG e PEG all'aumentare del numero di nodi per l'algoritmo genetico a *grana fine* eseguito su 128 individui.

	Numero di nodi					
	N=4	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
MED	5868	5499	5588	5544	5630	5789
MIG	5376	5353	5255	5221	5105	5261
PEG	7718	6072	7667	6763	6726	6424

Tabella 8: Valori di MED, MIG e PEG dopo 2000 generazioni per l'algoritmo genetico a *grana fine* applicato al TSP a 48 città.

		Numero di nodi						
		N=1	*N=4*	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
128 individui	**MED**	39894	24361	23271	23570	23963	22519	22567
	MIG	34207	20774	19830	20532	21230	20269	20593
	PEG	42127	30312	26677	27610	27634	28256	25931
256 individui	**MED**	33375	25313	22146	21616	21695	22247	21187
	MIG	29002	24059	20710	19833	20144	19660	19759
	PEG	40989	26998	23980	24007	23973	24337	22256
512 individui	**MED**	28422	23193	22032	21553	20677	20111	20364
	MIG	28987	22126	19336	20333	19093	18985	18917
	PEG	41020	25684	23450	22807	22213	21696	21647
1024 individui	**MED**	25932	23659	22256	20366	19370	18948	19152
	MIG	27010	21581	21480	18830	18256	18252	17446
	PEG	40901	25307	22757	21714	20766	19525	20661

Tabella 9: Valori di *fitness* ottenuti con l'algoritmo genetico a *grana fine* dopo 3000 generazioni al variare del numero di individui assegnati ad ogni nodo per il TSP a 105 città.

128 individui corrispondente alla dimensione 7 dell'ipercubo. L'algoritmo a *grana fine*, non includendo un algoritmo di ordinamento come l'algoritmo genetico a *grana grossa*, non mostra un andamento superlineare.

Il miglioramento dei valori di *fitness* all'aumentare del numero di nodi è dovuto alle caratteristiche dell'implementazione realizzata: questa tende a minimizzare i tempi di comunicazione a scapito della diversità delle soluzioni allocate sullo stesso nodo. In particolare, il criterio di selezione dei *partner* per l'accoppiamento, tende a scegliere soluzioni all'interno del nodo, allo scopo di minimizzare i tempi di comunicazione tra nodo e nodo. La conseguenza è una maggiore uniformità delle popolazioni allocate nei nodi, che si riflette in valori più alti di *fitness*.

Come mostrato in Tabella 14 in cui sono riportati i tempi medi di esecuzione, l'algoritmo genetico a *grana fine* risulta essere scalabile aumentando la dimensione della popolazione e la dimensione dell'ipercubo usato.

Una ulteriore caratteristica degna dell'algoritmo a *grana fine* è che la qualità delle soluzioni non viene degradata al crescere del numero dei processori impiegati, come invece avviene nell'algoritmo genetico *a grana grossa*. Il fatto è chiaramente osservabile confrontando le Figure 11, relativa all'algoritmo genetico *a grana grossa*, e 13, relativa all'algoritmo genetico *a grana fine* eseguito su 128 individui sul problema a 48 città; la stessa caratteristica si conserva anche per il problema a 105 città.

Nelle Tabelle 15 e 16 sono riportati, rispettivamente, i valori di *fitness* e i tempi di esecuzione della *grana fine*, in funzione del numero di individui allocati su ogni nodo: come si vede i risultati dipendono dal numero totale di individui allocati, in quanto il processo di *mapping* rende indipendente la struttura della popolazione dal numero di nodi sui quali questa viene allocata.

Ci si potrebbe aspettare che una popolazione di dimensione fissata, eseguita su uno o più nodi, ottenga valori identici di *fitness*, dato che la struttura della popolazione viene mantenuta dalla tecnica di *mapping* indipendente dal numero di nodi sui quali è eseguito l'algoritmo. Per esempio, si potrebbe pensare che la popolazione di 64 individui (ipercubo logico di dimensione 6, ogni individuo logicamente connesso con 6 vicini), raggiunga identici valori di fitness, sia che venga allocata su 16 nodi a gruppi di 4 individui, sia che venga allocata su 32 nodi a gruppi di 2 individui. Osservando la Tabella 15 si rileva come ciò non sia vero; in effetti gli algoritmi genetici implementati differenziano la popolazione iniziale in base al numero di nodi sui quali una popolazione viene allocata; una popolazione iniziale di 64 individui allocata su 16 nodi è diversa da una popolazione iniziale della stessa dimensione allocata su 8 nodi.

11 Conclusioni

In questo lavoro sono stati presentati i risultati dello studio effettuato sull'applicazione degli AG al problema del commesso viaggiatore. Sono stati dapprima implementati e provati AG sequenziali, allo scopo di effettuare uno studio preliminare sui diversi criteri di rimpiazzamento degli individui all'interno delle popolazioni, su diversi valori del parametro di crossover e sul tipo di *crossover*. Dai risultati ottenuti si è visto che il *crossover* a due punti trova soluzioni con *fitness* migliori rispetto al *crossover* ad un punto. La scelta del tipo di crossover è quindi un parametro importante negli AG.

Sono stati quindi implementati, su un'architettura parallela con topologia di interconnessione ad ipercubo, AG paralleli a *grana fine* e a *grana grossa*, e sono stati valutati analizzandone le prestazioni al variare di alcuni parametri. Per l'algoritmo genetico a *grana grossa* si sono ottenute soluzioni di fitness peggiori all'aumentare del numero di nodi, e uno *speedup* superlineare dovuto alla procedura di ordinamento utilizzata.

Per l'algoritmo genetico a *grana fine* si è adottata una tecnica di *mapping* della popolazione sui processori che ha permesso di rendere indipendente la dimensione della popolazione dal numero di nodi nell'esecuzione dell'algoritmo stesso. Tale tecnica di *mapping* ha permesso di conseguire *fitness* migliori all'aumentare della dimensione della popolazione con un incremento del valore di *speedup* al crescere del numero di individui dela popolazione.

Numero di nodi	Tempo (in sec.)	Speedup	Efficienza
1	6592	1	1
4	1739	3.79	0.94
8	908	7.25	0.9
16	481	13.7	0.85
32	261	25.25	0.78
64	147	44.84	0.7
128	84	78.47	0.61

Tabella 10: Valori di prestazione dell'algoritmo genetico a *grana fine* eseguito su una popolazione di 128 individui per il problema TSP a 105 città.

Numero di nodi	Tempo (in sec.)	Speedup	Efficienza
1	13187	1	1
4	3433	3.84	0.96
8	1764	7.47	0.93
16	921	14.31	0.89
32	488	27.02	0.84
64	266	49.57	0.77
128	149	88.5	0.69

Tabella 11: Valori di prestazione dell'algoritmo genetico a *grana fine* eseguito su una popolazione di 256 individui per il problema TSP a 105 città.

Numero di nodi	Tempo (in sec.)	Speedup	Efficienza
1	26380	1	1
4	6780	3.89	0.97
8	3465	7.61	0.95
16	1784	14.78	0.92
32	932	28.03	0.88
64	495	53.29	0.83
128	269	98.06	0.76

Tabella 12: Valori di prestazione dell'algoritmo genetico a *grana fine* eseguito su una popolazione di 512 individui per il problema TSP a 105 città.

Numero di nodi	Tempo (in sec.)	Speedup	Efficienza
1	52770	1	1
4	13441	3.92	0.98
8	6815	7.74	0.96
16	3489	15.12	0.94
32	1796	29.38	0.91
64	940	56.13	0.87
128	501	105.32	0.82

Tabella 13: Valori di prestazione dell'algoritmo genetico a *grana fine* eseguito su una popolazione di 1024 individui per il problema TSP a 105 città.

	Numero di nodi						
	N=1	*N=4*	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
128 individui	6592	1739	908	481	261	147	84
256 individui	13187	3433	1764	921	488	266	149
512 individui	26380	6780	3465	1784	932	495	269
1024 individui	52770	13441	6815	3489	1796	940	501

Tabella 14: Tempo di esecuzione dell'algoritmo genetico a *grana fine* dopo 3000 generazioni espressi in secondi, al variare del numero di individui assegnati ad ogni nodo per il TSP a 105 città.

Individui	Numero di nodi			
per nodo	*N=16*	*N=32*	*N=64*	*N=128*
1	27492	25364	24311	22567
2	26327	24521	22519	21187
4	26611	23963	22247	20364
8	23570	21695	20111	19152

Tabella 15: Valori di *fitness* medi ottenuti con l'algoritmo genetico a *grana fine* dopo 3000 generazioni al variare del numero di individui assegnati ad ogni nodo per il problema a 105 città.

Individui	Numero di nodi			
per nodo	*N=16*	*N=32*	*N=64*	*N=128*
1	78	80	83	85
2	139	142	147	149
4	255	261	266	269
8	481	488	495	501

Tabella 16: Tempo di esecuzione dell'algoritmo genetico a *grana fine* dopo 3000 generazioni espressi in secondi, al variare del numero di individui assegnati ad ogni nodo per il problema a 105 città.

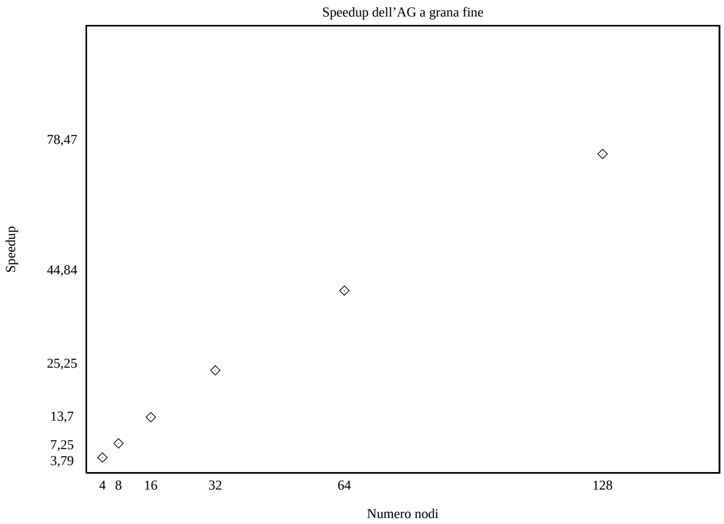

Figura 14: *Speedup* per l'algoritmo genetico a *grana fine* eseguito su 128 individui.

12 Ringraziamenti

Desideriamo ringraziare Giancarlo Bartoli, responsabile del laboratorio Calcolo Parallelo del CNUCE, per la disponibilità e la competenza dimostrate, senza le quali tale lavoro non avrebbe sicuramente raggiunto i medesimi risultati.

Riferimenti bibliografici

[1] R. Baraglia and D. Laforenza. Aspetti pratici ed esperienze d'uso di una architettura in ambiente scientifico general purponse. *Atti del Congresso Annuale AICA*, 4:699–724, 1994.

[2] A.A. Bertossi. *Strutture, algoritmi, complessità*. ECIG, 1990.

[3] H. Braun. On solving travelling salesman problems by genetic algorithms. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN , volume 496 of Lecture Notes in Computer Science*, pages 129–133. Springer-Verlag, 1991.

[4] E. Cantu-Paz. A summary of research on parallel genetic algoritms. Technical Report 95007, University of Illinois at Urbana-Champaign, Genetic Algoritms Lab. (IlliGAL), http://gal4.ge.uiuc.edu/illigal.home.html, July 1995.

[5] S. Cohoon, J. Hedge, S. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. *IEEE Transaction on CAD*, 10(4):483–491, April 1991.

[6] M. Dorigo and V. Maniezzo. Parallel genetic algorithms: Introduction and overview of current research. In *Parallel Genetic Algorithms*, pages 5–42. IOS Press, 1993.

[7] D. Goldberg and R. Lingle. Alleles, loci, and the tsp. In *Proc. of the First International Conference on Genetic Algorithms*, pages 154–159, 1985.

[8] M. Gorges-Schleuter. *Genetic Algoritms and Population Structure.* PhD thesis, University of Dortmund, 1991.

[9] J. Holland. *Adaptation in Natural and Artificial Systems.* Univ. of Mitchigan Press, 1975.

[10] F. Luccio. *La struttura degli algoritmi.* Bollati Boringhieri, 1982.

[11] Bucci M. and Circelli D. Algoritmi genetici e simulated annealing per la soluzione parallela di problemi di ottimizzazione combinatoriale. Master's thesis, Universitá degli studi di Pisa, Luglio 1996.

[12] Z. Michalewicz. *Genetic Algorithms + data structures = Evolution Programs.* Springer-Verlag, 1994.

[13] nCUBE Corporation. ncube2 processor manual. 1990.