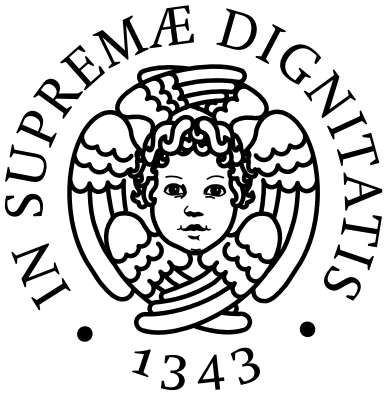
UNIVERSITÀ DEGLI STUDI DI PISA



Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Magistrale in Informatica

Tesi di Laurea

Estensione del pool evolution pattern di FastFlow per il supporto di algoritmi genetici ad isole

Relatore
prof. Marco Danelutto

Autore
Gioacchino Marfia

Anno Accademico 2013-2014

Indice

Capitolo 1

Introduzione

I *design pattern* furono introdotti in origine per descrivere una soluzione semplice ed elegante ad un problema specifico nella progettazione di software *object-oriented*. Tuttavia, l'idea che sta alla loro base è completamente generale e può essere applicata a differenti modelli di programmazione. Un *design pattern* è una rappresentazione di un problema di programmazione comune con una soluzione efficiente e testata per quel problema. Gli elementi principali di un pattern sono: un nome, una descrizione del problema ed una descrizione della soluzione. L'uso dei pattern permette di avere una metodologia di programmazione molto efficace, migliorando notevolmente il riuso e la manutenibilità del codice rispetto a quello ottenibile non utilizzandoli.

Uno degli aspetti più interessanti dei *parallel design pattern*, descritto in [20], è il partizionamento dello spazio di progettazione di un'applicazione parallela in quattro parti: il *finding concurrency*, l'*algorithm structure*, il *supporting structure* ed l'*implementation mechanisms*. Ognuno di questi contiene parecchi pattern modellanti le forme di parallelismo presenti a ciascun livello di astrazione nello spazio di progettazione. Tale partizionamento permette di separare le problematiche in due distinti domini di lavoro: *domain*

specific (ai quali appartengono i primi due spazi di progettazione) e *platform specific* (contenente i restanti due).

L'obiettivo della tesi è quello di estendere il pattern *pool evolution* [4], appartenente allo spazio di progettazione *algorithm structure*, presente nel framework FastFlow [3, 6, 10, 13]. Questa estensione permette di ampliare l'applicabilità del pattern a tutti quei problemi la cui soluzione è ricavabile dal lavoro svolto su più popolazioni, con l'obiettivo di aumentare la velocità e/o la qualità della soluzione trovata. A tale categoria appartegono gli algoritmi genetici.

Gli algoritmi genetici sono una sottoclasse degli algoritmi evolutivi, i quali costituiscono un paradigma computazionale per la risoluzione approssimata di problemi complessi. La loro implementazione permette la simulazione di sistemi dinamici che evolvono verso una condizione di stabilità. Gli algoritmi genetici operano su una popolazione di potenziali soluzioni applicando il principio della sopravvivenza del migliore, evolvendo verso una soluzione che si spera si avvicini quanto più possibile alla reale soluzione del problema. Ad ogni generazione si crea un nuovo insieme di soluzioni mediante un processo di selezione che, basandosi sulla loro bontà (attribuita dalla funzione di valutazione, detta *fitness*), sceglie i migliori individui della popolazione e li fa evolvere utilizzando una serie di operatori genetici.

Tra i modelli principali per l'implementazione degli algortimi genetici paralleli vi è il modello "ad isole". Non è ancora ben chiaro se il modello ad isole porti realmente a dei risultati in termini di convergenza. È noto piuttosto, che il modello si presta in misure diverse ad essere utilizzato su problemi differenti e nello specifico in quelli particolarmente sensibili alla variabilità delle soluzioni od in cui si può facilmente convergere verso minimi locali. In particolare si considera una variante del modello ad isole che prevede lo

scambio di informazioni, fra queste, per aumentare la variabilità dell'intera popolazione e per ridurre il rischio del fenomeno di stagnazione (convergenza verso minimi locali).

Nella tesi si forniscono due implementazioni del modello ad isole: la prima operante su sottopopolazioni, la seconda operante su una singola multi-popolazione. Entrambe queste versioni sono confrontate con il pattern *pool evolution* di FastFlow, analizzando le performance ottenute e sottolineando - ove necessario - gli aspetti che inducono a preferire le nuove versioni alla vecchia. A tal fine, si utilizzano un certo numero di applicazioni sviluppate secondo il paradigma di programmazione genetica. I risultati osservati mostrano che in caso di computazioni a grana fine, la fase di selezione degli individui può trasformarsi in un vero e proprio "collo di bottiglia". Successivamente sarà spiegato che, soprattutto in queste situazioni, le nuove versioni fornite garantiscono migliori risultati in termini di tempo di completamento, scalabilità ed efficienza.

La tesi è strutturata come illustrato di seguito. Nel capitolo 2, si presentano gli skeleton algoritmici e le varie tipologie di skeleton: data, control e stream parallel. Di quest'ultimo tipo di skeleton si introducono i due esempi principali: il *pipeline* ed il *farm*. Il capitolo chiarisce sia alcuni aspetti implementativi su come sfruttare: l'implementazione di uno skeleton, l'uso della cache; sia alcuni aspetti teorici della programmazione parallela come: il codice funzionale e non funzionale, le misure di performance (usate per stimare, predire e confrontare le soluzioni parallele proposte) e, per finire, le metodologie usate per identificare nuovi skeleton (implementati al verificarsi delle condizioni descritte nel manifesto di M. Cole [21]).

Il capitolo 3 fornisce un breve accenno ai problemi di ottimizzazione considerati in questo lavoro per comprendere il comportamento del pattern

introdotto. Si tratta dei problemi multi-obiettivo descrivendone caratteristiche, settori di impiego e soluzioni. Inoltre, si fa un breve cenno storico sull'evoluzione del paradigma computazionale sottostante gli algoritmi genetici.
Nel capitolo 4, si descrivono: il framework di programmazione parallela strutturata - FastFlow - utilizzato per implementare il nostro pattern e per ottenere i risultati sperimentali, le performance raggiungibili su questo, il significato di design pattern e più nello specifico di parallel design pattern, i pattern *pool evolution* e *parallel for*.

Il capitolo 5 tratta più nel dettaglio l'estensione fornita, operante su sottopopolazioni, soffermandosi sugli aspetti implementativi e mostrando l'evoluzione della struttura rispetto a quella iniziale.

Nel capitolo 6, si introducono le applicazioni utilizzate - sviluppate secondo il paradigma di programmazione genetica - per ottenere i risultati sperimentali mostrati. Per ognuna di queste è spiegata la codifica utilizzata in termini del nuovo pattern (nella specifica implementazione), il metodo adottato per la generazione della popolazione, il tipo di selezione e gli operatori genetici implementati, la definizione della funzione di valutazione e le condizioni di terminazione dell'algoritmo genetico.

Il capitolo 7 mostra il confronto tra i risultati delle applicazioni implementate con il modello ad isole e quelli ottenuti utilizzando il pattern *pool evolution*.

Nel capitolo 8 si definiscono due nuove classi per gli algoritmi genetici e per il modello ad isole con multi-popolazione. Si descrivono i dettagli implementativi di queste, sottolineando i motivi che hanno portato a queste due nuove definizioni. Inoltre, si mostra un confronto tra la versione operante su multi-popolazione e quella operante su sottopopolazioni su uno dei casi

visti nel capitolo 6. Le conclusioni ed i possibili sviluppi futuri sono discussi nel capitolo 9.

In appendice A si riportano il codice sorgente delle librerie implementate, rispettivamente A.1 per il modello ad isole con sottopopolazioni, A.2 per gli algoritmi genetici ed A.3 per il modello ad isole con multi-popolazione.

Capitolo 2

Skeleton algoritmici

Nell'implementazione di applicazioni parallele efficienti lo sforzo speso per la gestione degli aspetti non funzionali dell'applicazione è quello preponderante. Oltretutto, la conoscenza necessaria per sviluppare un'applicazione è totalmente indipendente da quella necessaria per implementare il pattern parallelo in modo efficiente. Questo fa sì che le applicazioni, che separano il codice per implementare il pattern parallelo da quello funzionale, siano più facili da sviluppare e da debuggare rispetto a quelle in cui entrambi gli aspetti non sono distinti. Gli algoritmi skeleton, introdotti da Murray Cole con la sua tesi di dottorato nel 1988 [27], furono definiti come pattern predefiniti che incapsulavano completamente la struttura di una computazione parallela. Tali pattern erano forniti al programmatore direttamente come chiamate di libreria, queste venivano usate (combinandole se possibile) fornendo parametri opportuni per costruire la propria applicazione parallela.

Da un altro punto di vista lo skeleton algoritmico può anche essere definito come una funzione di ordine superiore, che definisce un ben preciso modello di utilizzo del parallelismo, avente come parametri funzionali quelli che rappresentano la logica stessa dell'applicazione. La definizione di skele-

ton algoritmico ha assunto varie forme nel corso degli anni. In [9], è data la seguente definizione:

"uno skeleton algoritmico è un'astrazione di programmazione, parametrica, riusabile e portabile che modella un pattern parallelo noto, comune ed efficiente."

Gli aspetti fondamentali in questa definizione sono i seguenti:

- lo skeleton modella un pattern parallelo ben *noto* e *comune*. Non si è interessati a pattern troppo specifici che non sono comunemente impiegati nelle applicazioni. Questo non rappresenta una limitazione sostanziale in quanto la maggior parte delle computazioni parallele possono essere modellate utilizzando un piccolo numero di pattern paralleli [9];

- *portabilità*: deve essere possibile fornire un'implementazione efficiente del pattern su architetture differenti;

- *efficienza*: lo skeleton modella pattern paralleli che hanno un'implementazione efficiente sulle varie architetture;

- *riusabilità*: un pattern può essere utilizzato in differenti contesti/applicazioni senza dovergli apportare alcuna modifica;

- *parametricità*: lo skeleton è parametrico perché può specializzare il proprio comportamento in funzione dei suoi parametri (anche funzionali). Una delle più importanti caratteristiche degli skeleton implementati secondo un modello RISC [9], è la possibilità di definire una nuova struttura mediante annidamento degli skeleton elementari. Pertanto, uno skeleton può essere un parametro di un altro skeleton ed il risultato è una composizione di skeleton.

Definiamo con "programmazione parallela strutturata" [27] la programmazione parallela ottenuta utilizzando gli skeleton algoritmici. Scrivere un'applicazione parallela usando un framework specifico, basato su skeleton algoritmici, consiste nell'istanziare uno skeleton (od una composizione di skeleton) adatto alla risoluzione del particolare problema da risolvere e passare a questo (questi nel caso di composizione) la funzione (le funzioni) che ogni suo modulo deve eseguire.

Lo sviluppo di un'applicazione parallela con gli skeleton offre notevoli vantaggi rispetto a quello ottenuto con i tradizionali framework di programmazione, come MPI.

L'implementazione di una applicazione parallela utilizzando i framework strutturati richiede di seguire i seguenti passi:

1. valutare le possibili strutture parallele dell'applicazione;
2. individuare uno skeleton (od una composizione) che modella la struttura dell'applicazione parallela;
3. produrre il codice dell'applicazione parallela instanziando lo skeleton scelto, fornendo i parametri funzionali e non funzionali;
4. compilare l'applicazione;
5. eseguire l'applicazione sull'architettura target;
6. analizzare i risultati dal punto di vista funzionale e non funzionale;
7. in caso di inefficienze, migliorare la struttura attuale o scegliere un altro skeleton ripartendo dal punto 3.

Così facendo si evita al programmatore di doversi fare carico della gestione di tutti i dettagli necessari a sfruttare al meglio il parallelismo disponibile. Si semplificano in tal modo quelli che sono tutti gli aspetti di:

- gestione e verifica della correttezza;
- efficienza;
- portabilità funzionale e di performance;
- debugging;
- ottimizzazione;
- running dell'applicazione.

Questi aspetti, infatti, diventano di responsabilità di chi implementa il framework di programmazione parallela usato.
Gli skeleton possono essere suddivisi nelle seguenti classi:

- *data parallel*: l'elemento di input che si vuole processare, viene suddiviso in varie parti ed ogni computazione relativa ad una sottoparte viene eseguita in parallelo. Gli skeleton data parallel si usano per velocizzare una singola computazione spezzandola nella computazione di più sotto-task in parallelo. I più comuni skeleton data parallel sono la *map*, la *reduce*, il *parallel prefix* e lo *stencil* [9];
- *stream parallel*: sia uno *stream* una sequenza di lunghezza non nota di item/task disponibili uno dopo l'altro. In questi skeleton, il parallelismo è ottenuto grazie alla computazione simultanea su elementi differenti ed indipendenti che si presentano ad istanti di tempo successivi sullo stream di input. Ciascuna computazione termina con la spedizione di un singolo oggetto sullo stream di output. I più comuni skeleton stream parallel sono il *pipeline* ed il *farm* che verranno descritti nelle sezioni 2.1 e 2.2;

- *control parallel*: esprime pattern strutturati di coordinazione e modella le parti di una computazione parallela necessarie a supportare o coordinare gli skeleton paralleli. Include: *sequential skeleton* usato per incapsulare porzioni di codice sequenziale in modo da poter passare questo come parametro ad altri skeleton; *conditional* (per modellare *if-then-else*) ed *iterative skeleton* (per esempio: *forall skeleton*).

Nei vari framework di programmazione parallela gli skeleton sono offerti al programmatore principalmente in due modi alternativi:

- *costrutti primitivi*: gli skeleton vengono definiti come costrutti primitivi del linguaggio. È quindi necessario definire un nuovo linguaggio di programmazione;

- *libreria*: gli skeleton sono implementati semplicemente attraverso chiamate di libreria. Pertanto, scelto un determinato linguaggio di programmazione, gli skeleton sono implementati sopra tale linguaggio sfruttando i meccanismi di astrazione disponibili nel linguaggio.

2.1 Pipeline

Il pattern parallelo *pipeline* applica la funzione dello stage i-esimo ai risultati prodotti dalla funzione computata dallo stage *(i-1)*-esimo sui dati di input. In un *pipeline* a due stadi, dove il primo calcola f ed il secondo g, il calcolo parallelo consiste quindi nel computare $g(f(x_i))$ e $f(x_{i+1})$ contemporaneamente, cioè nel lavorare contemporaneamente su oggetti diversi dell'input stream.

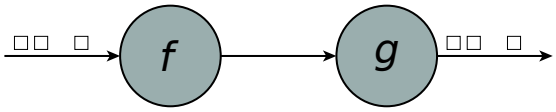


Figura 2.1: *Pipeline.*

Il grado di parallelismo è facilmente determinabile: corrisponde al numero di funzioni da applicare o stage da far seguire sui dati di input.

Sia T_S il tempo di servizio, ovvero il tempo impiegato per produrre due output successivi (oppure il tempo richiesto per accettare due input consecutivi). In un *pipeline* che lavora a regime tralasciando la fase iniziale in cui gli stage devono acquisire il primo task su cui lavorare, il tempo T_S è approssimabile al tempo speso per calcolare lo stage avente T_S maggiore nell'intero *pipeline*.

2.2 Farm

Il pattern parallelo *farm* modella computazioni *embarassingly parallel* su stream. Per chiarire meglio, possiamo pensare al pattern *farm* come una *map*[1] operante su uno stream. A differenza del *pipeline*, in un *farm* il grado di parallelismo non è deducibile dalla struttura della computazione, ma costituisce un parametro relativamente indipendente. Infatti, lavorando su uno stream di dati non si sa né quanti dati né quando questi si presenteranno in input. Una possibile implementazione di questo pattern prevede di definire tre diversi oggetti: un emitter E, un certo numero di worker n_w ed un collector C. Nel dettaglio:

[1]Skeleton data parallel usato per accelerare una singola computazione dividendola in sotto-task computati parallelamente.

- l'emitter E si occupa di leggere sul canale di input gli elementi x_1, x_2, ..., x_n e di schedularli verso i worker - con un'opportuna politica - al fine di garantire una corretta gestione del bilanciamento di carico fra i vari worker;
- il generico worker w_i ha il compito di applicare la funzione f all'elemento x_i che riceve dall'emitter E, spedendo il risultato $f(x_i)$ al collector C;
- il collector C acquisisce i risultati calcolati dai vari worker e li invia sul proprio canale (stream) di output.

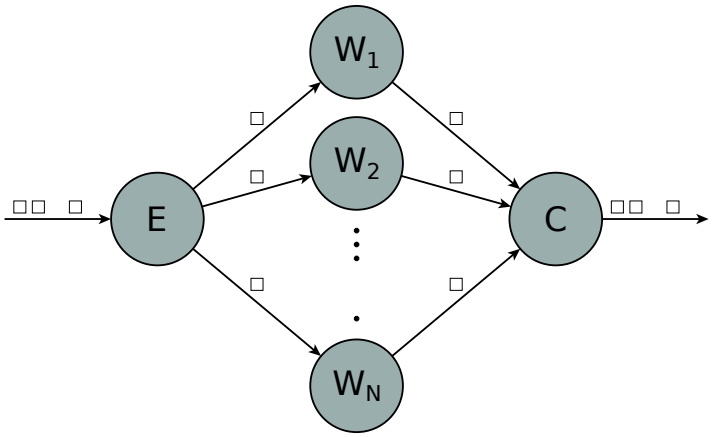


Figura 2.2: *Farm.*

La semantica di tale skeleton consiste quindi nell'applicare in parallelo la stessa funzione f a più task dell'input stream usando un certo numero, n_w, di worker. Il tempo che intercorre tra la spedizione di due risultati consecutivi può essere approssimato con il tempo speso per calcolare sequenzialmente la funzione f diviso il numero n_w di worker utilizzati.

L'emitter può usare varie strategie per schedulare i task verso i worker, fra cui:

- una strategia *round robin*. L'emitter E invia gli elementi x_i in modo ciclico a partire da 1 fino all'n-esimo worker. Se l'emitter E deve inviare l'elemento x_i, questo verrà spedito a quello con $id = i \ mod \ n_w$. Questa strategia è preferibile nelle situazioni in cui il tempo di servizio ha una bassa varianza fra i worker;

- una strategia *on demand*. In questo caso ogni worker comunica all'emitter E di aver terminato la computazione sul task precedentemente ricevuto e quindi di essere disponibile a ricevere un altro task su cui poter lavorare. L'emitter controlla se è presente un task sullo stream di input ed eventualmente lo spedisce ad uno dei worker che ne hanno fatto richiesta. Questa politica risulta utile nel caso in cui il lavoro compiuto dai worker abbia un tempo di servizio variabile. Infatti, essa tende a garantire miglior bilanciamento di carico ed a sfruttare al massimo le risorse disponibili riducendo il tempo di inattività dei worker. In una situazione di non-bilanciamento di carico occorre aspettare che la risorsa con la computazione più lunga termini prima di produrre tutti i risultati. Durante questa attesa, le altre risorse potrebbero restare inattive nel caso in cui abbiano già terminato la loro computazione. Il risultato è che si ha un tempo di completamento T_C, ovvero il tempo speso dall'inizio dell'applicazione fino alla sua terminazione, maggiore di quanto si possa ottenere in situazioni con bilanciamento di carico.
  Va però considerato che questa soluzione aggiunge chiaramente dei costi di comunicazione ulteriori rispetto alla *round robin*. Infatti, oltre al canale di comunicazione unidirezionale fra l'emitter E e tutti i worker, è necessario disporre di canali di comunicazione anche nella direzione opposta (per permettere al worker di inoltrare la richiesta, di essere pronto a ricevere ulteriori task, all'emitter).

FastFlow, descritto nel capitolo 4, offre parecchie possibilità di personalizzazione: è possibile definire un *farm* senza emitter o senza collector, ovvero con E o C inglobati negli stadi che precedono o seguono il *farm*. È anche possibile definire una politica di schedulazione dei task per l'emitter differente dalle due descritte sopra. Nel caso in cui si utilizzi il canale di feedback[2], si può - per esempio - implementare una strategia che prevede di spedire il task analizzato dal worker i ancora allo stesso worker i. Tale scelta punta a sfruttare al massimo la località favorendo la gestione dei dati in cache. Il processore trova infatti, che la posizione di memoria è in cache (*cache-hit*) ed evita così inutili trasferimenti dalla memoria principale dovuti a *cache-miss*.

La sezione 2.4 discute una tecnica di implementazione adatta a sfruttare al meglio la cache.

2.3 Implementazioni di uno skeleton

Alcune implementazioni di uno skeleton sono particolarmente significative e possono essere usate per supportare differenti skeleton. Inoltre, uno skeleton può essere implementato in termini di un altro skeleton. Per esempio, in uno skeleton data parallel si divide la struttura dati di input in partizioni e si opera in parallelo su queste. L'output è ottenuto combinando tutti i risultati delle partizioni. È possibile implementare pattern data parallel come la *map* o la *reduce* utilizzando i pattern stream parallel descritti prima. Quindi preso un dato in input si considera un *pipeline* di tre stage, in cui:

- il primo stage: crea le partizioni e le spedisce sul canale di input come se questo fosse uno stream di dati;

[2]Canale usato per rispedire i task dal collector (oppure, nel caso in cui questo non è definito, dai worker) allo stream di input.

- il secondo stage: utilizza un *farm* operante su tale stream, in cui ogni worker computa su una partizione;
- il terzo stage: ricompone la struttura dati collezionando tutti i task, formando così l'output dello skeleton data parallel.

Questa trasformazione, da uno skeleton ad un altro skeleton, è definita come *cross-skeleton template* in [9]. Consideriamo un altro esempio. Prendiamo il pattern *farm* visto nella sezione precedente ed aggiungiamo a questo un canale di feedback ottenendo un'implementazione adatta a rappresentare diversi pattern (sfruttando semplicemente la sola implementazione del *farm*). Il framework FastFlow (vedi capitolo 4) fornisce la possibilità di attivare questo canale di feedback che permette al collector C, se presente, oppure ai worker di spedire gli output nuovamente allo stream di input. Tale struttura può essere utilizzata per rappresentare quindi più pattern, è infatti possibile lavorare sui task per un numero x di volte (come nel pattern *pool evolution*, descritto nel capitolo 4.3, per esempio) o rappresentare altri pattern più complessi come il *divide and conquer*.

2.4 Pinning dei thread

Nei sistemi multiprocessore oppure in quelli multi-core, il sistema operativo distribuisce i diversi thread nelle risorse disponibili. Nel momento in cui l'esecuzione di un thread deve essere riattivata dal sistema operativo, l'algoritmo di schedulazione in generale ha la possibilità di allocare il thread su uno dei core a disposizione del sistema indipendentemente da dove fosse stato allocato nell'esecuzione precedente. In alcuni casi, al fine di ottenere migliori prestazioni (per esempio per aumentare l'efficienza nell'uso della cache),

si vuole evitare questa situazione permettendo di scegliere dove collocare in modo permanente un thread.

Infatti, se non si pone nessun vincolo, ogni qual volta un thread cambia core occorre spostare i dati nella nuova cache per farla funzionare a pieno regime. Tale operazione ha un costo pari alle latenze dovute ai *cache-miss* incontrati. Se, invece, il thread fosse schedulato sempre sullo stesso core, la probabilità che nella cache siano ancora presenti i dati relativi alla sua computazione sarebbe notevolmente maggiore. In questo caso si ridurrebbe chiaramente il numero di *cache-miss* incontrati, migliorando le prestazioni complessive anche in maniera significativa.

Sia X un core specifico. Un altro motivo per vincolare un thread ad X è il caso in cui questo debba svolgere un lavoro fondamentale per il sistema. Solitamente, non solo si preferisce che il thread operi solo su X ma si vorrebbe impedire che altri thread possano essere schedulati su X.

Il pinning dei thread consiste quindi nel mappare ognuno dei thread utilizzati per implementare la computazione parallela su un determinato core o più in generale su un determinato insieme di core, modificando il comportamento originario dell'algoritmo di schedulazione utilizzato nella macchina in questione. Il kernel Linux fornisce un'insieme di funzioni per permettere questa assegnazione, definendo un *affinity set* per ogni thread. Lo schedulatore allocherà quindi il thread solo su uno dei core appartenenti al suo *affinity set*. Nelle applicazioni usate, tale tecnica è utilizzata sulla macchina *Xeon Phi* sfruttando la funzione *threadMapper::instance()* $\rightarrow$ *setMappingList(const char str[])* presente in FastFlow (vedi capitolo 4).

Se più thread sono mappati su uno stesso core, va tenuto conto che questi non opereranno in parallelo ma, chiaramente, lo schedulatore li alternererà nelle loro esecuzioni - parallelismo virtuale. In questo caso, l'*overhead* do-

vuto all'alternarsi dei thread deve essere preso in considerazione. Questa condizione può presentarsi per esempio:

- nel caso in cui le risorse richieste (numero di thread) siano maggiori di quelle effettivamente disponibili (numero di core);
- nel caso in cui l'*affinity set* definito non sfrutta correttamente la numerazione dei core della macchina target su cui è in esecuzione l'applicazione. Consideriamo, per esempio, una macchina con quattro core aventi ciascuno quattro thread numerati consecutivamente. Pertanto con i numeri da 1 a 4 indichiamo tutti i thread del primo core, con i numeri da 5 a 8 quelli del secondo etc... Se l'*affinity set* della nostra applicazione è $\{1, 2, 3, 4\}$, allora, i thread saranno assegnati sullo stesso core, alternandosi nell'esecuzione piuttosto che sfruttare gli altri core disponibili lavorando in parallelo.

2.5 Codice funzionale e non funzionale

Il codice necessario per implementare un'applicazione parallela è composto da due distinti tipi di codice molto differenti fra di loro:

1. *codice funzionale*: rappresenta il codice necessario per calcolare i risultati della computazione parallela, cioè la funzione f da applicare sui dati in input per calcolare il risultato. Questo codice dipende dal tipo dell'applicazione;
2. *codice non funzionale*: rappresenta il codice necessario per orchestrare la computazione di quello funzionale in parallelo. Include il codice per installare le attività parallele, per schedulare queste attività sulle risorse disponibili e per coordinare la loro esecuzione fino al completamento

dell'applicazione. Questo codice dipende dall'architettura target e dal framework di programmazione parallela utilizzato.

In generale, il primo rappresenta il codice necessario per calcolare *cosa* l'applicazione realmente computa, mentre il secondo è il codice necessario per determinare *come* questi risultati funzionali sono realmente calcolati in parallelo.

2.6 Modelli di performance

La motivazione che spinge a realizzare ed affrontare un problema in modo parallelo è il raggiungimento di prestazioni migliori. Non tutti i problemi sono adatti per essere parallelizzati. In generale ci si aspetta di dedicare, nell'esecuzione parallela con grado di parallelismo pari ad n, un tempo pari ad $\frac{1}{n}$ (con n grado di parallelismo) rispetto a quello della versione sequenziale. Parecchi fattori impediscono però di raggiungere tale comportamento, primo fra tutti l'*overhead*. L'*overhead*, inteso come tempo perso nel calcolo della parte funzionale dell'applicazione parallela, è causato dalla gestione del codice non funzionale ed è purtroppo dipendente dal grado di parallelismo utilizzato.

Il poter stabilire un limite teorico alle performance permette di avere notevoli vantaggi:

1. la possibilità di decidere se conviene o meno rappresentare la soluzione di un problema in un determinato modo (quindi essere in grado di predire le performance di un'applicazione);

2. la possibilità di scegliere la soluzione più semplice nel caso in cui un problema possa essere risolto in diversi modi (possibilità di confrontare più soluzioni);

3. la possibilità di stimare/valutare l'*overhead* introdotto.

A tal fine, non basta applicare la legge di *Amdhal* (vedi sezione 2.6.1) ma occorre utilizzare opportuni "modelli di performance". Questi permettono di misurare le performance di un'applicazione su una data macchina in funzione di una serie di parametri dipendenti dall'applicazione o dalla macchina target. Consideriamo due distinte categorie di misure, quelle che:

- misurano il tempo assoluto speso nell'esecuzione di un'applicazione parallela;
- misurano la velocità nello spedire i risultati.

Nel primo insieme rientrano le misure di *latenza* e di *tempo di completamento*, rispettivamente il tempo per calcolare l'output su un input e la latenza complessiva.

Nel secondo insieme abbiamo il *tempo di servizio*, ovvero il tempo che intercorre tra lo spedire due output consecutivi o tra accettare due input consecutivi, e la *larghezza di banda* che è l'inverso del tempo di servizio.

Solitamente si è però interessati a delle misure di performance derivate, quali: *speedup*, *scalabilità* ed *efficienza*.

2.6.1 Legge di Amdhal

La legge di *Amdhal* stabilisce che la quantità di lavoro non parallelizzabile di un'applicazione determina il massimo *speedup* teorico raggiungibile da questa applicazione. Se si ha una percentuale f di lavoro inerentemente sequenziale allora tale parte non può essere parallelizzata. Sia T il tempo richiesto per eseguire l'intero task sequenzialmente, $f \cdot T$ sarà il tempo per eseguire la parte seriale mentre $(1 - f) \cdot T$ sarà la frazione parallelizzabile.

Quindi, nel migliore dei casi, è quest'ultimo termine che può essere ridotto opportunamente sfruttando il parallelismo diventando $\frac{(1-f)\cdot T}{n}$.

La definizione di *speedup* data, ovvero $sp(n) = \frac{T_{\text{seq}}}{T(n)}$, secondo quanto affermato sopra si modifica diventando:

$$sp(n) = \frac{T}{f \cdot T + (1-f) \cdot \frac{T}{n}}$$

Al tendere del grado di parallelismo ad infinito:

$$\lim_{n\to\infty} sp(n) = \frac{T}{f \cdot T} = \frac{1}{f}$$

Ovvero, se si ha una percentuale f di lavoro inerentemente seriale allora non si può ottenere uno *speedup* maggiore di $\frac{1}{f}$.

Tutte queste considerazioni valgono in ambito teorico, nella pratica la situazione peggiora. Infatti tutto ciò che è necessario per sfruttare il parallelismo aggiunge un costo, l'*overhead*. Il limite precedente diventa:

$$\lim_{n\to\infty} sp(n) = \lim_{n\to\infty} \frac{T}{f \cdot T + (1-f) \cdot (\frac{T}{n} + T_{\text{ov}}(n))} = \frac{1}{f + (1-f) \cdot \frac{T_{\text{ov}}(n)}{T}}$$

2.6.2 Speedup

Lo *speedup* si esprime come il rapporto tra il miglior tempo di servizio T_S (o il tempo di completamento T_C) dell'esecuzione sequenziale e quello ottenuto dall'esecuzione parallela con grado di parallelismo pari ad n, ovvero:

$$sp(n) = \frac{T_{\text{seq}}}{T(n)}$$

Esso fornisce una misura di quanto è buona la parallelizzazione rispetto alla migliore soluzione sequenziale. Lo *speedup* è limitato superiormente dall'equazione $f(n) = n$, questo perchè se $sp(n) > n$ allora per la definizione data si avrebbe che il tempo parallelo $t_{\text{p}} < \frac{T_{\text{seq}}}{n}$ portando ad un assurdo: T_{seq} non sarebbe il miglior tempo sequenziale ma lo sarebbe $n \cdot T_{\text{p}}$.

Ci si aspetta quindi, che lo *speedup* non superi mai il grado di parallelismo usato. Tuttavia, in alcune situazioni è possibile ottenere uno *speedup super-lineare*. Per esempio, consideriamo tutti quei casi in cui si riesce a sfruttare nell'esecuzione parallela la cache meglio di quanto non si riesca a farlo nell'esecuzione sequenziale: quel che accade è che i dati non riescono a sfruttare bene la cache nella versione sequenziale ma riescono a farlo in quella parallela; i *cache-hit* (ovvero il processore trova che la posizione di memoria richiesta è in cache) riducono gli accessi alla memoria principale portando un guadagno che potrebbe essere più grande dell'*overhead* necessario per installare le attività parallele e distribuire loro i dati.

2.6.3 Scalabilità

Un'altra misura derivata è la *scalabilità*, ovvero il rapporto tra il tempo di servizio T_S (od il tempo di completamento T_C) ottenuto eseguendo l'applicazione parallela con grado di parallelismo pari ad 1 ed il tempo ottenuto dalla stessa applicazione con grado di parallelismo pari ad n.

$$scalab(n) = \frac{T_{\text{par}}(1)}{T_{\text{par}}(n)}$$

È chiaramente diversa dallo *speedup* perchè, anziché fornire un confronto con la migliore versione sequenziale, misura quanto è efficiente l'implementazione parallela nel raggiungere migliori performance su gradi di parallelismo sempre più grandi. Si possono avere, quindi, applicazioni con ottima *scalabilità* ma pessimo *speedup*, ovvero applicazioni in cui è maggiore il tempo impiegato con la versione parallela rispetto a quello impiegato con la versione sequenziale. Per gli stessi motivi dello *speedup*, tale misura è limitata dalla funzione $f(n) = n$.

2.6.4 Efficienza

L'*efficienza* è il rapporto fra il tempo di esecuzione ideale e quello dell'esecuzione parallela con grado di parallelismo pari ad n o se vogliamo:

$$\frac{sp(n)}{n}$$

L'efficienza misura la capacità dell'applicazione parallela di sfruttare le risorse disponibili.

Essendo per definizione legata allo *speedup* anche questa misura presenta un upper bound, cioè la funzione $f(n) = 1$. Non è possibile confrontare due applicazioni conoscendone soltanto il valore di *efficienza*, a meno che queste non abbiano lo stesso grado di parallelismo.

2.7 Identificare nuovi skeleton

L'identificazione di nuovi skeleton può avvenire in seguito a due motivi: il primo basato sull'analisi delle applicazioni parallele esistenti, il secondo basato sulle necessità di implementare una particolare applicazione. Entrando nel dettaglio, nel primo caso, l'idea consiste nel cercare nuovi skeleton analizzando delle applicazioni parallele note, quindi:

1. si selezionano un certo numero di applicazioni il cui codice è accessibile ed/od il cui algoritmo parallelo sia chiaro;

2. si analizzano queste applicazioni cercando di individuare nuovi skeleton. Questi si dividono in *domain specific*, ovvero skeleton che servono per un solo dominio, e *general purpose* (per più domini);

3. si analizzano e si ricavano i pattern paralleli usati in queste applicazioni, attraverso un lavoro di vero e proprio *reverse engineering*;

4. dall'output del punto precedente si identificano gli skeleton che non appartengono all'insieme utilizzato;

5. si verifica se questo è uno skeleton singolo oppure se è ricavato dalla composizione di più skeleton, in tal caso si prova a decomporlo fino alle singole unità. A questo punto si verifica che: modelli un pattern parallelo comune ed efficiente, sia facile da usare, sia definito da un insieme di parametri globali, tolleri possibili variazioni fatte dall'utente e sia implementabile efficientemente su un range di architetture parallele (requisiti derivati dal "manifesto" di Cole [21]);

6. si definiscono: nome, sintassi, insieme di parametri funzionali e non, semantica funzionale e parallela;

7. finita la fase di perfezionamento del nuovo skeleton, si procede all'implementazione.

Nel secondo caso, la necessità di definire un nuovo skeleton nasce durante l'implementazione di una specifica applicazione parallela. Se durante l'analisi si identifica un pattern, quindi un modello tipico e riusabile in altri contesti, e tale pattern non è né presente nel framework attuale né riproducibile attraverso una combinazione degli skeleton presenti, allora, si procede come segue:

1. si effettua un'analisi come quella del punto 5 precedente, ovvero si verifica che il nuovo skeleton sia composizione di più skeleton o sia già uno skeleton base;

2. si cercano altri casi in cui tale skeleton possa essere utile. Si seguono i primi tre punti visti sopra e se un certo numero di situazioni sono

individuate, quindi possono essere rappresentate con il nuovo pattern, si procede alla sua definizione;

3. si perfeziona la definizione dello skeleton considerando le necessità individuate nelle altre situazioni dove il nuovo pattern potrebbe essere utilizzato. Si forniscono per finire: nome, sintassi, parametri e semantica;

4. finita la fase di perfezionamento del nuovo skeleton, si procede all'implementazione.

Capitolo 3

Ottimizzazione

In questo capitolo si descrivono due diverse tipologie di problemi di ottimizzazione. Come infatti si vedrà nel capitolo 6, le applicazioni sviluppate rientrano nei problemi di ottimizzazione multi-obiettivo ed in quelli programmati secondo il paradigma di programmazione genetica. Per quanto riguarda le ottimizzazioni multi-obiettivo se ne espongono brevemente la loro importanza, gli ambiti di utilizzo, le differenze dai problemi mono-obiettivo, le metodologie di risoluzione e si accenna agli algoritmi presenti in letteratura. Nella seconda sezione si discutono gli algoritmi genetici: ad una prima parte riguardante l'evoluzione storica segue una descrizione degli aspetti principali che caratterizzano tale tipologia di algoritmi.

3.1 Problemi multi-obiettivo

La programmazione matematica classica, lineare (PL) o intera (PLI), tratta problemi caratterizzati da un'unica e ben definita funzione obiettivo. Tuttavia, esistono numerosi settori dove i problemi di ottimizzazione lavorano su più obiettivi contemporaneamente:

- la pianificazione del trasporto di merci e/o di persone;
- il calcolo dei percorsi;
- lo scheduling;
- la scelta di investimenti;
- la scelta di progetti da sviluppare.

In tutti questi campi si possono voler ottimizzare contemporaneamente diversi obiettivi, come per esempio: il tempo totale di esecuzione, la qualità, il rischio, il bilanciamento del carico di lavoro, i ritardi nelle consegne, il profitto, il costo totale, etc...

In generale in un problema multi-obiettivo non esiste una soluzione che ottimizza tutte le funzioni. Inoltre, i vari obiettivi sono generalmente in contrapposizione fra loro. Bisogna, pertanto, definire l'insieme delle soluzioni a cui siamo interessati e considerare nuovi metodi ad hoc per calcolare queste soluzioni (differentemente da quanto fatto in quelli a singola ottimizzazione dove si minimizzava/massimizzava una sola funzione). Chiaramente la complessità di uno stesso problema cresce all'aumentare degli obiettivi forniti. In queste situazioni lo scopo è individuare uno specifico *ottimo di Pareto* [1, 5, 18, 25] ma, come vedremo, questo può richiedere di generare tutta la *frontiera* od un suo sottoinsieme. Sia x soluzione al problema multi-obiettivo, diciamo allora che x è un *ottimo di Pareto* se non esiste nessuna soluzione y che domina x.

La *dominanza* è un concetto cruciale per gli algoritmi multi-obiettivo. Sia f_i l'i-esima funzione da massimizzare (o minimizzare), una soluzione x_1 domina un'altra soluzione x_2 (nel caso di problemi di massimo) se valgono entrambe le relazioni:

1. $f_{\mathrm{i}}(x_1) \geq f_{\mathrm{i}}(x_2)$, per i=1...n;

2. $f_{\mathrm{i}}(x_1) > f_{\mathrm{i}}(x_2)$, per almeno un valore di i.

È importante notare come la *dominanza* va verificata per ciascuna funzione obiettivo, affermare che la soluzione x_1 domina la soluzione x_2 per un solo obiettivo i non basta per stabilire quale delle due preferire (non si ha nessuna informazione sulle restanti n-1 funzioni).

Definiamo con *fronte di Pareto* l'insieme delle soluzioni ottime, quello formato da tutti i punti non dominati, ovvero da quei punti per i quali non ne esiste nessun altro che sia migliore contemporaneamente per tutti gli obiettivi considerati nella funzione di ottimizzazione.

Più in generale, si definisce *fronte* l'insieme formato dagli elementi che non si dominano a vicenda; per cui al primo *fronte* faranno parte tutte quelle soluzioni che dominano tutte le altre, al secondo quelle dominate dal primo ma che a loro volta dominano tutte le altre e così via. Un punto può far parte del *fronte di Pareto* anche se non domina nessuno poiché l'importante è che non sia dominato da altri.

Come anticipato prima, nell'identificazione dell'*ottimo di Pareto* si può essere costretti a generare tutta la *frontiera* oppure un suo sottoinsieme. A tal fine, si assume l'esistenza di un decisore in grado di selezionare i punti migliori. In base al ruolo svolto dal decisore nella strategia di soluzione del problema, i metodi risolutivi possono essere suddivisi in cinque categorie:

- *metodi senza preferenze*: il decisore non ha nessun ruolo e si considera soddisfacente l'aver trovato un qualunque *ottimo di Pareto*;

- *metodi a posteriori*: si genera l'insieme di tutti gli *ottimi di Pareto*, tale output viene passato al decisore che sceglierà la soluzione più opportuna. L'algoritmo ε-*constrained* [5, 18] - per esempio - è un metodo

a posteriori che lavora ottimizzando una sola funzione e trasformando le altre in vincoli. La soluzione ottima del problema trasformato in mono-obiettivo non è necessariamente un *ottimo di Pareto* del problema multi-obiettivo. Tuttavia, risolvendo iterativamente con valori ε_i opportunamente aggiornati, è possibile generare l'intera *frontiera di Pareto*. Parecchi algoritmi sono stati sviluppati per questa categoria, per quanto riguarda il paradigma evolutivo ricordiamo: *EMO*, *SPEA-2* e *NSGA-II* [2];

- *metodi a priori*: il decisore fornisce le direttive per poter stabilire quale è la migliore soluzione. Tali direttive indirizzano la ricerca verso la soluzione migliore senza dover necessariamente generare tutti gli *ottimi di Pareto*. Un metodo in questa categoria è il *goal programming* [18] in cui il decisore fornisce dei valori target ed il problema diventa quello di minimizzare la distanza delle funzioni obiettivo da questi valori. Un'altra metodologia, la *scalarizzazione* [5], consiste nell'assegnare ad ogni funzione obiettivo un peso che trasforma il problema in mono-obiettivo. Ogni soluzione ottima del problema scalare è una soluzione efficiente (ovvero una soluzione per cui si può migliorare un obiettivo solo peggiorandone un altro) del problema multi-obiettivo;

- *metodi interattivi*: il decisore fornisce le sue specifiche durante l'evoluzione del processo di risoluzione, raffinando la ricerca verso la soluzione più adatta;

- *metodi ibridi*: ottenuti combinando gli algoritmi evolutivi con i metodi di ricerca locale. Un metodo ibrido può essere utile perchè assicura una migliore convergenza al *fronte ottimo di Pareto* e richiede uno sforzo computazionale inferiore rispetto a quello ottenuto applicando ciascun

metodo singolarmente. Combinare i due approcci è una scelta naturale. Infatti, alle buone proprietà di convergenza verso ottimi locali possedute dai metodi di ricerca locale si aggiunge la prospettiva globale degli algoritmi evolutivi [23].

Chiaramente le soluzioni ottime saranno quelle presenti sul primo *fronte* (essendo l'unico contenente elementi che non sono dominati da altri). Tuttavia, essendo l'insieme degli *ottimi di Pareto* potenzialmente infinito, tale singola *frontiera* può non essere sufficiente per rappresentare opportunamente l'insieme ottimo. A tal fine gli algoritmi sviluppati devono tener conto degli elementi presenti nei fronti successivi, il modo in cui si selezionano gli elementi da più fronti è strettamente dipendente dall'algoritmo utilizzato.

3.2 Algoritmi genetici

Gli algoritmi genetici sono una sottoclasse degli algoritmi evolutivi, i quali costituiscono un paradigma computazionale per la risoluzione approssimata di problemi complessi. La loro implementazione permette la simulazione di sistemi dinamici che evolvono verso una condizione di stabilità (massimo/minimo). Tra la fine degli anni '50 e l'inizio degli anni '60 i ricercatori nel campo della computazione evolutiva cominciarono ad interessarsi ai sistemi naturali nella convinzione che potessero costituire un modello per nuovi algoritmi di ottimizzazione.

In quest'ottica i meccanismi evolutivi possono essere adatti per affrontare alcuni problemi computazionali complessi riguardanti la ricerca della soluzione tra un numero enorme di alternative, come per esempio, l'individuazione di una proteina con determinate caratteristiche tra un numero elevatissimo di possibili sequenze di amminoacidi; oppure la ricerca di una serie di regole o

equazioni che permettano di prevedere l'andamento dei mercati finanziari, del clima, etc... Algoritmi progettati per la soluzione di tali problemi dovranno essere adattivi, quindi in grado di "modellare" il proprio "comportamento" a situazioni altamente variabili. Da questo punto di vista gli organismi viventi possono essere considerati ottimi risolutori di problemi, poichè sono in grado di sopravvivere nel loro ambiente, sviluppando comportamenti ed abilità che sono il risultato della naturale evoluzione.

L'evoluzione biologica è assimilabile a un metodo di ricerca all'interno di un grandissimo numero di soluzioni, costituite dall'insieme di tutte le sequenze genetiche. I risultati, ovvero le soluzioni desiderate, sono organismi altamente adattati e quindi dotati di forte capacità di sopravvivenza e di riproduzione in un ambiente mutevole; pertanto, essi sono in grado di trasmettere alle generazioni future il loro materiale genetico. Essenzialmente, l'evoluzione di una specie è regolata da due processi fondamentali: la selezione naturale e la riproduzione. Quest'ultima determina la ricombinazione del materiale genetico dei genitori generando un'evoluzione molto più rapida di quella che si otterrebbe se tutti i discendenti contenessero semplicemente una copia dei geni di un genitore, modificata casualmente da una mutazione. Si tratta di un processo ad alto grado di parallelismo: non opera su un individuo per volta, ma mette alla prova e cambia milioni di individui in parallelo. In breve, quindi, un algoritmo genetico è un metodo euristico di ricerca ed ottimizzazione ispirato al principio della selezione naturale che regola l'evoluzione biologica.

Gli algoritmi genetici operano su una popolazione di potenziali soluzioni applicando il principio della sopravvivenza del migliore, evolvendo verso una soluzione che si spera si avvicini quanto più possibile alla reale soluzione del problema. Ad ogni generazione si crea un nuovo insieme di soluzioni

mediante un processo di selezione che, basandosi sulla loro bontà (attribuita dalla funzione di valutazione, detta *fitness*), sceglie i migliori individui della popolazione e li fa evolvere utilizzando una serie di operatori genetici. Sia P la popolazione e V un sottoinsieme dei numeri interi $\mathbb{N}$ o dei numeri reali $\mathbb{R}$, la *fitness* F è così definita:

$$F : P \rightarrow V$$

Assumendo di mantenere costante la dimensione della popolazione, la fase di selezione può essere rappresentata imponendo che l'uso degli operatori genetici - responsabili della creazione di nuovi individui - sia soggetto ad una probabilità. I nuovi individui andranno a sostituire un pari numero di individui, costituendo difatti la nuova popolazione per la generazione (o iterazione) successiva. Tale processo porta ad una evoluzione verso individui che meglio si adattano all'ambiente, ovvero, all'insieme di soluzioni che meglio rispondono al problema posto in partenza.

Queste soluzioni sono quelle che hanno maggiore probabilità di trasmettere i propri geni alle generazioni future. Il processo viene reiterato per un numero x di volte fino a quando si raggiunge un'approssimazione accettabile della soluzione al problema o si raggiunge il massimo numero di iterazioni fissato. Questi algoritmi sono, per esempio, utilizzati in tecniche di intelligenza artificiale, in robotica, nella bio-computazione, nello studio dell'evoluzione dei sistemi cellulari paralleli, in particolari problemi di gestione e sistemi di ottimizzazione in ingegneria.
Gli algoritmi genetici sono caratterizzati dalle seguenti proprietà:

- possibilità di risolvere problemi complessi senza una conoscenza a priori del metodo di risoluzione;

- capacità di auto-modificazione nel caso in cui si abbia un cambiamento nei parametri del problema trattato;
- capacità di evolvere il sistema delle soluzioni mediante operatori isomorfi a quelli su cui si fonda l'evoluzione biologica.

I primi algoritmi evolutivi, progettati per la soluzione di problemi di ottimizzazione, non produssero risultati convincenti poichè veniva data molta importanza alla mutazione delle soluzioni, in accordo con i test di biologia dei primi anni '60 che mettevano in risalto l'operatore della mutazione piuttosto che il processo riproduttivo per la generazione di nuovi geni. A metà degli anni '60 un progresso significativo fu segnato dalla proposta di John Holland, i cui algoritmi genetici sottolinearono per la prima volta l'importanza della riproduzione.

In alcune applicazioni gli algoritmi genetici trovano buone soluzioni in tempi ragionevoli. In altre possono impiegare giorni, mesi o anche anni per trovare una soluzione accettabile. Ma poiché essi lavorano con popolazioni di soluzioni indipendenti, è possibile distribuire il carico computazionale su più calcolatori/processori (secondo un pattern *Master/Worker*[1]), che produrranno simultaneamente diverse evoluzioni con la conseguente riduzione dei tempi di calcolo.

Gli algoritmi genetici sono frequentemente utilizzati nei problemi di ottimizzazione per i quali non si conoscono algoritmi di soluzione di complessità lineare o polinomiale. Fondamentalmente, gli elementi costituenti un algoritmo genetico sono:

[1]Pattern che modella l'esecuzione concorrente di task su una collezione di worker identici, di fatto un *farm*.

- *Popolazione*: costituita da un numero n_i di individui. Ogni individuo rappresenta una possibile soluzione al problema;

- *Fitness*: permette di valutare quanto una soluzione è adatta a risolvere il problema dato. È una funzione $F : P \to V$, dove P rappresenta la popolazione e V può essere un sottoinsieme dei numeri interi $\mathbb{N}$ o dei numeri reali $\mathbb{R}$. Quindi, a ciascun individuo è associato un valore di *fitness*, a valori di F migliori corrispondono individui migliori;

- *Principio di Selezione*: ha il compito di selezionare gli individui della popolazione (le possibili soluzioni) da sottoporre alla fase di evoluzione. I principi di selezione possono essere affidati totalmente al caso, selezionando quindi individui qualsiasi o tenendo conto del loro valore di *fitness*: per esempio la selezione per *torneo* oppure la selezione per *roulette* [25] in cui le soluzioni con *fitness* maggiori/minori avranno possibilità più elevate di partecipare alla riproduzione e quindi di trasmettere alle future generazioni il proprio corredo genetico.

  In particolare, la selezione per *torneo* si effettua partizionando la popolazione in gruppi di $K \geq 2$ soluzioni scelte in modo casuale. Per ogni gruppo viene selezionata la soluzione con *fitness* più alta/bassa, a seconda se la funzione obiettivo (*fitness*) è da massimizzare o minimizzare. Nel particolare caso $K = 2$, si ha il *torneo* classico. Nella selezione per *roulette* la popolazione è rappresentata come una ruota di roulette, suddivisa in settori. Ciascun individuo corrisponde a un settore la cui area è proporzionale al valore di *fitness*:

$$p_{\mathrm{i}} = \frac{f_{\mathrm{i}}}{\sum_{j=1}^{N} f_{\mathrm{j}}}$$

Il processo consiste nella simulazione di n lanci di una pallina e nella selezione dell'individuo appartenente al settore scelto casualmente dalla pallina. Sostanzialmente, in questo processo i migliori individui avranno una porzione più grande della ruota (che può essere vista come l'intervallo di probabilità [0, 1]) e verranno quindi selezionati per essere sottoposti agli operatori genetici con maggiore frequenza rispetto agli individui peggiori.

In figura 3.1 si può osservare un esempio in cui gli individui 1 e 3, rispettivamente identificati dalla zona blu e rossa, hanno una porzione nel range di probabilità notevolmente più grande rispetto agli altri individui della popolazione. Nel peggiore dei casi, questi due individui saranno selezionati continuamente per divenire i genitori da sottoporre all'operazione di *crossover*, escludendo difatti tutti gli altri e indirizzando la convergenza verso soluzioni molto simili ad essi. Per evitare una tale situazione, che potrebbe portare a stagnazione (convergenza rapida verso minimi locali), si potrebbe sviluppare un meccanismo di invecchiamento che impone la scelta dello stesso individuo per un massimo di *age* volte durante tutta la fase di selezione;

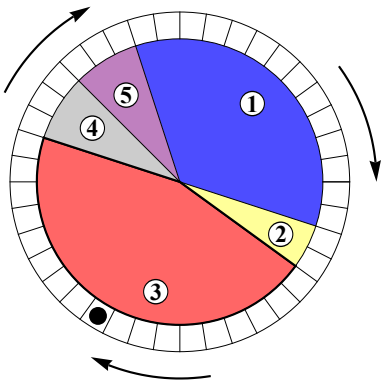


Figura 3.1: Rappresentazione roulette.

- *Operatori Genetici*: combinano gli individui delle diverse soluzioni al fine di esplorare nuove zone dello spazio delle soluzioni. Una volta che un gruppo di soluzioni viene individuato come idoneo alla riproduzione, l'operatore genetico di *crossover* combina l'informazione genetica dei genitori formando così una nuova generazione di soluzioni. Un altro operatore genetico largamente utilizzato è la *mutazione*, essa agisce modificando casualmente uno o più geni dell'individuo su cui opera. L'uso del *crossover* è fondamentale perchè permette di combinare il corredo genetico di almeno due individui per generare figli. In questo modo, le generazioni successive porteranno nel loro corredo genetico caratteristiche sviluppatesi nella generazione precedente. La *mutazione*, d'altro canto, rappresenta la sola operazione per aggiungere variabilità agli individui, è l'unico modo per generare geni (quindi informazioni) che non appartengono al corredo genetico della popolazione. In tal modo si permette la formazione di nuove caratteristiche che potrebbero essere più correlate con il sistema di appartenenza, spingendo difatti l'evoluzione degli individui verso una condizione che più si adatta al sistema.

Lo schema generale di un algoritmo genetico può essere riassunto come segue:

begin

Inizializzazione della popolazione

Valutazione degli elementi della popolazione

repeat

1. *Esegui il crossover degli individui scelti secondo opportuno criterio*
2. *Esegui la mutazione degli individui scelti secondo opportuno criterio*
3. *Esegui la clonazione dei migliori individui*

4. Valuta la fitness della nuova popolazione

5. Seleziona un sottoinsieme della popolazione in base alla fitness

until *(soluzione raggiunta)* ***or*** *(numero massimo generazioni raggiunto)*
end

I modelli principali per l'implementazione degli algortimi genetici paralleli sono:

- *modello globale*: trattare il problema con una sola popolazione, occupandosi di parallelizzare la funzione di valutazione e le fasi di evoluzione;
- *modello ad isole*: in cui si lavora in parallelo con più popolazioni - *multi-popolazione* - od in cui si divide la popolazione in *sottopopolazioni* indipendenti. In tal modo, il processo evolutivo avviene in parallelo fra le varie isole. Si ha la possibilità di mantenere una maggiore diversità genetica all'interno della popolazione complessiva, questo permette una migliore esplorazione dello spazio delle soluzioni;
- *modello diffusivo*: la popolazione è distribuita spazialmente, solitamente su una griglia bidimensionale. L'i-esimo individuo è sostituito nella successiva generazione da un nuovo elemento generato applicando gli operatori genetici ai suoi "vicini". Questo modello permette agli ottimi locali di non diffondersi velocemente all'interno dell'intera popolazione, evitando una convergenza prematura.

Una variante del modello ad isole prevede di far migrare parte della popolazione da un'isola ad un'altra. Con *crossbinding* si indica la tecnica che prevede lo scambio incrociato di informazione fra le varie isole (attraverso

lo scambio di messaggi durante l'elaborazione). Questa strategia permette di aumentare leggermente la variabilità genetica dell'intera popolazione, riducendo il rischio del fenomeno di stagnazione (convergenza rapida verso minimi locali). In letteratura si considerano due tipologie di migrazioni: o si permette ad un individuo di raggiungere qualsiasi isola oppure occorre definire un concetto di "vicinanza" fra le isole e si permette la migrazione solo verso quelle più vicine.
La politica di migrazione può essere implementata in differenti modi:

1. si può infatti imporre che solo una determinata percentuale lasci la popolazione attuale;

2. si può richiedere la migrazione di individui con un valore di *fitness* al di sopra o al di sotto di un valore specifico;

3. si può lasciare il verificarsi o meno dell'azione al caso, implementando politiche di migrazione casuali.

Comunque, al fine di garantire una maggiore velocità di convergenza: la quantità di individui da far migrare, il criterio di scelta di questi e la frequenza di questi scambi sono strettamente dipendenti dal tipo di applicazione che si sta implementando. Un alto numero di individui che migrano riconduce la soluzione a quella dell'algoritmo operante su una sola popolazione, un numero troppo ristretto tende a non rimescolare opportunamente il patrimonio genetico e quindi non permette di superare minimi locali trovati nelle singole isole.

Capitolo 4

Framework FastFlow

FastFlow è un framework per la programmazione parallela strutturata, per sistemi multi/many-core a memoria condivisa, sviluppato dall'università di Pisa e di Torino. FastFlow è implementato come una libreria C++. Questa scelta fornisce al programmatore la possibilità di estendere o modificare (per mezzo delle caratteristiche del paradigma di programmazione orientata ad oggetti) le classi presenti nel framework, sfruttando così gli efficienti meccanismi di comunicazione (una delle caratteristiche principali di FastFlow) messi a disposizione per implementare nuovi skeleton.

Questi meccanismi con bassa latenza sono particolarmente adatti per computazioni a grana fine come quelle che vengono eseguite su stream di pacchetti su reti di calcolatori ad alta banda. Il tempo di comunicazione stimato per inviare un task fra due attività concorrenti FF (ammesso che uno sia disponibile ad inviare e l'altro disponibile a ricevere) è dell'ordine delle decine di nanosecondi [10].

FastFlow è stato progettato per fornire al programmatore dei pattern efficienti che permettono, fra l'altro, di implementare applicazioni stream parallel a grana fine. In particolare tale framework permette di sviluppare

applicazioni dove il parallelismo da utilizzare è espresso mediante costrutti ad alto livello, consentendo al programmatore di non occuparsi di gestire i meccanismi di basso livello per la gestione del parallelismo.

Il framework permette anche di implementare skeleton data parallel attraverso l'uso degli skeleton stream parallel offerti agli alti livelli della libreria (tale tecnica prende il nome di *cross-skeleton template* [9], come già descritto nella sezione 2.3). L'intero framework di programmazione è stato incrementalmente sviluppato secondo un progetto a livelli che si appoggia sul framework di programmazione Pthread/C++. Un primo livello, *simple streaming networks*, fornisce un'efficiente coda *lock-free* single-producer single-consumer (SPSC). Tutto FastFlow si basa su meccanismi di sincronizzazione *lock-free*. Per poter dare una definizione di *lock-free* introduciamo prima il concetto di *non-blocking*.

Un algoritmo è definito *non-blocking* se è garantito che i thread, che sono in competizione per acquisire una risorsa condivisa, non rimanderanno indefinitamente la loro esecuzione a causa della mutua esclusione. In tali algoritmi una struttura dati è sempre accessible da tutti i processi e quelli inattivi (temporaneamente o permanentemente) non possono renderla inaccessibile. È garantito che dei processi attivi saranno in grado di completare un'operazione in un numero finito di passi. Un algoritmo *lock-free* garantisce il progresso di almeno uno dei thread in esecuzione. Questo significa che dei thread possono essere ritardati arbitrariamente, almeno uno progredisce ad ogni passo (contribuendo in tal modo ad evitare *deadlock*). Quindi l'intero sistema fa sempre progressi, sebbene alcuni thread potrebbero progredire più lentamente rispetto ad altri.

Tradizionalmente l'approccio usato nella programmazione multithreading si basa sull'utilizzo di meccanismi lock per sincronizzare gli accessi alle strut-

ture dati condivise, utilizzando per esempio: *mutex*, *semafori*, *monitor*, ecc... Se un thread tenta di acquisire una risorsa in possesso di un altro thread, sarà sospeso finchè il lock sulla risorsa non sarà rilasciato.

Un thread bloccato non può effettuare nessuna operazione finchè non gli è nuovamente permesso. I continui cambi di contesto, dovuti all'alternarsi dei thread, aggiungono *overhead* che influenzano i costi di esecuzione soprattutto nelle computazioni a grana fine. Oltretutto, il non corretto uso dei meccanismi di sincronizzazione potrebbe portare a situazioni di *deadlock* o *starvation*.

Nella libreria POSIX le operazioni di sincronizzazione come la lettura e la scrittura su code condivise SPSC vengono realizzate utilizzando *mutex* e *condition*. Viceversa, FastFlow mette a disposizione dei meccanismi di comunicazione molto più leggeri. Per esempio, le scritture e le letture dalle code FIFO singolo-produttore singolo-consumatore non richiedono l'utilizzo di alcun meccanismo di lock. Nel caso in cui un thread cerchi di leggere da una coda vuota il flusso di controllo è immediatamente ritornato, con l'esito del fallimento dell'operazione di lettura, senza che tale thread venga in alcun modo sospeso.

Il livello *simple streaming networks* può essere utilizzato come primo livello del supporto run-time: il canale di comunicazione è asincrono (essendo *lock-free*) e le operazioni di lettura e scrittura sono non bloccanti (algoritmo di Lamport [28]). Utilizzando FastFlow a questo livello è possibile costruire reti di comunicazione (a basso livello) in modo del tutto simile a quando si programma usando i thread POSIX.

Il secondo livello, l'*arbitrary streaming networks*, fornisce implementazioni *lock-free* per code single-producer multiple-consumer (SPMC), multiple-producer single-consumer (MPSC) e multiple-producer multiple-consumer

(MPMC) sopra la SPSC implementata nel primo livello.

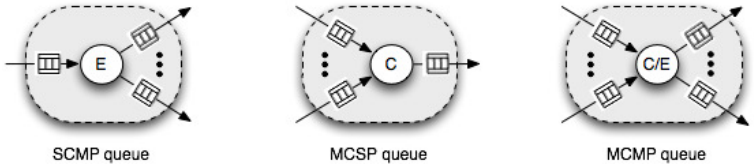


Figura 4.1: Code SPMC, MPSC e MPMC.

Le comunicazioni 1:N, N:1, N:M possono essere realizzate usando, per esempio, meccanismi di lock che incapsulano: le scritture sulle singole code SPSC, nel caso N:1, e le letture sulle singole code SPSC, nel caso 1:N. Così facendo, ovviamente, si paga un *overhead* dovuto alla sincronizzazione diretta per l'accesso alla sezione critica.

In un approccio *lock-free*, risulta necessario l'utilizzo di operazioni atomiche per forzare la corretta serializzazione degli aggiornamenti dei produttori e dei consumatori alla fine della coda. In FastFlow le comunicazioni 1:N, N:1, N:M sono realizzate tramite un thread arbitro senza l'impiego di meccanismi di lock. Infatti, nel caso SPMC sarà un unico thread produttore - chiamato emitter - che deciderà di inviare i dati ad un ben determinato consumatore (precisamente scrivendo i dati nella relativa coda SPSC dedicata per la comunicazione fra il produttore e tale consumatore). Invece, per quanto riguarda le code MPSC, sarà il thread consumatore - chiamato collector - che andrà a prelevare i dati dalla relativa coda SPSC del produttore. Infine, per le code MPMC, il thread arbitro - chiamato collector-emitter (CE) - si comporta sia da produttore che da consumatore.

Il terzo livello, lo *streaming networks patterns*, fornisce i comuni pattern stream parallel: il *pipeline* ed il *farm*. Specializzando questi si possono implementare pattern più complessi come per esempio: *divide and conquer*, *map* e *reduce*.

Per supportare lo sviluppo di applicazioni parallele su multi-core a memoria condivisa il framework fornisce due possibilità di utilizzo:

- *astrazione di skeleton*: per implementare applicazioni completamente modellate secondo i concetti degli skeleton algoritmici. Bisogna scrivere l'applicazione fornendo, sia il codice funzionale (wrappato in un'opportuna sottoclasse *ff_node*) sia la composizione di uno o più skeleton che modellano il pattern voluto. Fatto ciò, occorre far partire la computazione aspettando che termini;

- *acceleratore*: usato per parallelizzare solo delle parti di un'applicazione esistente. In questo caso il programmatore fornisce una composizione di skeleton che gira sui core disponibili ed implementa una versione parallela accelerata della funzione da calcolare. Può essere orchestrata usando codice asincrono, quindi delegando la richiesta e chiedendo i risultati solo quando servono. È uno strumento software che può essere usato per velocizzare porzioni di codice usando i core inutilizzati dall'applicazione principale.

La libreria è realizzata utilizzando l'approccio basato sui template, in cui le attività interne di uno skeleton sono realizzate da un insieme di thread. Più formalmente si definisce *process template* [9] una rete parametrica di attività concorrenti, definita da un grafo $G = (N, A)$ dove gli $n_{\mathrm{i}} \in N$ rappresentano le attività concorrenti ed A è un insieme di coppie $\{ (h, k) \in A \}$ che denotano la comunicazione fra il nodo $n_{\mathrm{h}} \in N$ e $n_{\mathrm{k}} \in N$.

4.1 Performance

In questa sezione, si dà una breve descrizione sulle performance raggiunte utilizzando FastFlow per costruire un programma parallelo sfruttando gli

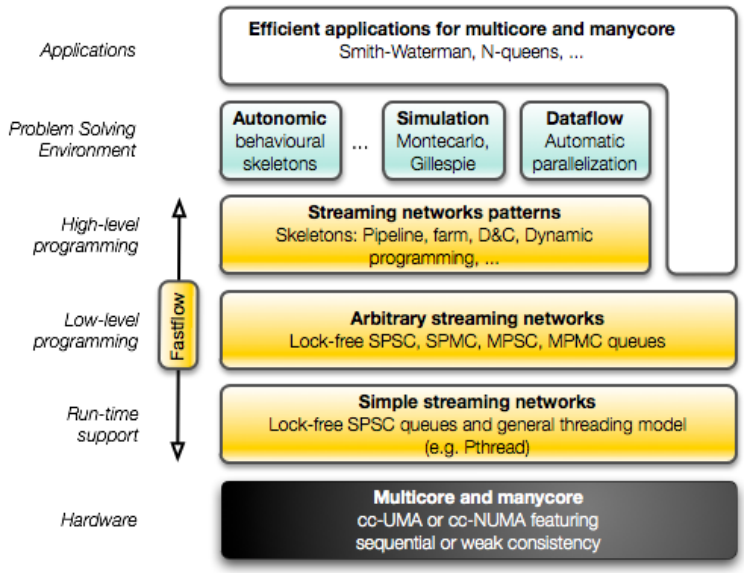


Figura 4.2: Architettura di FastFlow.

skeleton forniti oppure usando FF come acceleratore software. Quando si usa il pattern *farm* ci si aspetta che l'incremento delle prestazioni sia proporzionale al numero dei worker usati (ovvero al grado di parallelismo). Bisogna considerare determinati fattori che possono portare ad avere prestazioni non concordi alle aspettative.

Infatti, si ricorda che porzioni di codice seriale rappresentano un limite in base a quanto stabilito dalla legge di Amdhal (sezione 2.6.1). Oltretutto, in questo pattern il grado di parallelismo non è facilmente identificabile in quanto lavorando su uno stream non si sa quando i dati siano disponibili (come descritto nella 2.2). Un grado di parallelismo troppo basso limiterebbe lo *speedup* - non sfruttando il parallelismo - mentre un grado troppo alto porterebbe ad un decremento dell'*efficienza*. In quest'ultimo caso, i task non si presentano sullo stream di input ad una velocità adatta a non far rimanere inattivi i worker. Un altro problema da considerare, per esempio, si verifica quando il tempo di computazione della funzione assegnata al worker è minore

del tempo di invio e di recupero del task da parte dell'emitter.

Assumiamo che queste problematiche non siano presenti nell'applicazione sviluppata. Allora, sia T_{seq} il tempo per computare m task sequenzialmente, ci si aspetta che la stessa computazione usando n_w worker si avvicinerà al tempo teorico $\frac{T_{\text{seq}}}{n_{\text{w}}}$.

Nel caso in cui l'applicazione usi il pattern *pipeline*, il tempo di servizio T_S aspettato è $T_{\text{s}} = max\{T_{\text{s}_1}, T_{\text{s}_2}, ..., T_{\text{s}_k}\}$. La computazione di m costa quindi $m \cdot T_{\text{s}}$. Lo *speedup* diventa:

$$\frac{m \cdot \sum_{i=1}^{k} T_{\text{s}_i}}{m \cdot max\{T_{\text{s}_1}, T_{\text{s}_2}, ..., T_{\text{s}_k}\}} = \frac{\sum_{i=1}^{k} T_{\text{s}_i}}{max\{T_{\text{s}_1}, T_{\text{s}_2}, ..., T_{\text{s}_k}\}}$$

Nell'eventualità in cui la computazione effettuata dagli stage usati sia bilanciata, stesso ordine di costo per calcolare le diverse funzioni che corrispondono agli stadi del *pipeline*, lo *speedup* si può approssimare a k. Infatti se T_{s_i} è il massimo fra tutti i tempi degli stage, allora la frazione precedente è minore od uguale a:

$$\frac{k \cdot T_{\text{s}_i}}{T_{\text{s}_i}}$$

rapporto che semplificato è uguale a k.

4.2 Parallel design patterns

I parallel design pattern sono stati introdotti nei primi anni '00 con l'obiettivo di fornire soluzioni a problemi paralleli ricorrenti. Rappresentano un modo ingegnerizzato di considerare il problema connesso allo sfruttamento del parallelismo. Un design pattern è una rappresentazione di un problema comune con una soluzione efficiente e collaudata per quel problema. Compaiono usualmente nella programmazione ad oggetti anche se l'idea è generale e può essere applicata a differenti modelli di programmazione. Un pattern

in questo ambito non è un vero e proprio costrutto ma consiste in un "buon modo" (un consiglio) per strutturare la propria applicazione, il programmatore che segue tale "consiglio" otterrà un'applicazione più facile da riusare e da mantenere. Un parallel design pattern è descritto da un preciso insieme di elementi, il cui significato è qui spiegato:

- *problem*: il problema da risolvere;
- *context*: il contesto dove il pattern può essere opportunamente usato;
- *forces*: le differenti caratteristiche che influenzano il parallel design pattern;
- *solution*: una descrizione di una o più soluzioni possibili al problema risolto dal pattern.

Uno degli aspetti più interessanti relativi ai parallel design pattern è che questi partizionano lo spazio di progettazione in quattro parti:

- *finding concurrency*: include i parallel design pattern che modellano differenti generi di attività parallele o concorrenti che possono essere usate in un'applicazione parallela;
- *algorithm structure*: contiene i parallel design pattern che modellano i ricorrenti pattern degli algoritmi paralleli;
- *supporting structure*: include i parallel design pattern che possono essere usati per implementare i pattern del livello precedente;
- *implementation mechanisms*: contiene i pattern che modellano i meccanismi di base usati per implementare un'applicazione parallela.

L'importanza di tale struttura di progettazione risiede sulla separazione dei problemi: il programmatore dell'applicazione lavorerà nei primi due livelli e quindi nel dominio dell'applicazione ignorando difatti l'architettura utilizzata. Viceversa, il programmatore di sistema lavorerà negli ultimi due livelli.

4.3 Pattern pool evolution

Tale pattern è stato proposto ed introdotto in FastFlow, come spiegato in [4], perché cattura differenti contesti di utilizzo. Quelli legati agli algoritmi evolutivi e quelli legati a computazioni simboliche. Come visto nella sezione 4.2, lo spazio di progettazione può essere diviso in quattro parti: *finding concurrency* che modella i differenti tipi di attività parallele/concorrenti che possono essere usate nell'applicazione parallela, *algorithm structure* che modella i differenti algoritmi ricorrenti, *supporting structure* che include i pattern usati per implementare quelli del livello precedente e per finire l'*implementation mechanisms* che modella i meccanismi base usati per implementare un'applicazione parallela.

Il *pool evolution*, visto che modella un ben preciso algoritmo parallelo, appartiene al secondo spazio di progettazione elencato. Questo pattern non solo è utile a rappresentare algoritmi genetici ma anche tutte quelle applicazioni che, per fornire un risultato, iterano l'applicazione di determinate funzioni, nella fase di evoluzione, (che siano *crossover*, *mutazioni* o che dir si voglia) fino al verificarsi di una qualche condizione di terminazione. L'idea è quindi illustrata dal seguente pseudocodice:

begin
  while not(*termination(Pop)* ***and*** *iterations*$\geq$ *0)* ***do***
    tmpPop = selection(Pop);
    newPop = evolution(tmpPop);
    Pop = Pop $\bigcup$ *filter(newPop);*
  end while
end

È quindi un processo iterativo in cui l'i-esima iterazione lavora su ciò che è computato dall'iterazione *(i-1)*-esima. Evidenziamo le caratteristiche peculiari di questo processo:

- *Pop*: è l'input che viene sottoposto alla fase di selezione ed i cui risultati sono sottoposti ad evoluzione al fine di evolvere verso una popolazione "migliore" *newPop*, che meglio si adatta alle caratteristiche che l'ambiente gli richiede. Tale popolazione subisce un filtraggio e l'output si integra o no (a seconda della strategia seguita) con parte della popolazione iniziale per andare a formare l'input dell'iterazione successiva;

- *selection*: è una funzione che identifica gli individui che devono sottoporsi alla fase di evoluzione. Può essere implementata utilizzando, per esempio, la funzione identità oppure una qualsiasi altra funzione, che predilige un individuo ad un altro in base al valore di uno o più attributi, etc...;

- *Evolution*: racchiude tutti gli operatori necessari a modificare e quindi garantire l'evoluzione della popolazione. È *embarassingly paral-*

lel in quanto un solo individuo od un gruppo di questi può evolvere indipendentemente dagli altri selezionati;

- *filter*: è anch'essa una selezione, crea la nuova popolazione per l'iterazione successiva filtrando in base a qualche parametro (valore di *fitness* per esempio). Si possono utilizzare diverse tecniche. Per esempio: all'insieme sottoposto ad evoluzione si possono integrare gli individui dell'iterazione precedente (i genitori), oppure si possono creare nuovi individui, o si può decidere di ridurre la dimensione della popolazione considerando solo l'output dell'evoluzione, etc. . . ;

- *termination*: verifica che le condizioni di terminazione siano soddisfatte, il tal caso si arresta il processo di evoluzione. La terminazione è generalmente implementata in modo sequenziale.

In questo pattern è possibile sviluppare l'applicazione personalizzando i comportamenti delle varie fasi, infatti: è possibile effettuare la fase di *selection* e/o quella di *filter* sequenzialmente parallelizzando la sola *evolution* oppure una od entrambe queste fasi possono essere parallelizzate, utilizzando opportunamente l'implementazione del *parallel for* (sezione 4.4) presente in FastFlow, ottenendo una sequenza di *farm*. La *selection* e la *filter* possono essere viste come delle vere e proprie *MapReduce*. Nella *selection*, per esempio, in fase di *map* si valutano gli elementi ed in fase di *reduce* si filtra in base a ciò che si desidera. *Map* e *MapReduce* sono implementate utilizzando il pattern *ParallelForReduce* (vedi sezione 4.4) che permette la parallelizzazione efficiente di *parallel for* e *parallel for* con riduzione, implementato utilizzando un *farm* con feedback. Ovvero un *farm* il cui emitter E schedula i task ai worker che eseguono il loro calcolo e spediscono l'output nuovamente all'emitter E.

Nella *filter*, si possono utilizzare diverse tecniche in base al problema affrontato. In caso di stagnazione è possibile, per esempio, rimpiazzare alcuni individui generandone di nuovi o clonando i migliori della popolazione attuale. Altrimenti si possono semplicemente integrare gli individui dell'iterazione i-esima con quelli ottenuti applicando gli operatori di evoluzione a questi, unendo così figli e genitori per l'iterazione successiva.

Un altro aspetto da considerare riguarda la dimensione della popolazione. Alcuni algoritmi genetici lavorano con popolazione di dimensione costante mentre altri tendono a modificare la dimensione durante le iterazioni. La gestione di tale situazione è lasciata al programmatore che usa il pattern, è infatti compito suo gestire eventuali allocazioni o deallocazioni di memoria.

Il pattern *pool evolution* è fornito con due costruttori (mostrati in figura 4.3): uno che supporta l'esecuzione stand-alone dello stesso, processando solo la popolazione di input specificata come parametro; l'altro che supporta l'esecuzione su popolazioni che appaiono sullo stream di input. Inoltre, ad un'istanziazione della classe *poolEvolution* è possibile associare un ambiente esterno di tipo *env_t* contenente tutte le informazioni globali necessarie alle varie fasi descritte dell'intero processo ed interamente personalizzabile dall'utente a seconda delle richieste dell'applicazione. Il tipo *env_t* è passato come parametro del template e l'oggetto ambiente è passato al costruttore del pool.

```
/* constructor: to be used in non-streaming applications */
poolEvolution(size_t maxp,     /* maximum parallelism degree in all phases */
              std::vector<T> &pop,          /*  the initial population    */
              selection_t sel,              /*  the selection function    */
              evolution_t evol,             /*  the evolution function    */
              filtering_t fil,              /*  the filter function       */
              termination_t term,           /*  the termination function  */
              const env_t &E = env_t());    /*  user's environment        */

/* constructor: to be used in streaming applications */
poolEvolution(size_t maxp,     /* maximum parallelism degree in all phases */
              selection_t sel,              /*  the selection function    */
              evolution_t evol,             /*  the evolution function    */
              filtering_t fil,              /*  the filter function       */
              termination_t term,           /*  the termination function  */
              const env_t &E = env_t());    /*  user's environment        */
```

Figura 4.3: Costruttori del pattern *pool evolution* di FastFlow.

4.4 Pattern parallel for

Lo skeleton che implementa il *PARFOR* [7] è un *farm* con feedback, in cui i canali di comunicazione sono come, al solito, implementati utilizzando una coda SPSC *lock-free*. Il thread *forSched* implementa la politica di schedulazione dei task. Un task è semplicemente una coppia di long int $< start, stop >$ che definisce una porzione del range delle iterazioni del loop. I thread dei worker ricevono i task su richiesta, eccetto per i primi. Lo schedulatore spedisce un nuovo task ai worker solo se il precedente task assegnato al worker è tornato indietro attraverso il canale di feedback. Tale politica assicura di lavorare in modo ben bilanciato fra i thread worker senza la necessità di implementare tecniche più complesse e costose.

La politica di scheduling attualmente implementata lavora come segue: le iterazioni sono divise in porzioni di uguale dimensione, logicamente porzioni

contigue sono assegnati allo stesso thread provando ad eguagliare il più possibile il numero di porzioni per ogni thread nel pool. Il *forSched* - durante la computazione del ciclo for - spedisce parti ai worker su richiesta. Se *forSched* non ha nessuna porzione di iterazione da fornire al thread richiedente, prova a sottrarne una parte ad un altro thread (tale tecnica è nota come *job stealing*). Per implementare questa politica, una tabella di task - contenente sia gli indici $< minimo, massimo >$ (dell'iterazione) da eseguire che il numero di task attualmente disponibili - è mantenuta come struttura dati privata nel *forSched*. L'obiettivo principale di tale politica di schedulazione è quello di provare ad ottenere un buon trade-off tra il bilanciamento di carico e l'assegnazione di una parte dell'iterazione allo stesso thread. Quest'ultima, è un fattore importante per le performance nei sistemi con cache condivisa perchè aumenta la probabilità di trovare i dati cercati in uno dei livelli di cache presenti. È implementata usando una tabella contenente una *entry* per ciascun thread worker. Così facendo, si riduce l'*overhead* di comunicazione dovuto all'indirizzamento dei dati non locali su piattaforme *NUMA* (*non-uniform memory access*). Nell'implementazione di *PARFOR* non è presente nessuna struttura dati condivisa lungo i thread, si evitano così *overhead* dovuti ai meccanismi di lock.

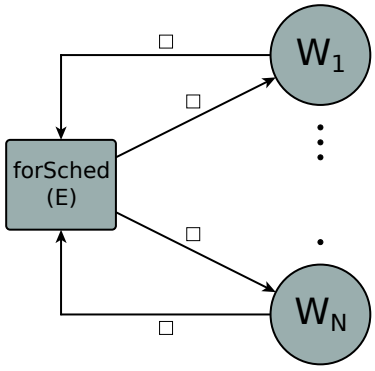


Figura 4.4: Pattern *parallel for* di FastFlow.

Capitolo 5

Modello ad isole con sottopopolazioni

Nella sezione 4.3 si è discusso del pattern *pool evolution* evidenziandone le caratteristiche ed i diversi contesti di utilizzo. L'articolo [4], accenna all'idea di realizzare un'implementazione per trattare la possibilità di lavorare in parallelo con sottopopolazioni. Questo capitolo provvede a fornire tale estensione come un pattern di FastFlow, descrivendo i dettagli implementativi e spiegando i cambiamenti apportati alla struttura del pattern esistente che opera su singola popolazione. Il risultato ottenuto permette quindi di implementare quello che nel paradigma genetico viene chiamato modello ad isole.

Spieghiamo brevemente quali sono i passi fondamentali per implementare tale modello. La struttura dati fornita in input al costruttore, contenente gli individui, è partizionata nel numero di isole richieste che coincide con il grado di parallelismo nE che si vuole utilizzare. Su ciascuna partizione è applicato il processo di evoluzione, descritto nella sezione 4.3, finchè si raggiunge l'ottimo cercato, se conosciuto, oppure finchè non sono trascorse

un certo numero di iterazioni (parametro passato al costruttore della classe) attivando il processo di scambio degli individui fra le isole. Questa operazione è permessa da un metodo definito dal programmatore, secondo la politica da lui ritenuta più adatta, che è passato al costruttore della nuova classe.

Entrando nel dettaglio, nel pattern *pool evolution* presente in FastFlow, la fase di selezione prevede la creazione di nuovi individui da sottoporre all'evoluzione. Nel caso in cui si lavori con computazioni a grana fine, questa fase può trasformarsi in un vero e proprio "collo di bottiglia". Per questa ragione, si è scelto di ridurre il più possibile il numero di allocazioni/deallocazioni di memoria; la soluzione fornita opera con puntatori ad oggetti. Al programmatore è richiesto di definire, all'interno della sua classe rappresentante l'individuo, un metodo *Clone()* che si occupa di effettuare una clonazione in profondità dell'oggetto stesso. All'interno del metodo *svc()* (presente in tutte le classi che estendono un *ff_node* e che definisce il codice funzionale dell'applicazione) del nuovo pattern si usa l'operazione di clonazione dell'intera popolazione per creare il vettore *buffer* passato agli operatori per l'evoluzione.

Dopo aver controllato che il metodo per confrontare gli ambienti sia stato correttamente definito, si passa per il metodo *termination()*. Se le condizioni non sono soddisfatte, si procede ad utilizzare il pattern *ParallelFor-Reduce* presente in FastFlow (descritto nella sezione 4.4) per delegare la computazione delle partizioni di *input* alle nE isole .

In ciascuna di queste si seguono i passi descritti nella sezione 4.3, ovvero (tralasciando le variabili introdotte per trattare correttamente gli accessi alla struttura globale) si applicano i metodi *selection()*, *evolution()* e *filter()* passati al costruttore. Una prima differenza da sottolineare tra la nuova versione e la vecchia riguarda la fase di evoluzione; sebbene questa libreria definisca

una classe che eredita dal pattern *pool evolution* presente nel framework, è stato fornito un nuovo metodo di evoluzione che opera su una partizione piuttosto che su un singolo individuo. Gli operatori genetici, come per esempio il *crossover*, solitamente necessitano di almeno due individui per poter operare correttamente. Quindi, il nuovo metodo di evoluzione riceve in input:

- un'istanza della classe *ParallelForReduce*;
- un vettore di individui;
- l'eventuale ambiente;
- un intero indicante l'isola di appartenenza.

Il tipo di evoluzione dipende dal programmatore che può decidere di lavorare sequenzialmente sul vettore ricevuto in input od usare opportunamente l'istanza del *ParallelForReduce* per aggiungere un ulteriore grado di parallelismo all'interno dell'isola.

Per realizzare il modello ad isole spiegato nella sezione 3.2 sono stati aggiunti al costruttore (mostrato in figura 5.1) del *pool evolution* due parametri:

- *delta*: è un intero che indica la frequenza di scambio (tale valore deve essere maggiore di 0 viceversa l'errore *"delta must be greater than 0."* sarà segnalato). Come già descritto, al compiersi di *delta* iterazioni (se il numero delle sottopopolazioni è maggiore di 1) si rimescolano le informazioni fra le varie isole;
- *mix()*: è il metodo usato per rimescolare (far migrare) gli individui nel modo ritenuto più consono dal programmatore. Il metodo riceve in input due vettori, quello contenente tutti gli individui iniziali e quello ottenuto per clonazione. Sia l'*offset* uguale al numero totale degli

individui diviso il numero delle sottopopolazioni, l'i-esima isola contiene gli individui compresi nell'intervallo $[\mathrm{i} \cdot \mathrm{offset};\ (\mathrm{i}+1) \cdot \mathrm{offset} - 1]$ del vettore. Il programmatore può eseguire l'operazione di rimescolamento dell'informazione tra le isole i e k, selezionando e scambiando un individuo appartenente all'intervallo $[\mathrm{i} \cdot \mathrm{offset};\ (\mathrm{i}+1) \cdot \mathrm{offset} - 1]$ con uno appartenente all'intervallo $[\mathrm{k} \cdot \mathrm{offset};\ (\mathrm{k}+1) \cdot \mathrm{offset} - 1]$. Se si vuole mantenere la stessa informazione tra le isole del vettore iniziale e quelle del vettore ottenuto per clonazione occorre applicare l'operazione ad entrambe le strutture dati.

```
/* constructor: to be used in non-streaming applications */
poolEvolutionE(size_t maxp,    /* maximum parallelism degree in all phases */
               std::vector<T> & pop,          /*  the initial population    */
               pool::selection_t sel,         /*  the selection function    */
               evolution_tp evolP,            /*  the evolution function    */
               pool::filtering_t fil,         /*  the filter function       */
               pool::termination_t term,      /*  the termination function  */
               mix_t m,                       /*  the mixing function       */
               size_t d,                      /*  the mixing frequency      */
               const env_t &E = env_t());     /*  user's environment        */


/* constructor: to be used in streaming applications */
poolEvolutionE(size_t maxp,    /* maximum parallelism degree in all phases */
               pool::selection_t sel,         /*  the selection function    */
               evolution_tp evolP,            /*  the evolution function    */
               pool::filtering_t fil,         /*  the filter function       */
               pool::termination_t term,      /*  the termination function  */
               mix_t m,                       /*  the mixing function       */
               size_t d,                      /*  the mixing frequency      */
               const env_t &E = env_t());     /*  user's environment        */
```

Figura 5.1: Costruttore della classe dell'estensione del pattern *pool evolution*.

Nella nuova classe si definiscono i seguenti attributi e metodi:

- *loopevol_tmp*: un vettore di istanze del pattern *ParallelForReduce* (di dimensione pari al numero delle isole), ciascuno operante su una partizione. Permette di definire un ulteriore grado di parallelismo sulle isole per le fasi descritte in sezione 4.3;

- *nwE*: è un intero che indica il grado di parallelismo da usare all'interno di ogni isola (partizione);

- *env_tmp*: è un vettore di ambienti (di dimensione pari al numero delle isole), in posizione i-esima è presente l'ambiente locale dell'isola i;

- *compareE()*: è un metodo che deve essere fornito per permettere un confronto fra gli ambienti delle varie isole (*NULL* di default. Qualora il numero di *processing element* utilizzati sia maggiore di 1 è richiesta obbligatoriamente la sua definizione, viceversa l'errore *"compareE() must be defined."* sarà segnalato). La *compareE()* è fondamentale per la gestione dell'ambiente globale; quando si decide di rimescolare gli individui, l'ambiente globale va opportunamente aggiornato con le informazioni del miglior ambiente locale. Per determinare il miglior ambiente locale va, dunque, fornito un opportuno operatore di confronto fra gli ambienti: il metodo *CompareE()*;

- *setCompareE(e)*: è un metodo usato per settare *compareE()* con il paramentro e passato in input;

- *enablePR(nwE)*: metodo a cui si passa in input un intero indicante il grado di parallelismo da usare per inizializzare le istanze del pattern *ParallelForReduce* nell'eventuale *MapReduce* (visto nella sezione 4.3) effettuata su ciascuna isola;

- *setIte(i)*: prende in input un intero, usata per settare il numero massimo di iterazioni dell'algoritmo (se previste);
- *getIte()*: restituisce il numero di iterazioni massime previste (valore di default INT_MAX).

Per il corretto aggiornamento dell'ambiente, il programmatore deve fornire all'interno della classe rappresentante l'ambiente *env* l'overload dell'operatore di assegnamento. Questo deve essere definito in modo da poter effettuare una copia in profondità dell'oggetto.

Spostiamo l'attenzione sull'operazione di migrazione. Questa, da un punto di vista teorico, può incidere su due aspetti: riattivare il processo di evoluzione nel caso in cui ci si trovi in fase di stagnazione (convergenza su un minimo locale ed impossibilità di allontanarsi da questo) e cercare di aumentare la velocità di convergenza verso l'ottimo globale. Entrambi gli aspetti dipendono dall'aumento di variabilità nelle sottopopolazioni indotto dal rimescolamento degli individui. Questa operazione risulta vantaggiosa (fondamentale in alcuni casi) per tutti quei problemi particolarmente sensibili alle variazioni delle soluzioni. In sintesi: un'isola che riceve individui - i cui caratteri differiscono dalla media degli abitanti e che meglio si adattano all'ambiente circostante - si modificherà di conseguenza tendendo verso questi, migliorando così le sue caratteristiche complessive.

Va posta molta attenzione sul grado di parallelismo utilizzato e quindi del numero di isole da considerare. Se, infatti, da un lato è vero che ad un grado di parallelismo maggiore corrisponde una maggiore velocità di completamento dell'algoritmo, dall'altro lato è anche vero che un grado di parallelismo troppo elevato crea un grande numero di isole contenenti ciascuna pochi individui. In questa situazione, non si ha la necessaria variabilità locale per garantire l'evoluzione (per esempio: un'operatore come il *crosso-*

ver si troverebbe a far accoppiare sempre gli stessi individui incrociando gli stessi patrimoni genetici) rallentando difatti la velocità di convergenza. Una possibile soluzione consiste nell'aumentare la frequenza di scambio; questa, però, porta ad avere una situazione analoga a quella della versione che opera su una singola popolazione.

Spostiamo l'attenzione alla fase di filtraggio. Al metodo *filter*, si passa un riferimento sia alla popolazione di input che alla struttura contenente gli elementi selezionati dalla struttura creata dalla clonazione. Il programmatore deve aver chiaro che le modifiche agli individui di input (qualsiasi tipo di aggiornamento: incremento o decremento attributi, copie degli individui migliori, etc...) vanno opportunamente gestite in questo metodo e quindi sono sotto la sua responsabilità. Viceversa, essendo l'evoluzione svoltasi sull'area di memoria temporanea *buffer*, le modifiche non saranno trasferite sull'area di *input*.

La libreria è implementata preoccupandosi di gestire correttamente gli aggiornamenti sui vettori *input* e *buffer* ogni qual volta il controllo passi dall'isole alla struttura centrale. Questo perchè non è stato posto nessun vincolo né sulla dimensione della popolazione né sull'allocazione/deallocazione di oggetti in memoria.

La figura 5.2 mostra la versione del pattern presente in FastFlow in cui le fasi di selezione e filtraggio (rispettivamente S_1,..., S_N e F_1,..., F_N, sono qui rappresentate utilizzando il pattern *MapReduce* ma possono semplificarsi ad un singolo nodo). La figura 5.3 invece mostra, in verticale per chiarezza, la nuova versione ottenuta dall'estensione descritta in questo capitolo; la struttura pertanto cambia passando da una sequenza di *farm* (o singola *farm*) ad un *farm* in cui ogni worker è un pattern *pool evolution* (sequenza di *farm* o singola *farm*).

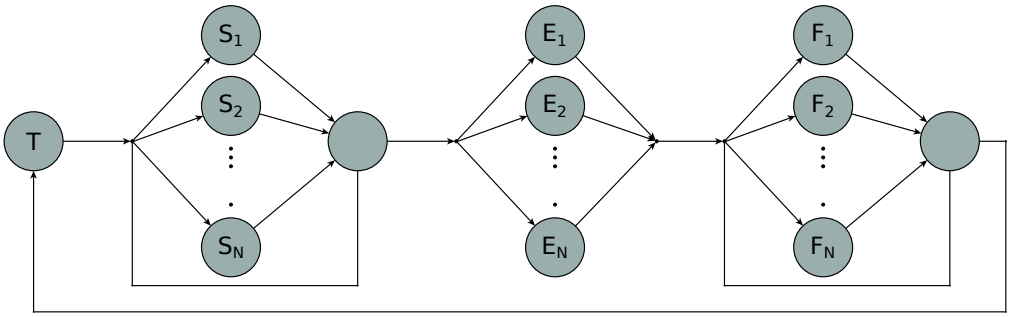


Figura 5.2: Pattern *pool evolution* di FastFlow. Ciascun cerchio rappresenta un thread, le frecce rappresentano i canali di comunicazione che sono implementati per mezzo dei buffer lock-free di FastFlow. T indica il thread che si occupa di valutare le condizioni di terminazione, con S_K si indicano i thread che si occupano della fase di *selezione*, con E_K quelli della fase di *evoluzione* e con F_K quelli della fase di *filter*.

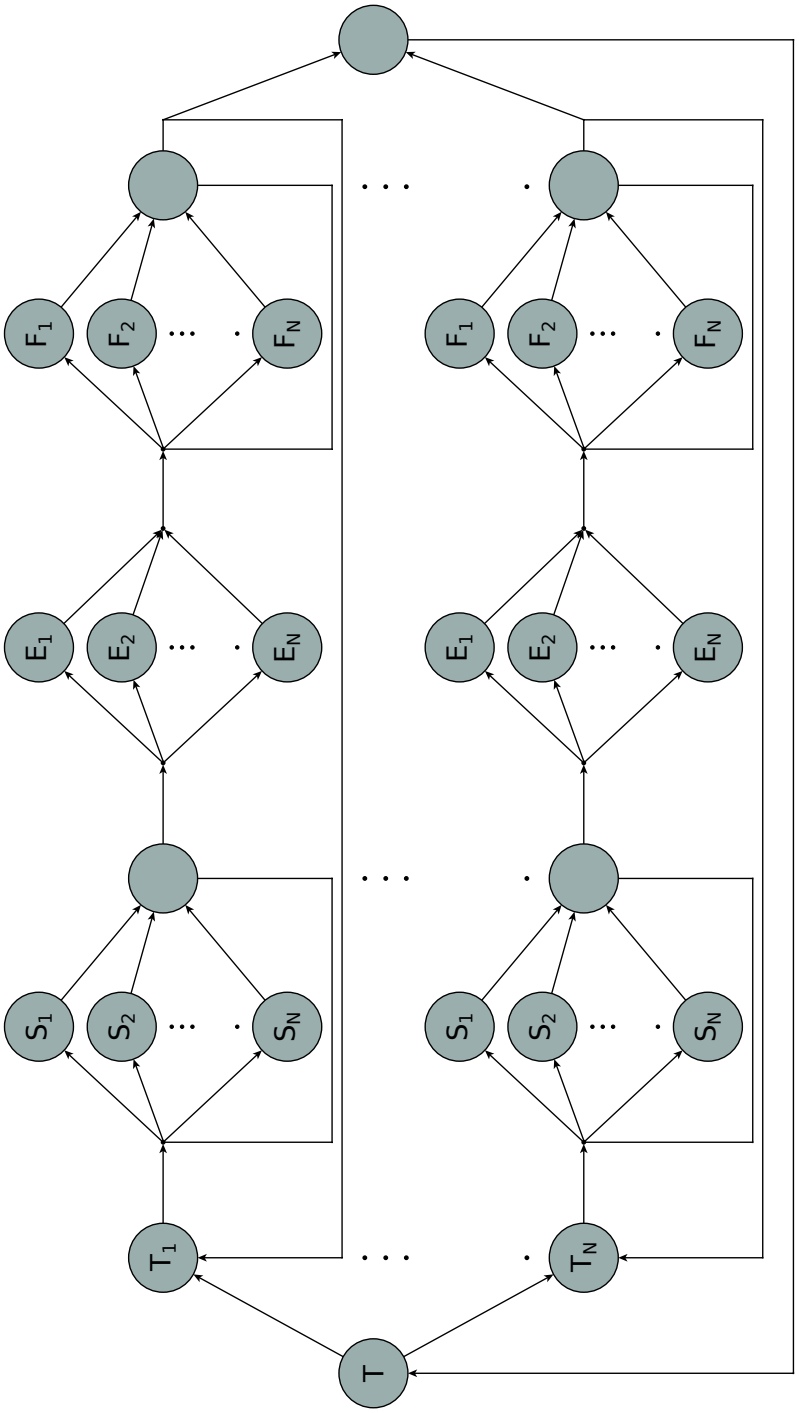


Figura 5.3: Estensione del pattern *pool evolution* di FastFlow. Ciascun cerchio rappresenta un thread, le frecce rappresentano i canali di comunicazione che sono implementati per mezzo dei buffer lock-free di FastFlow. T indica il thread che si occupa di valutare le condizioni di terminazione globali, mentre T_K verifica le condizioni di terminazione locali sulla K-esima isola (ottimo globale raggiunto $\vee$ iterazioni concluse $\vee$ occore rimescolare gli individui). Con S_K si indicano i thread che si occupano della fase di *selezione*, con E_K quelli della fase di *evoluzione* e con F_K quelli della fase di *filter*.

Capitolo 6

Applicazioni

In questo capitolo si mostrano una serie di problemi affrontabili secondo la logica del nuovo pattern descritto nella sezione 5. Le applicazioni implementate sono:

- *Hello world genetic;*
- *Minimum of a function;*
- *Compute function given n known points;*
- *Sudoku;*
- *K-means clustering;*
- *Knapsack.*

Per ognuna di queste, nelle sezioni che seguono, si specificano i dettagli implementativi e si riporta una breve descrizione: della codifica usata per rappresentare gli individui, degli operatori genetici di *crossover* e *mutazione* implementati e delle fasi del pattern già descritto nella sezione 4.3 e nel capitolo 5.

Le soluzioni fornite sono state sviluppate secondo il paradigma genetico, quindi ogni problema si è trasformato in un problema di ottimizzazione da minimizzare o massimizzare, utilizzando una popolazione di dimensione costante. A tal fine, la fase di selezione del pattern applica la funzione identità $f(x) = x$ e quella di filtraggio fa sopravvivere i migliori individui confrontando genitori e figli. Le condizioni di terminazione e gli operatori genetici cambiano in base all'applicazione in esame. Come già detto nel capitolo 5, tutte le classi usate per rappresentare gli individui contengono al loro interno la definizione del metodo *Clone()* per clonare la popolazione iniziale.

6.1 Hello world genetic

Il problema consiste nel ricostruire in maniera genetica una stringa data. Gli esperimenti sono condotti sulla stringa ottenuta prendendo i primi 1153 caratteri da *Il cinque maggio* (di A. Manzoni).

vXdW tl myk genio r tacqul $\rightarrow$ *vide il mio genio e tacque*

Figura 6.1: Esempio di ricostruzione della stringa.

Segue una descrizione degli aspetti cruciali.

Popolazione

Rappresenta gli individui soggetti agli operatori genetici, ovvero i candidati a rappresentare la soluzione al problema. Ogni individuo è l'istanza di una classe che ha un array di caratteri di lunghezza pari a quella della stringa da ricostrurire, 1153, ed un campo intero che rappresenta la bontà della soluzione.

Crossover

È un'operatore genetico che lavora su più di un individuo per volta. In questa implementazione ne riceve due in input e si occupa di crearne altri due come combinazione dei genitori. Il *crossover* è fondamentale per la convergenza, gli individui creati continuano ad avere parte dell'informazione (codice genetico) dei genitori. Solitamente un *crossover* su stringhe crea due figli assegnandogli le parti alternate dei genitori sezionati in n punti (punti di taglio) generati in modo casuale. In questa soluzione l'operatore è implementato con 1 solo punto di taglio (generato in modo random tra 0 e la dimensione della stringa da ricostruire). È applicato con probabilità $p_c = 0,9$.

Mutazione

È un'operatore genetico utile ad aggiungere variabilità alla popolazione. Preso un individuo in input, si genera una posizione random nell'array (tra 0 e la dimensione della stringa da ricostruire), il carattere presente in tale posizione è sostituito con un altro generato anch'esso in modo casuale. È applicato con probabilità $p_{\mathrm{m}} = 0,9$.

Selezione

Ad ogni iterazione va selezionata la porzione di popolazione soggetta agli operatori genetici. Oltre alla funzione identità, già descritta all'inizio di questo capitolo, una selezione è realizzata applicando gli operatori genetici (agli elementi della popolazione) con una data probabilità. Inoltre, per evitare di far generare figli sempre alle stesse coppie, ogni sottopopolazione è soggetta ad una permutazione degli individui.

Fitness

La funzione di valutazione usata per attribuire un valore indicante la bontà dell'individuo. In questo caso è implementata calcolando la distanza di *Hamming* fra le due stringhe, ovvero il numero di caratteri differenti tra l'individuo generico e la stringa ottima (quella da ricostruire). Valori minori indicano una distanza minore dalla stringa ottima, quindi individui migliori. In tal caso si rientra nella categoria dei problemi di minimizzazione in cui si conosce il minimo da ottenere. In questo caso la soluzione ottima ha *fitness* pari a 0.

Terminazione

Le condizioni di terminazione dell'algoritmo sono due. Il processo di evoluzione genera un individuo con valore di *fitness* uguale a 0 (ovvero si è ricostruita

la stringa senza errori), oppure è terminato il numero delle massime iterazioni permesse.

6.2 Minimum of a function

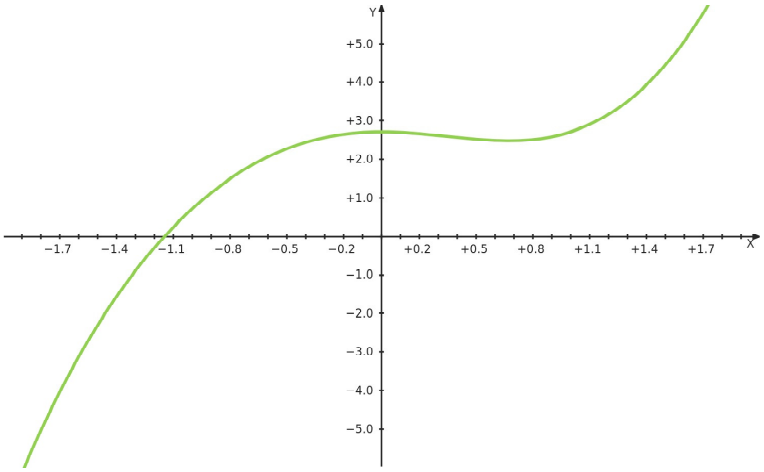


Figura 6.2: $f(x) = e^{\cos(x)} + x^3$.

Il problema consiste nel calcolare il minimo di una funzione data in input, in uno specifico intervallo di ricerca [a, b].

Popolazione

Rappresenta gli individui soggetti agli operatori genetici, ovvero i candidati alla soluzione del problema. Gli individui sono rappresentati come una coppia *(x, f(x))*. Per la fase di generazione si è scelto di partizionare equamente l'intervallo di ricerca nel numero di individui disponibili.

Crossover

Nell'implementazione fornita, questo operatore riceve in input due individui e ne genera altrettanti. Per prima cosa si ordinano i genitori per ascissa, sia x_1 il punto di ascissa minore e x_2 quello di ascissa maggiore. Il primo figlio creato è il punto medio x_m fra x_1 e x_2. Sia d la distanza tra x_m ed uno dei suoi genitori. Allora, il secondo figlio è creato assegnandogli la posizione

x2+d (si pone attenzione a non generare un punto fuori dall'intervallo fornito in input). Si calcola la *fitness* dei nuovi punti creati, l'operatore restituisce due figli facendo sopravvivere solo i migliori fra i quattro punti *x1*, *x2*, *x1+d*, *x2+d*. È applicato con probabilità $p_c = 0,9$.

Mutazione

L'operatore implementato riceve in input un punto e decide (con probabilità pari a $0,5$) se muoverlo verso l'estremo sinistro oppure verso l'estremo destro dell'intervallo di ricerca specificato in input. Dopodiché, una nuova ascissa è generata in modo casuale considerando la distanza fra la posizione attuale del punto e l'estremo scelto (ponendo attenzione a non superare gli estremi dell'intervallo considerato). Quindi il punto è spostato nella nuova posizione. Si applica con probabilità $p_m = 0,3$.

Selezione

Si selezionano gli individui da sottoporre agli operatori genetici combinando: la funzione identità nella *selection()* del pattern, la permutazione degli individui nelle sottopopolazioni e l'uso delle probabilità nell'applicazione degli operatori nella fase evolutiva.

Fitness

La funzione utilizzata calcola il valore dell'ordinata *f(x)* del punto x ricevuto in input. È un problema di minimizzazione in cui non si conosce l'ottimo da trovare.

Terminazione

L'algoritmo termina la sua esecuzione alla conclusione del numero massimo

delle iterazioni permesse.

6.3 Compute function given n known points

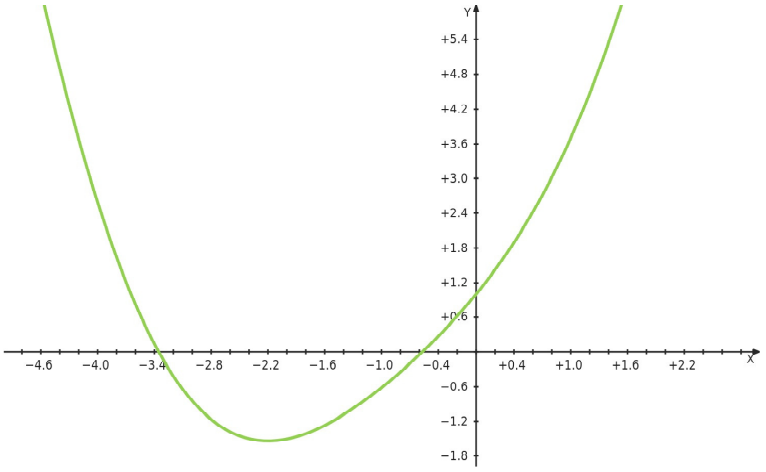


Figura 6.3: $f(x) = \cos(x) + \sin(-x) + x^2 + 3 \cdot x$.

Il problema consiste nel ricostruire una funzione a partire dal valore *f(x_i)* noto per un certo insieme di punti x_i. Per definire gli individui bisogna prima definire il linguaggio delle funzioni ammissibili e quindi occorre definire la grammatica G che permette di generare gli elementi appartenenti a tale linguaggio. La grammatica usata produce gli elementi che seguono:

- costanti k;
- variabile x;
- funzioni unarie: *coseno*, *seno* e *valore assoluto*;
- funzioni binarie: *massimo*, *minimo*, *somma*, *sottrazione*, *moltiplicazione*, *divisione* ed *elevamento a potenza*.

Nell'applicazione degli operatori e nel calcolo della *fitness* si è scelto di rappresentare gli individui utilizzando la notazione polacca. La notazione polacca permette di semplificare l'intera gestione. Infatti, non occorre includere

le parentesi nel linguaggio per gestire la priorità degli operatori. Inoltre, si semplifica sia la valutazione di un'espressione appartenente al linguaggio generato dalla grammatica G sia l'implementazione degli operatori genetici, *crossover* e *mutazione*, perchè permette di identificare direttamente un sottoalbero. Quindi, per esempio, l'espressione $2 \cdot (3 + 4)$ si trasforma nell'espressione $\cdot\ 2\ +\ 3\ 4$ che non necessita l'uso di parentesi.

Popolazione

Rappresenta gli individui soggetti agli operatori genetici, ovvero i candidati alla soluzione del problema. Ogni individuo è un'istanza di una classe che ha un vettore rappresentante un albero generato utilizzando la grammatica G ed un valore indicante la bontà della soluzione. In fase di generazione si creano individui, in modo casuale, di altezza massima uguale ad 1. L'eventuale crescita dell'albero è lasciata, pertanto, all'applicazione degli operatori genetici.

Crossover

L'operatore implementato, presi in input due individui, verifica se l'altezza dei due alberi non supera il massimo valore predefinito. In tal caso, due nodi sono scelti casualmente (uno per ciascun albero) e scambiati. Viceversa, se l'albero ha altezza superiore alla massima consentita è sostituito con un altro di altezza pari ad 1 (generato anch'esso in modo casuale). Questa strategia è una vera e propria "potatura" dell'albero, serve per impedire che gli individui crescano notevolmente rallentando non di poco l'esecuzione dell'algoritmo. Questo problema si presenta perchè da un punto di vista semantico l'equazione $x+0$ è equivalente all'equazione $x+0+0+0$. È applicato con probabilità $p_c = 0,9$.

Mutazione

Sono stati implementati due operatori di *mutazione*, entrambi lavorano su un singolo individuo. Il primo, identifica un sottoalbero e lo sostituisce con un altro generato casualmente con altezza massima pari a quella dell'albero da sostituire. Il secondo tipo di *mutazione* seleziona un nodo dell'albero, riconosce il suo tipo (costante, variabile, funzione unaria o binaria) e lo sostituisce con un altro nodo appartenente alla stessa categoria (in tal modo gli individui creati appartengono sempre al linguaggio definito). Entrambi gli operatori di mutazione sono applicati con probabilità $p_{\mathrm{m}} = 0,7$.

Selezione

Gli individui da sottoporre agli operatori genetici sono selezionati combinando: la funzione identità nella *selection()* del pattern, la permutazione degli individui nelle sottopopolazioni e l'uso delle probabilità nell'applicazione degli operatori nella fase evolutiva.

Fitness

Si utilizza la funzione di valutazione calcolata come segue:

$$\sqrt{\frac{\sum_{i=0}^{N}(f'(x_{\mathrm{i}}) - f(x_{\mathrm{i}}))^2}{N}}$$

Il valore $f'(x_{\mathrm{i}})$ si riferisce alla funzione calcolata sul punto approssimato ottenuto durante l'evoluzione, mentre $f(x_{\mathrm{i}})$ indica il valore ottenuto applicando la funzione al punto fornito in input all'applicazione. Le funzioni utilizzate sono del tipo: $f(x) = \cos(x) + \sin(-x) + x^2 + 3 \cdot x$. È un problema da minimizzare di cui si conosce l'ottimo.

Terminazione

La funzione di terminazione dipende sia dal numero di iterazioni, sia dalla distanza ottenuta tra la funzione ricostruita e quella da ricostruire. Se tale distanza è inferiore a 10^{-6} (errore ritenuto accettabile) o sono state raggiunte il massimo numero di iterazioni permesse, si termina la ricostruzione.

6.4 Sudoku

Il gioco del Sudoku consiste nel riempire una matrice 9x9 - in cui sono presenti un determinato numero di valori (vincoli) - con una permutazione dei numeri da 1 a 9 in modo da non aver ripetizione né nelle righe né nelle colonne né nei box.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figura 6.4: Esempio sudoku.

Il gioco del Sudoku non è facilmente affrontabile da un punto di vista genetico in quanto l'algoritmo facilmente converge verso un minimo locale e diventa difficile spostare le soluzioni da questo punto. Solitamente il Sudoku è affrontato in due diversi modi:

- simulando il metodo di soluzione seguito dall'uomo. Partendo dalla matrice con i vincoli si identifica ad ogni iterazione la riga, la colonna od il box a cui manca un solo elemento per essere completata. Si inserisce il numero e si evita che questo possa essere usato nuovamente nella stessa riga, colonna o box. Finito il riempimento delle celle che hanno una sola possibilità si comincia ad analizzare quelle che ne hanno due e così via. Un'ulteriore regola consiste nel verificare se esistono due celle (nella stessa riga, colonna o box), tra quelle che hanno solo

due possibilità, con gli stessi valori ammissibili. In tal caso, una cella assumerà un valore e l'altra il restante. Questi due valori sono quindi eliminati dalle altre celle appartenenti alla stessa riga, colonna o box. Tale regola è da applicare successivamente alle celle che hanno solo tre possibilità, poi quelle con quattro e così via. Utilizzando questa strategia si riesce a completare la matrice del Sudoku, anche se non è sempre garantita la correttezza. Infatti si possono avere matrici in cui è possibile seguire più di una via, in questi casi con piccoli accorgimenti il problema è comunque risolvibile;

- usando *depth first search* (*dfs*), ovvero analizzando ogni albero presente nello spazio delle soluzioni con una visita in profondità. Ad ogni passo, si inserisce un elemento verificando che non venga generato un conflitto (ovvero una ripetizione del numero inserito nella riga, colonna o box di appartenenza). Al verificarsi di questo, si prova a modificare l'ultima cella considerata inserendo un altro numero (se possibile). Se questa non ha più valori ammisibili da esaminare, si azzera il suo contenuto e si risale nell'albero spostandosi sulla cella compilata precedentemente. A questo punto si ripete il procedimento cercando un altro possibile valore per la cella, e così via. Sostanzialmente si analizza un ramo dell'albero dello spazio delle soluzioni, nel caso pessimo si torna indietro fino alla radice scegliendone un'altra. Si ripete il tutto finchè si completa la matrice.

  Per ridurre lo spazio di ricerca, anziché selezionare la prima cella vuota disponibile (oppure una scelta casualmente), è conveniente riempire la matrice seguendo le celle in ordine crescente dei valori ammissibili. Un'altra possibile soluzione lavora con la dimensione della popolazione variabile. Partendo con un solo elemento e considerando le celle in or-

dine crescente di valori ammissibili, si estende la popolazione creando individui per ogni valore possibile presente nella cella i-esima. Ogni qual volta si verifica un conflitto l'individuo è eliminato e si prosegue con gli altri. In questo caso, bisogna trovare un accorgimento per evitare che il numero di individui cresca eccessivamente.

La soluzione implementata è sviluppata secondo il paradigma di computazione genetico e prevede di generare in modo casuale l'intera popolazione applicando gli operatori genetici fino ad ottenere la soluzione migliore. Prima di creare gli individui si effettua una fase di preprocessing in cui si analizza la matrice fornita e si determinano i valori ammissibili per ogni cella. La fase di generazione si preoccupa di inserire in modo casuale elementi in una cella curandosi del solo fatto che in ogni box debba essere presente una permutazione identica. Questa metodologia non garantisce di trovare le soluzioni a tutte le matrici fornite. In quelle etichettate come "difficili", infatti, l'algoritmo non sempre converge all'ottimo globale ma può andare in stagnazione.
Segue una descrizione degli aspetti cruciali.

Popolazione

Ogni individuo è rappresentato da un'istanza della classe che ha al suo interno: una matrice di interi 9x9 ed un valore indicante la bontà dell'individuo.

Crossover

L'operatore di *crossover* implementato riceve in input due individui e ne genera altrettanti. Dal primo si ricava un vettore contando, per ciascuna colonna, il numero dei valori unici. Si somma il contenuto delle celle *(0,1,2)*, *(3,4,5)* e *(6,7,8)* del vettore. Le colonne della matrice identificate dalla tripla con somma maggiore (le più corrette) sono assegnate al primo figlio che viene

completato prendendo le restanti sei colonne dall'altro genitore. Il secondo figlio è creato ripetendo lo stesso procedimento, scambiando il ruolo degli individui di input, con la differenza di considerare le righe anziché le colonne. È applicato con probabilità $p_{\mathrm{c}} = 0,9$.

Mutazione

La *mutazione* implementata è applicata su ogni box della matrice. In ciascuno di questi si identificano due elementi e si verifica se il valore di una cella è presente nella lista dei valori ammissibili dell'altra, in caso di esito positivo i due valori sono scambiati. Si applica con probabilità $p_{\mathrm{m}} = 0,9$.

Selezione

La selezione degli individui da sottoporre agli operatori genetici è realizzata combinando: la funzione identità nella *selection()* del pattern, la permutazione degli individui nelle sottopopolazioni e l'uso delle probabilità nell'applicazione degli operatori nella fase evolutiva.

Fitness

La funzione di valutazione conta i valori unici per riga e per colonna. Non si controlla il box in quanto il processo di generazione assicura la sua correttezza. Inoltre, gli operatori genetici usati sono implementati ponendo attenzione a non violare questa correttezza. La rappresentazione fornita rende il gioco del Sudoku un problema da massimizzare ad un valore noto, 162 (somma dei valori unici nelle righe e nelle colonne).

Terminazione

Le condizioni per terminare l'esecuzione sono due. L'aver raggiunto il mas-

simo numero di iterazioni oppure l'aver ottenuto un individuo con valore di *fitness* pari a 162.

6.5 K-means clustering

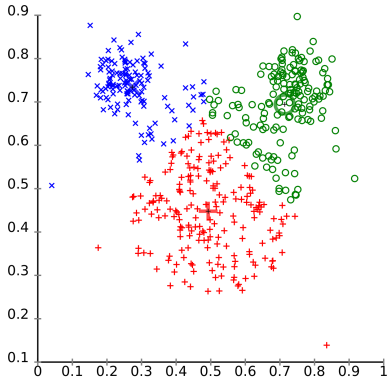


Figura 6.5: Esempio cluster[1].

L'algoritmo *K-means* è un algoritmo di clustering il cui obiettivo consiste nel partizionare i dati forniti in K gruppi contenenti oggetti simili al loro interno e dissimili fra un gruppo ed un altro. Il concetto di similarità è usualmente definito come una distanza, due oggetti sono simili se distano poco l'uno dall'altro. L'obiettivo dell'algoritmo *K-means* è quello di minimizzare la varianza intra-cluster. Per ogni gruppo si identifica un centroide e si segue tale procedura iterativa:

1. si scelgono in modo casuale K punti, questi rappresentano i centroidi dei K cluster;

2. si calcola la distanza fra ogni punto del dataset ed i K centroidi. Ciascun punto è quindi assegnato al cluster identificato dal centroide più vicino;

3. si calcola la posizione del nuovo centroide come media dei punti del suo cluster di appartenenza;

[1]Immagine da Wikipedia, `http://en.wikipedia.org/wiki/K-means_clustering`.

4. se i nuovi centroidi differiscono dai precedenti si ritorna al punto 2, altrimenti la ricerca termina.

La funzione di similarità usata e la scelta dei K centroidi identificati all'inizio, sono aspetti che possono influenzare la convergenza del *K-means*. Pertanto, l'algoritmo non garantisce la certezza di convergere su un minimo globale. L'applicazione sviluppata segue quanto citato negli articoli [16, 17, 19], qui si utilizza il pool esteso per implementare un algoritmo genetico il cui compito è far evolvere la popolazione al fine di trovare i K centroidi migliori. Fatto ciò, si esegue il classico algoritmo di clustering *K-means* con i K punti trovati. Per le verifiche è stato utilizzato il dataset "Pima Indians Diabetes Data Set" (UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes`). Si descrivono gli elementi principali dell'implementazione.

Popolazione

Gli individui sono rappresentati come un'istanza di una classe che ha un vettore contenente K punti, i centroidi, ed un campo indicante il valore di *fitness*.

Crossover

L'operatore di *crossover* implementato, riceve in input due individui. Genera in modo casuale un punto di taglio e crea i due figli assegnandogli le parti alternate dei genitori sezionati in tale punto. È applicato con probabilità $p_c = 0,7$.

Mutazione

L'implementazione fornita di questo operatore identifica, per l'individuo ricevuto in input, uno dei K centroidi e lo sostituisce con uno dei punti presenti

nel dataset. È applicato con probabilità $p_{\mathrm{m}} = 0,3$.

Selezione

Gli individui sottoposti agli operatori genetici sono selezionati combinando: la funzione identità nella *selection()* del pattern, la permutazione degli individui nelle sottopopolazioni e l'uso delle probabilità nell'applicazione degli operatori nella fase evolutiva.

Fitness

Calcola la somma delle distanze euclidee al quadrato dei punti dal loro centroide, ovvero:

$$\sum_{i=1}^{K} \sum_{x \in C_{\mathrm{i}}} dist^2(m_{\mathrm{i}}, x)$$

Terminazione

L'algoritmo termina quando è concluso il numero massimo di iterazioni richieste.

6.6 Knapsack

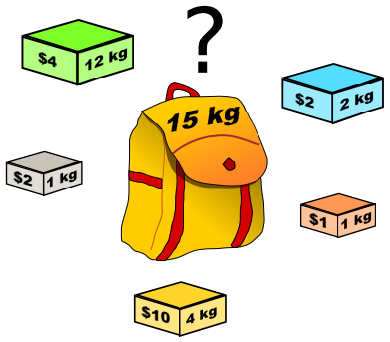


Figura 6.6: Esempio Knapsack[2].

Il problema dello zaino, o *Knapsack problem* che dir si voglia, è un problema di ottimizzazione combinatoria. Dato uno zaino con capacità C ed n oggetti, caratterizzati anch'essi da almeno due attributi (un peso ed un valore), il problema consiste nello scegliere opportunamente gli oggetti da inserire nello zaino senza eccedere la sua capacità, massimizzando il valore ottenuto. Questo problema si presta bene per modellare diverse situazioni della realtà. Oltretutto, se anziché considerare un singolo valore abbiamo un vettore di valori, si rientra nella categoria dei problemi multi-obiettivo trattati nella sezione 3.1. In questa applicazione utilizziamo il *Knapsack problem* per modellare un problema di ottimizzazione finanziaria bi-obiettivo descritto di seguito.

Sono dati n investimenti ai quali è associato per ciascuno un costo, un profitto ed un livello di sicurezza. Stabilito il massimo valore disponibile per gli investimenti, il problema consiste nell'identificare l'insieme degli investimenti che garantiscono maggior profitto e maggior sicurezza. In questo caso entrambi gli obiettivi sono da massimizzare (in generale non è così, come già descritto nella sezione 3.1). Nella soluzione fornita si è utilizzata la tecnica

[2]Immagine da Wikipedia, `http://en.wikipedia.org/wiki/Knapsack_problem`.

della *scalarizzazione* assegnando lo stesso peso ($\alpha = 0.5$) ad entrambi gli obiettivi, in tal modo il problema diventa un mono-obiettivo da massimizzare. Si descrivono i punti principali dell'implementazione.

Popolazione

La popolazione è rappresentata da un'istanza di una classe che ha al suo interno un vettore booleano di lunghezza pari al numero di investimenti considerati. In tale vettore un valore *true* nella posizione i-esima indica la presenza dell'oggetto nello zaino, viceversa, un valore *false* indica che l'investimento è stato scartato. Inoltre, nella classe è presente un altro attributo - di valore reale - che indica la bontà dell'individuo.

Crossover

L'operatore è implementato come il classico *crossover* ad un taglio. Presi in input due individui, genera in modo casuale un punto di taglio e crea i due figli assegnandogli in modo alternato il patrimonio genetico dei genitori sezionati in tale punto. È applicato con probabilità $p_{\mathrm{c}} = 0,9$.

Mutazione

L'operatore è implementato in modo da identificare una posizione, generata in modo casuale, nel vettore e negare il valore booleano presente. È applicato con probabilità $p_{\mathrm{m}} = 0,3$.

Selezione

La fase di selezione degli individui da sottoporre agli operatori genetici è realizzata combinando: la funzione identità nella *selection()* del pattern, la permutazione degli individui nelle sottopopolazioni e l'uso delle probabilità

nell'applicazione degli operatori nella fase evolutiva.

Fitness

Come spiegato sopra, il problema è stato trasformato utilizzando la *scalarizzazione* per cui non ci si preoccupa di calcolare l'intera *frontiera di Pareto* (come già descritto nella sezione 3.1). Sia dato il peso α, la funzione da massimizzare diventa:

$$\sum_{i=1}^{N} \alpha f_1 + (1 - \alpha) f_2$$

Terminazione

L'algoritmo termina una volta concluse le iterazioni richieste.

Capitolo 7

Risultati sperimentali

Il pattern fornito è implementato come un'ulteriore classe C++ del framework Fastflow. Il pattern è realizzato per operare su architetture multicore. Tutti i test sono stati effettuati su Titanic (12 core con frequenza 800.000 MHz, 2 thread per core, 512KB di cache L2 e 128 KB di L1 per core) e Xeon Phi (60 core con frequenza di 1052.630 MHz, 4 thread per core, 512 KB di cache L2 e 32 KB di L1 per core). I grafici che seguono sono ottenuti con quest'ultima macchina mediando i risultati di dieci elaborazioni ottenuti facendo variare il numero di core da 1 a 60. Nei grafici, si mostrano (nella legenda si specifica: con la label *v2* l'output ricavato con il pattern *pool evolution*, senza la label i risultati ottenuti con l'estensione fornita): il tempo di completamento, la scalabilità e l'efficienza (i valori riportati sono approssimati alla seconda cifra decimale).

Hello world genetic

Confrontando i risultati si ottiene che:

- il minimo tempo di completamento (T_C) del nuovo pattern è raggiunto utilizzando 59 core ed è uguale a $693,93ms$ con una scalabilità di $38,72$;
- viceversa il vecchio pattern paga l'operazione di creazione nuovi individui, nella fase di selection, ottenendo come minimo T_C $44445,62ms$. Valore ottenuto sfruttando solo 2 core, con una scalabilità di $1,32$;
- la massima differenza fra le scalabilità, ottenute variando il grado di parallelismo da 1 a 60, è $38,40$ a favore del nuovo pattern (valore conseguito con 59 core);
- la massima differenza fra le due efficienze, al variare dei *processing element* usati, è $0,90$ per l'estensione fornita (con 11 core).

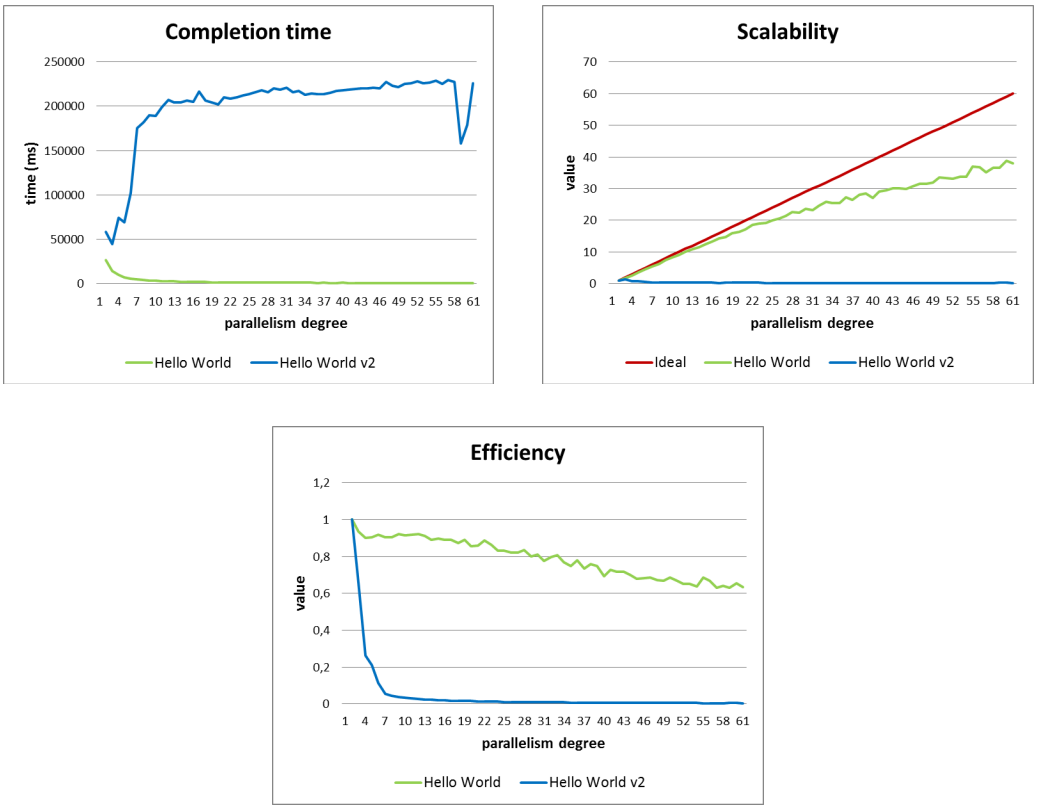


Figura 7.1: *Hello world genetic.* Run con 1024 individui, 2048 iterazioni e frequenza di rimescolamento degli individui pari a 64. Hello world è relativo alla nuova versione del pattern, con sottopopolazioni, Hello world v2 a quella vecchia.

Minimum of a function

Segue un raffronto tra gli output conseguiti:

- il minimo tempo di completamento T_C raggiunto con l'estensione fornita è pari a $161,30ms$. A tale valore corrisponde una scalabilità di $9,64$, valori ricavati con 58 core;
- il minimo T_C ricavato con il *pool evolution* presente nel framework è di $2342,29ms$ ottenuto con 5 core. A questo corrisponde una scalabilità di $1,27$. Tali risultati sono da attribuire alla creazione di nuovi individui che trasforma la fase di selezione in un "collo di bottiglia" degradando le prestazioni in modo significativo;
- la massima differenza fra le due scalabilità, ottenute variando il grado di parallelismo da 1 a 60, è $9,28$ a favore della nuova versione (valore conseguito con 52 core);
- la massima differenza fra le due efficienze, al variare dei *processing element* utilizzati, è $0,33$ a favore della nuova versione (valore ricavato usando 17 core).

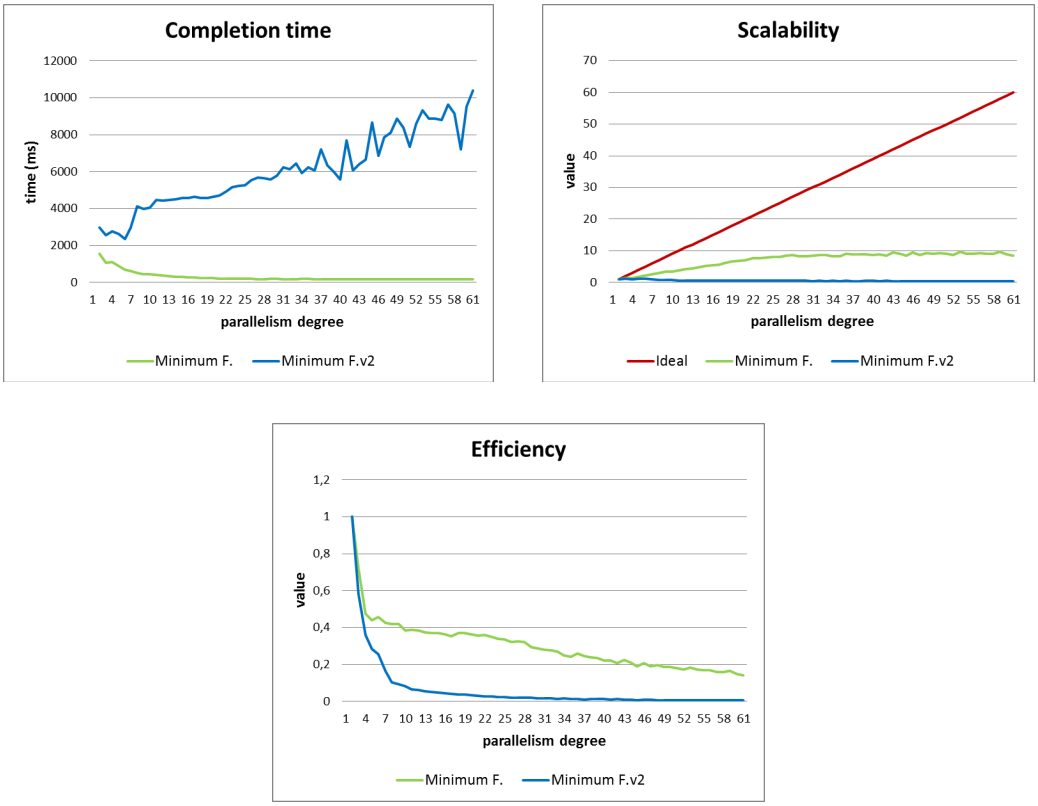


Figura 7.2: *Minimum of a function.* Run con 1024 individui, 1024 iterazioni, scambio individui (fra le isole) ogni 64 iterazioni, intervallo [-10; 10] e $f(x) = x^{12} - x^8 - x^5 + \frac{x}{2} - 1,14$. Minimum F. è relativo alla nuova versione del pattern, con sottopopolazioni, Minimum F. v2 a quella vecchia.

Compute function given n known points

Confrontando i risultati si ottiene che:

- il minimo tempo di completamento (T_C) ricavato con la nuova versione è di $2477,25ms$ con una scalabilità di $26,39$, contro un T_C di $3942,78ms$ ed una scalabilità di $18,24$ (vecchia versione). Valori ricavati utilizzando 58 core;
- la massima differenza fra le due scalabilità, variando i *processing element* usati da 1 a 60, è di $10,70$ a favore del nuovo pattern (valore ricavato sfruttando 60 core);
- la massima differenza fra le due efficienze, al variare del grado di parallelismo, è di $0,21$ per l'estensione fornita (valore raggiunto con 38 core).

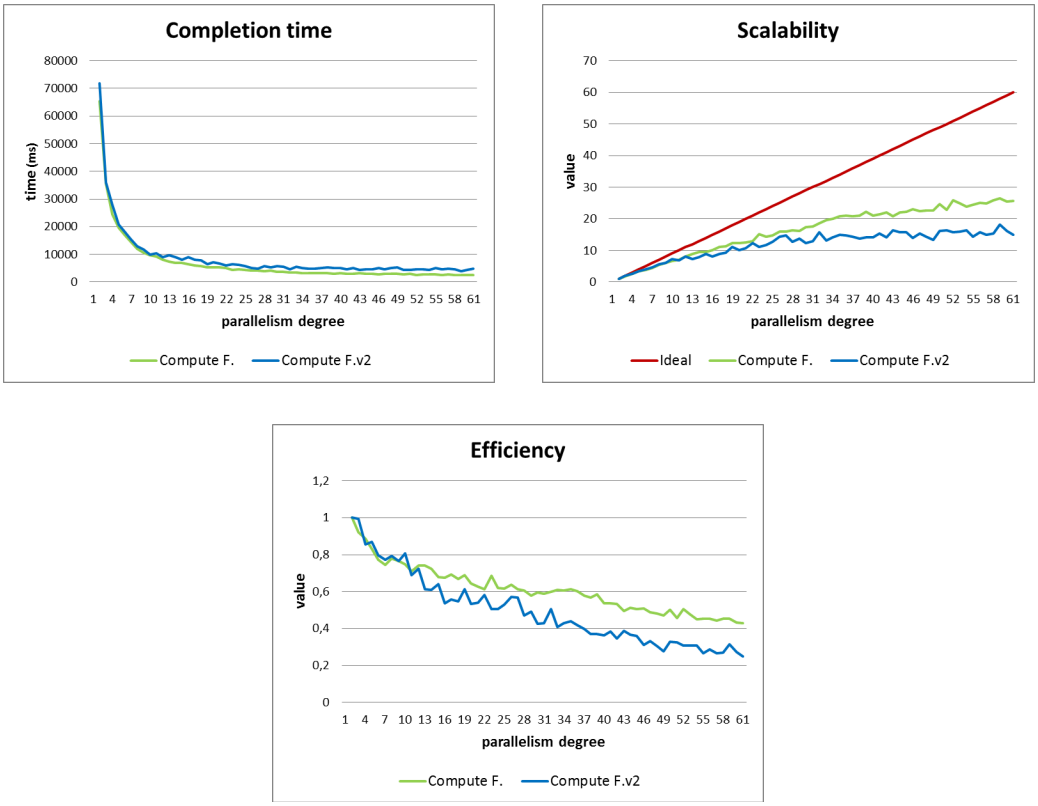


Figura 7.3: *Compute function given n known points.* Run con 128 individui, 128 iterazioni e frequenza di scambio degli individui (fra le isole) uguale a 64. Compute F. è relativo alla nuova versione del pattern, con sottopopolazioni, Compute F. v2 a quella vecchia.

Sudoku

Come descritto nella sezione 6.4, la soluzione fornita per questo problema dipende molto dalla casualità. Se, infatti, l'algoritmo converge su un minimo locale allora probabilmente non si muoverà da questo, ritornando come soluzione migliore una matrice che ha mediamente 2 errori. Viceversa, i tempi di soluzione possono essere molto brevi (circa $90ms$). Segue una comparazione tra le due versioni:

- il minimo tempo di completamento (T_C) del nuovo pattern è di $1259,47ms$ raggiunto sfruttando 43 core, la scalabilità è pari a $7,39$;
- viceversa, il minimo T_C raggiunto con il vecchio pattern è di $1830,20ms$ ricavato con 3 core e la scalabilità è di $8,06$ (superlineare, poichè 8 esecuzioni su 10 hanno trovato l'ottimo globale in un tempo medio di $351,57ms$);
- a causa della superlinearità, nella vecchia versione, si raggiunge quasi $2,69$ di efficienza con 3 core;
- la massima differenza fra le due scalabilità, al variare dei *processing element* da 1 a 60, è di $5,49$ a favore del nuovo pattern *pool evolution* (con 43 core);
- per la superlinearità la massima differenza fra le due efficienze è circa $2,03$ a favore della vecchia versione del pattern *pool evolution* (con 3 core).

Figura 7.4: *Sudoku.* Run eseguito con 128 individui, 1024 iterazioni e rimescolamento individui ogni 64 iterazioni. Sudoku è relativo alla nuova versione del pattern, con sottopopolazioni, Sudoku v2 a quella vecchia.

K-means clustering

Come descritto nella sezione 6.5, con questo problema si vuole mostrare un diverso utilizzo del pattern. Si utilizza il pool esteso per implementare un algoritmo genetico il cui compito è far evolvere la popolazione al fine di trovare i K centroidi migliori. Fatto ciò, si esegue il classico algoritmo di clustering *K-means* con i K punti trovati.

In figura 7.5 si conferma quanto detto negli articoli [16, 17, 19], ovvero utilizzare i K centroidi iniziali restituiti dall'algoritmo genetico (piuttosto che selezionarli in modo casuale) permette di ridurre il numero di iterazioni necessarie affinchè l'algoritmo *K-means* termini.

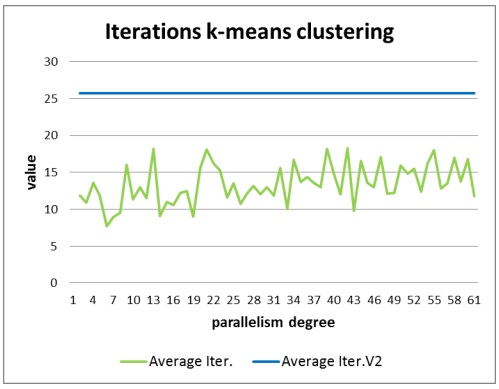


Figura 7.5: *Kmeans.* Run con 128 individui, 128 iterazioni, 4 cluster richiesti con scambio degli individui fra le isole ogni 64 iterazioni. Average Iter. è relativo alla nuova versione del pattern, con sottopopolazioni, Average Iter. v2 è la media ottenuta dall'esecuzione dell'algoritmo *K-means*.

Knapsack

Dal confronto dei risultati ottenuti con le due versioni segue che:

- il minimo tempo di completamento (T_C) della nuova versione è di $151,68ms$, con una scalabilità di $10,28$ (sfruttando 60 core);
- il minimo T_C conseguito con il vecchio pattern è di $3076,53ms$, con una scalabilità di $1,23$ (valori ricavati con 2 core). Anche in questo problema le continue creazioni di nuovi individui degradano le prestazioni in modo significativo;
- la massima differenza fra le due scalabilità, al variare del grado di parallelismo tra 1 e 60, è di $9,95$ a favore della nuova versione (valore ottenuto con 60 core);
- la massima differenza fra le due efficienze, variando i core utilizzati, è di $0,37$ a favore della nuova versione fornita (con 17 core).

Figura 7.6: *Knapsack*. Run eseguito con 1028 individui, 1024 iterazioni e frequenza di rimescolamento individui uguale a 64. Knapsack è relativo alla nuova versione del pattern, con sottopopolazioni, Knapsack v2 a quella vecchia.

7.1 Conclusioni

Come discusso e mostrato nei grafici di questo capitolo, la versione del pattern che implementa il modello ad isole con sottopopolazioni garantisce sempre migliori risultati sulle applicazioni sviluppate. Sebbene le differenze ottenute nelle misure di performance di tempo di completamento, scalabilità ed efficienza siano molto più significative in quei casi dove la computazione è a grana fine (dovuto al "collo di bottiglia" che viene a crearsi nella fase di selezione della vecchia versione); tali miglioramenti si presentano, in misura meno significativa, anche in quei casi in cui la computazione non è a grana fine (vedi risultati ottenuti con l'applicazione *Compute function given n known points*).

Capitolo 8

Modello ad isole con multi-popolazione

Nel capitolo 5 si è mostrato come l'estensione del pattern *pool evolution* permette di trattare il modello ad isole con sottopopolazioni descritto nella sezione 3.2. Con tale estensione, chiaramente, è possibile sfruttare i metodi definiti nella classe *poolEvolution*. Un limite di questa versione è rappresentato dal fatto che le sottopopolazioni hanno la stessa dimensione e condividono i metodi di evoluzione. Per fornire un'implementazione dello stesso modello ma operante su multi-popolazione, si sono dovute definire due nuove classi. La prima - *Evolution* - fornisce un'implementazione per trattare gli algoritmi genetici; la seconda - *EvolutionE* (che sfrutta la precedente) - permette di lavorare su multi-popolazione garantendo i vantaggi descritti a pagina 103.

Iniziamo con il descrivere la classe *Evolution*. I parametri di input aggiunti o modificati al costruttore (vedi figura 8.1) del pattern *pool evolution*, descritto nella sezione 4.3, sono:

1. la dimensione della popolazione dell'isola piuttosto che la struttura dati contenente gli individui;

2. un metodo *generate()* che riceve in input un vettore e l'ambiente locale dell'isola. Il suo compito è quello di generare gli individui;

3. un metodo *ag()* per l'evoluzione genetica in cui il programmatore definirà tutte le fasi necessarie (ovvero *selection*, *evolution* e *filter* non saranno più passate come parametri).

Il costruttore si occuperà di inizializzare la popolazione (non più nel metodo *svc()* della classe) e creare una struttura dati - come descritto nel capitolo 5 - per mezzo del metodo *Clone()* di cui è richiesta la definizione al programmatore all'interno della classe rappresentante l'individuo.

```
Evolution(size_t maxp,        /* maximum parallelism degree in all phases */
          size_t size,                /*     population size              */
          generate_t gen,             /*     the generation function      */
          ag_t ag,                    /*     the evolution function       */
          termination_t term,         /*     the termination function     */
          const env_t &E = env_t());  /*     user's environment           */
```

Figura 8.1: Costruttore della classe *Evolution*.

Si aggiungono o si modificano, alla versione del pattern *pool evolution* presente in FastFlow, i seguenti metodi:

- *get_input()*: ritorna l'indirizzo della popolazione creata;
- *get_buffer()*: restituisce l'indirizzo della popolazione creata dalla clonazione;
- *getEnv()*: ritorna l'indirizzo dell'ambiente;
- *getIte()*: restituisce il numero di iterazioni, se previste, viceversa il valore *INT_MAX*;

- *setIte(i)*: setta il numero di iterazioni, se previste, ad il valore i ricevuto in input;

Il codice sorgente di tale classe è presente in appendice A.2.

Per implementare il modello ad isole operante su multi-popolazione definiamo la classe *EvolutionE*. I parametri *maxp*, *pop*, *sel*, *evolP*, *fil* e *term* del costruttore della classe definita nel capitolo 5 sono rimpiazzati da un vettore contenente tutte le isole. L'i-esima posizione di questo contiene un'istanza della classe *Evolution*[1] descritta sopra. Inoltre, in questa versione, il metodo *mix()* riceve in input soltanto il vettore contenente le isole. In figura 8.2 si mostra il costruttore della classe.

```
EvolutionE(std::vector<T> &input,      /*      the islands          */
           mix_t m,                    /*      the mixing function  */
           size_t d);                  /*      the mixing frequency */
```

Figura 8.2: Costruttore della classe *EvolutionE*.

In appendice A.3 si riporta il codice sorgente dell'implementazione della classe descritta.

Questa nuova definizione permette maggiore flessibilità, facilità di utilizzo e personalizzazione al programmatore. Infatti, come già detto, il vettore fornito al costruttore contiene istanze della classe *Evolution*. Ciascuna di queste è indipendente dalle altre, quindi ogni isola creata può avere:

- una differente dimensione della popolazione;
- un differente metodo di generazione;
- un differente metodo di terminazione;

[1]Essendo il vettore di tipo T, può essere passata una qualunque altra classe - con un differente codice funzionale - che implementi i metodi di *Evolution* (vedi Appendice A.2).

- un differente metodo che si occupi dell'evoluzione genetica.

Queste differenze tra le isole non possono essere realizzate con la versione, descritta nel capitolo 5, che opera su sottopopolazioni. Infatti, l'implementazione di quella classe impone alle isole di condividere i metodi di terminazione e di evoluzione; oltretutto, ogni isola è definita con lo stesso numero di individui (a meno di arrotondamenti).

Un altro vantaggio da sottolineare è che questa versione permette una gestione dello scambio degli individui più semplice. Le isole sono posizioni differenti di un vettore; per scambiare individui dall'isola x all'isola y non occorre tenere in considerazione alcun *offset* per muoversi all'interno della stessa struttura dati, ma si considerano semplicemente posizioni diverse del vettore. Un'implementazione siffatta permette anche maggiore personalizzazione nell'affrontare il problema.

Per dimostrare ciò, si consideri il problema della ricerca del minimo globale di una funzione descritto nella sezione 6.2. Il metodo di generazione usato riceve in input un range di ricerca ed il numero di punti (individui) richiesti. Per determinare la posizione degli individui, si divide l'intervallo per il numero di punti dati e si posiziona ogni individuo a distanza regolare l'uno dall'altro. Con la versione multi-popolazione, visto che le isole possono avere popolazioni di dimensione differente, è possibile partizionare il range di generazione fra le isole. In tal modo si può, per esempio, creare un maggior numero di individui in un sotto-intervallo dove si suppone sia presente il minimo globale e meno negli altri, aumentando così le probabilità di trovare l'ottimo. Questo modo di operare influenza la velocità di convergenza e la bontà delle soluzioni trovate.

Si consideri la funzione in figura 8.3:

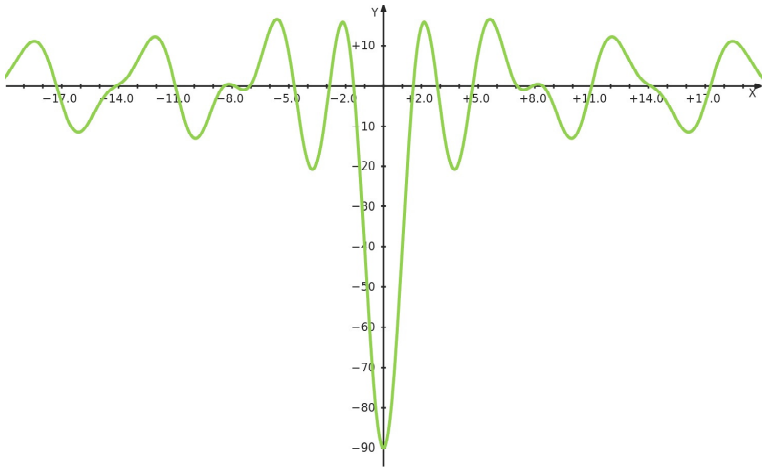


Figura 8.3: $f(x) = -50 \cdot \frac{sin(2 \cdot x)}{x} + 10 \cdot cos(x)$.

Confrontiamo gli output ottenuti cercando il minimo globale utilizzando il modello ad isole con multi-popolazione e con sottopopolazioni. I parametri di input utilizzati sono:

- grado di parallelismo (numero di isole) pari a 4;
- 64 individui complessivi;
- 128 e 64 iterazioni;
- operazione di scambio degli individui eseguita ogni 32 iterazioni;
- range di analisi della funzione [-8000, 6000].

La fase di generazione degli algoritmi utilizza la metodologia descritta precedentemente. Quindi, nella versione operante su sottopopolazioni, ogni individuo è stato piazzato ad una distanza in x di $218,75$. Viceversa, nella versione con multi-popolazione si sono costruiti i sotto-intervalli: [-8000; -4500], [-4500; -1000], [-1000; 2500], [2500; 6000]. Ciascuno di questi è stato assegnato al metodo di generazione di ogni isola. Si è supposto di avere alte probabilità che il minimo fosse in $x = 0$. Nell'isola avente tale punto

nel suo intervallo di generazione sono stati creati 40 individui contro gli 8 creati per ciascuna delle restanti. Gli esperimenti sono stati ripetuti 100 volte per entrambe le versioni descritte. La funzione data ha un minimo nel punto di ascissa $x = 0$ con valore $f(x) = -90$. Nelle esecuzioni con 128 iterazioni il valore medio del minimo dell'ordinata ottenuto con la versione operante su multi-popolazione è di $-72,09$ contro $-66,90$ di quella operante su sottopopolazioni. Invece, nell'output ottenuto con 64 iterazioni (mostrato in figura 8.4), la multi-popolazione ottiene un valore medio del minimo dell'ordinata pari a $-51,90$ restituendo 46 volte un risultato con errore sotto il 10%. Viceversa, il valore medio del minimo dell'ordinata ottenuto con le sottopopolazioni è $-48,81$ con solo 12 risultati aventi errore sotto il 10%.

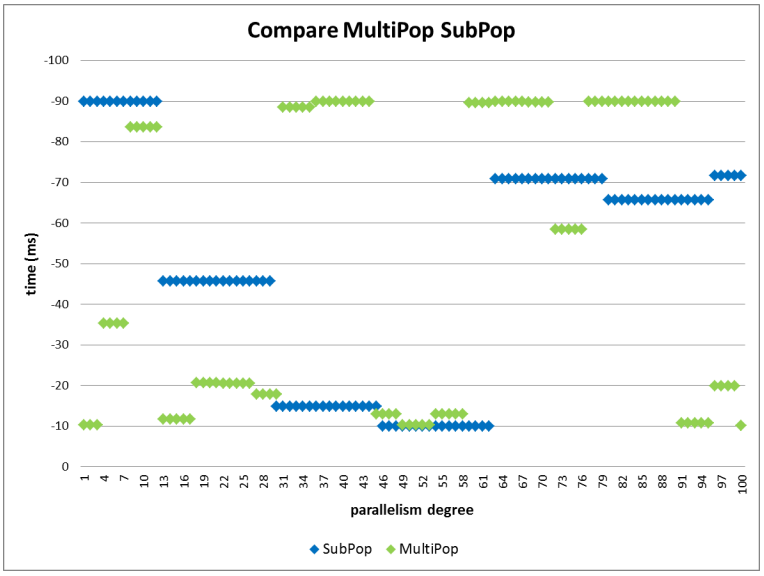


Figura 8.4: Find the minimum. Confronto tra multi-popolazione e sottopopolazioni. Run con 64 individui, 64 iterazioni, frequenza di scambio individui (fra le isole) pari a 32, intervallo [-8000; 6000] e $f(x) = -50 \cdot \frac{sin(2 \cdot x)}{x} + 10 \cdot cos(x)$.

L'algoritmo usato per far migrare gli individui da un'isola ad un'altra è sintetizzato nelle tabelle mostrate in figura 8.1. L'idea è quella di portare sull'isola i un certo numero di individui proveniente da tutte le restanti isole. Assumendo di avere popolazioni di uguale dimensione, si ha:

A	**B**	**C**	**D**
x_1	x_5	x_9	x_{13}
x_2	x_6	x_{10}	x_{14}
x_3	x_7	x_{11}	x_{15}
x_4	x_8	x_{12}	x_{16}

A	**B**	**C**	**D**
x_5	x_1	x_2	x_3
x_9	x_{10}	x_6	x_7
x_{13}	x_{14}	x_{15}	x_{11}
x_4	x_8	x_{12}	x_{16}

Tabella 8.1: Settori di popolazione delle isole **A**, **B**, **C** e **D**. Da sinistra a destra si mostra la situazione di ogni isola prima e dopo lo scambio.

8.1 Conclusioni

Come discusso in questo capitolo, il motivo dell'introduzione di un pattern per il modello ad isole con multi-popolazione deriva dal voler fornire maggiore personalizzazione nell'implementazione di una soluzione ad un problema affrontato secondo il paradigma di computazione genetica. La nuova versione non nasce, quindi, per migliorare le performance del pattern operante su sottopopolazioni. Tuttavia, nei problemi in cui la variabilità fra le isole del numero di individui e/o dei metodi di evoluzione utilizzati risulta essere significativa, la soluzione con multi-popolazione può garantire risultati migliori. Nell'esempio descritto a pagina 105, per esecuzioni con 64 iterazioni, il numero di valori che hanno un errore medio inferiore al 10% è maggiore nella nuova versione (in cui varia il numero degli individui fra le isole) rispetto a

quello ottenuto utilizzando il modello ad isole con sottopopolazioni, questo numero tende a coincidere con il crescere del numero delle iterazioni.

Capitolo 9

Conclusioni

La tesi presenta due implementazioni per trattare il modello di computazione genetica detto "ad isole". Ovvero, implementazioni adatte per quei problemi la cui soluzione è ricavabile dal lavoro svolto su più popolazioni con l'obiettivo di aumentare la velocità e/o la qualità della soluzione trovata. La prima implementazione, ricavata per estensione diretta del pattern *pool evolution* [4] (rispettando quindi le segnature dei suoi metodi) presente nel framework di programmazione parallela strutturata FastFlow [3, 6, 10, 13], opera su sottopopolazioni; mentre la seconda, ricavata definendo due nuove classi, opera su singola multi-popolazione.

Nel dettaglio, le versioni fornite implementano una variante del modello ad isole che prevede lo scambio di informazioni fra queste al fine di aumentare la variabilità dell'intera popolazione, riducendo il rischio del fenomeno di convergenza verso minimi locali (stagnazione).

I risultati ottenuti mostrano che per computazioni a grana fine le nuove versioni fornite garantiscono risultati migliori in termini di tempi di completamento, scalabilità ed efficienza. Questo è principalmente dovuto alla scelta di gestire il lavoro clonando la popolazione iniziale ed operando su un'area

di memoria temporanea. Infatti, mentre il costo di creazione di un nuovo individuo ad ogni iterazione nella fase di selezione è mascherato nel caso in cui la fase di evoluzione abbia una certa complessità, tale costo si trasforma in un vero e proprio "collo di bottiglia" nel caso di computazioni a grana fine.

La versione introdotta per trattare il modello ad isole su singola multi-popolazione garantisce maggiore flessibilità, facilità di utilizzo e personalizzazione al programmatore. In tal modo ogni isola è indipendente dalle altre, questo permette a ciascuna di: creare gli individui con un diverso metodo di generazione, operare su popolazioni di dimensione differente, usare metodi distinti per la terminazione e per l'evoluzione genetica.

Queste differenze tra le isole non possono essere realizzate con la versione, descritta nel capitolo 5, ottenuta per estensione del pattern *pool evolution*. Infatti, l'implementazione di quella classe impone alle isole di condividere i metodi di terminazione e di evoluzione e di operare su una popolazione di uguale dimensione (a meno di arrotondamenti). Tra i possibili sviluppi futuri si potrebbe:

1. identificare per ogni applicazione sviluppata il miglior insieme dei parametri (numero individui, probabilità di applicazione degli operatori, numero delle isole e grado di parallelismo al loro interno, numero iterazioni e frequenza di scambio) e valutare l'effetto di altre tecniche di selezione e filtraggio per ottenere l'ottimo globale del problema di ottimizzazione trattato, massimizzando il più possibile le misure di performance;

2. definire un modello che in base ad alcuni parametri (come per esempio il numero degli individui) stabilisca il valore ottimo del numero delle isole e del grado di parallelismo al loro interno.

Bibliografia

[1] *Pareto efficiency.* http://en.wikipedia.org/wiki/Pareto_efficiency, 9 April 2015.

[2] *Multi-objective optimization.* http://en.wikipedia.org/wiki/Multi-objective_optimization, 5 April 2015.

[3] *FastFlow* home page. http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about, March 2015.

[4] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick and M. Torquati. *Pool evolution: a parallel pattern for evolutionary and symbolic computing*. Springer US, March 18, 2015.

[5] Modelli e Metodi di Supporto alle Decisioni. http://www.or.unimore.it/corsi/MMSD/PM/Slides/PM.pdf, A.A. 2014/15.

[6] M. Aldinucci, M. Danelutto, P. Kilpatrick and M. Torquati. *FastFlow: high-level and efficient streaming on multi-core*, in Programming Multi-core and Many-core Computing Systems, S. Pllana and F. Xhafa, Ed., Wiley, 2014.

[7] M. Danelutto and M. Torquati. *Loop parallelism: a new skeleton perspective on data parallel patterns*, in Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing, Torino, Italy, 2014.

[8] A. J. Umbarkar and M. S. Joshi. *0/1 Knapsack Problem using Diversity based Dual Population Genetic Algorithm*. I.J. Intelligent Systems and Applications, 2014, 10, 34-40.

[9] M. Danelutto. *Distributed System: Paradigms and Models*. Support material - Laurea magistrale in Computer Science and Networking. Pisa, versione September 25, 2013.

[10] M. Danelutto. *Structured parallel programming in FastFlow*. CoreGRID Programming model Institute. July 2013, Cluj, Romania.

[11] Y. Sato, N. Hasegawa and M. Sato. *Acceleration of Genetic Algorithms for Sudoku Solution on Many-core Processors*. Springer Berlin Heidelberg, July 08, 2013, pp. 421-444.

[12] A. Chiu , E. Nasiri and R. Rashid. *Parallelization of Sudoku*. University of Toronto December 20, 2012.

[13] M. Aldinucci, M. Danelutto, M. Torquati, M. Meneghin, P. Kilpatrick et al. *FastFlow: high-level programming patterns with non-blocking lock-free run-time support*. Politecnico di Milano, Dipartimento di Elettronica ed informazione, Milano, Italy, December 5, 2012.

[14] M. Aldinucci, M. Danelutto, M. Torquati, M. Meneghin and P. Kilpatrick. *Accelerating code on multi-cores with FastFlow*. Talk given at Euro-Par 2011, Bordeaux, France. September 2011.

[15] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin and M. Torquati. *Accelerating code on multi-cores with FastFlow*, in Proc. of 17th Intl. Euro-Par 2011 Parallel Processing, Bordeaux, France, 2011, pp. 170-181.

[16] V. Chittu and N. Sumathi. *A Modified Genetic Algorithm Initializing K-Means Clustering*. Global Journal of Computer Science and Technology, Volume 11 Issue 2 Version 1.0 February 2011.

[17] K. Chander, Dr. D. Kumar and V. Kumar. *Enhancing Cluster Compactness using Genetic Algorithm Initialized K-means*. International Journal of Software Engineering Research & Practices Vol.1, Issue 1, Jan, 2011.

[18] Modelli di Sistemi di Produzione/Servizio. `http://www.or.unimore.it/corsi/MSP_MSS/MultiObjective.pdf`, A.A. 2010/11.

[19] B. Al-Shboul and S.-H. Myaeng. *Initializing K-Means using Genetic Algorithms*. World Academy of Science, Engineering and Technology 54 2009.

[20] T. Mattson, B. Sanders and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[21] M. Cole. *Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming*. Parallel Computing, 30(3):389-406, 2004.

[22] G. Folino. *Algoritmi evolutivi e programmazione genetica: strategie di progettazione e parallelizzazione*. Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni. Dicembre 2003.

[23] T. Goel and K. Deb. *Hybrid methods for multi-objective evolutionary algorithms*. 2001.

[24] G. Spezzano and D. Talia. *Calcolo parallelo, automi cellulari e modelli per sistemi complessi*. FrancoAngeli, 1999.

[25] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag Berlin Heidelberg, 1996.

[26] M. Bucci and D. Circelli. Tesi di Laurea: *Algoritmi genetici e simulated annealing per la soluzione parallela di problemi di ottimizzazione combinatoriale*. Pisa 1995/1996.

[27] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. ad Distrib. Computing. Pitman, 1989.

[28] L. Lamport. *Specifying concurrent program modules*. ACM Transactions on Programming Languages and Systems, 5(2):190–222, April 1983.

[29] P. Berggren and D. Nilsson. *A study of Sudoku solving algorithms*. `http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group6Alexander/final/Patrik_Berggren_David_Nilsson.report.pdf`.

[30] M. Schermerhorn. *A Sudoku Solver*. `https://www.cs.rochester.edu/u/brown/242/assts/termprojs/Sudoku09.pdf`. CSC 242: Artificial Intelligence.

[31] *GenD: Genetic Doping Algorithm - Le tribù*. `http://www.semeion.it/semeion2/index.php/it/algoritmi-genetici/gend-genetic-doping-algorithm/le-tribu`. Semeion Centro Ricerche di Scienza della Comunicazione.

Appendice A

Nelle sezioni che seguono si riporta il codice sorgente delle classi definite. In A.1 quello del pattern che implementa il modello ad isole con sottopopolazioni, ricavato per estensione del pattern *pool evolution* di FastFlow. In A.2 è riportato il codice sorgente della classe *Evolution*, implementata per trattare gli algoritmi genetici. In A.3, infine, quello della classe *EvolutionE* ottenuta per estensione della *Evolution*, che permette di implementare il modello ad isole con multi-popolazione.

A.1 Sottopopolazioni

```
1  #include <iostream>
   #include <vector>
   #include <ff/node.hpp>
   #include <ff/parallel_for.hpp>
5  #include <ff/poolEvolution.hpp>
   #include <unordered_set>

   namespace ff{
9
     template<typename T, typename env_t=char>
     class poolEvolutionE: public ff::poolEvolution<T, env_t>{
       public:
13       typedef poolEvolution<T, env_t> pool;
         typedef void (*evolution_tp) (ParallelForReduce<T> &, std::vector<T>&,
                       const env_t&, const int);
         typedef void (*mix_t)(std::vector<T> &, std::vector<T> &);
17       typedef bool (*compareE_t)(env_t e1, env_t e2);

       protected:
         size_t delta, nwE, ite_max = INT_MAX;
21       evolution_tp evolutionP;
         mix_t mix;
         compareE_t compareE = NULL;
         std::vector<env_t> env_tmp;
25       std::vector<ParallelForReduce<T> *> loopevol_tmp;

       public:

29       poolEvolutionE(size_t maxp, std::vector<T> & pop,
                        typename pool::selection_t sel,
                        evolution_tp evolP,
                        typename pool::filtering_t fil,
33                      typename pool::termination_t term,
                        mix_t m,
                        size_t d,
                        const env_t &E= env_t(),
37                      bool spinWait=true)
                        :poolEvolution<T, env_t>(maxp, pop, sel, NULL, fil,
                         term, E, spinWait), evolutionP(evolP), mix(m),
```

```
                        delta(d){
41                        assert(delta > 0 &&
                                 ''delta must be greater than 0.'');
                          env_tmp.resize(this->pE);
                          for(int i=0; i<this->pE; ++i){
45                          env_tmp[i] = this->env;
                            loopevol_tmp.push_back(NULL);
                          }
                        }
49
      poolEvolutionE(size_t maxp,
                     typename pool::selection_t sel,
                     evolution_tp evolP,
53                   typename pool::filtering_t fil,
                     typename pool::termination_t term,
                     mix_t m,
                     size_t d,
57                   const env_t &E= env_t(),
                     bool spinWait=true)
                     :poolEvolution<T, env_t>(maxp, sel, NULL, fil, term,
                      E, spinWait), evolutionP(evolP), mix(m), delta(d){
61                     assert(delta > 0 &&
                              ''delta must be greater than 0.'');
                       env_tmp.resize(this->pE);
                       for(int i=0; i<this->pE; ++i){
65                       env_tmp[i] = this->env;
                         loopevol_tmp.push_back(NULL);
                       }
                     }
69
      int getIte() { return ite_max; };
      void setIte(int i) { ite_max = i; };
      void setCompareE(compareE_t e) { compareE = e; };
73
      void enablePR(int nw){
        if(loopevol_tmp[0] == NULL){
          nwE = nw;
77        for(int i=0; i<this->pE; ++i){
            loopevol_tmp[i] = new ParallelForReduce<T>(nwE, true);
            loopevol_tmp[i]->disableScheduler(true);
          }
```

```
81          }
          };

        protected:

85
          void* svc(void* task) {
            if (task) this->input = ((std::vector<T>*)task);
            for(const auto &e : (*this->input))
89            this->buffer.push_back(e->Clone());

            int cnt = 0;
            std::unordered_set<T> copy_buffer;
93
            copy_buffer.insert(this->buffer.begin(), this->buffer.end());
            std::vector< std::vector<T> > tP(this->pE);
            std::vector< std::vector<T> > tBuf(this->pE);
97
            if(this->pE>1)
              assert(compareE != NULL && ``compareE() must be defined.'');

101         auto F = [&](const long start, const long stop, const int thid){
              if (start==stop) return;
              int cnt_tmp = 0;
              int ite_max_tmp = ite_max;
105           std::vector<T> tmpB;
              typename std::vector<T>::iterator it;
              tP[thid].clear();
              tBuf[thid].clear();
109
              it = (*this->input).begin();
              tP[thid].insert(tP[thid].begin(), it+start, it+stop);
              it = this->buffer.begin();
113           tBuf[thid].insert(tBuf[thid].begin(), it+start, it+stop);

              do{
                tmpB.clear();
117
                selection(*loopevol_tmp[thid], tBuf[thid], tmpB, env_tmp[thid]);
                evolutionP(*loopevol_tmp[thid], tmpB, env_tmp[thid], thid);
                filter(*loopevol_tmp[thid], tP[thid], tmpB, env_tmp[thid]);
121
```

```
        tBuf[thid].swap(tmpB);
        --ite_max_tmp;
        ++cnt_tmp;
        if(this->pE>1 && cnt_tmp==delta){ cnt=delta; break; }
      }while(ite_max_tmp>0 &&
             !(this->termination(tP[thid], env_tmp[thid])));
    };

    while(ite_max>0 &&
          !(this->termination(*(this->input), this->env))){
      (this->loopevol).parallel_for_idx(0, this->input->size(), 1, 0, F,
                                        this->pE);

      (*this->input).clear();
      this->buffer.clear();
      for(int i=0; i<this->pE; ++i){
        (*this->input).insert((*this->input).end(), tP[i].begin(),
                              tP[i].end());
        this->buffer.insert(this->buffer.end(), tBuf[i].begin(),
                            tBuf[i].end());
      }

      if(this->pE>1){
        if(cnt==delta){
          ite_max = ite_max - delta;
          mix(*(this->input), this->buffer);
          cnt = 0;
        }
        else
          ite_max = 0;

        for(int i=0; i<this->pE; ++i)
          if(compareE(env_tmp[i], this->env))
            this->env = env_tmp[i];
      }
      else{
        ite_max = 0;
        this->env = env_tmp[0];
      }
    }
```

```
            if(loopevol_tmp[0] != NULL)
              for(int i=0; i<this->pE; ++i) loopevol_tmp[i]->threadPause();
165
            this->loopevol.threadPause();
            for(const auto &e : copy_buffer) delete e;

169         for(const auto &e : this->buffer)
              if(copy_buffer.count(e)==0) delete e;

            copy_buffer.clear();
173         this->buffer.clear();
            return (task?this->input:NULL);
          }
        };
177   }
```

A.2 Classe Evolution

```
#include <iostream>
#include <vector>
#include <ff/node.hpp>
#include <ff/parallel_for.hpp>
#include <unordered_set>

namespace ff{
  template<typename T, typename env_t=char>
  class Evolution : public ff_node{
    public:
      typedef void (*generate_t) (std::vector<T> &, env_t &);
      typedef void (*ag_t) (ParallelForReduce<T> &, std::vector<T> &,
                            std::vector<T> &, const int,
                            const int, env_t &);
      typedef bool (*termination_t) (const std::vector<T> &pop, env_t &);
      typedef env_t envT;

    protected:
      std::vector<T> input;
      std::vector<T> buffer;
      size_t maxp, pE, size, iter = INT_MAX, id = INT_MAX;
      env_t env;

      generate_t generate;
      ag_t ag;
      termination_t termination;
      ParallelForReduce<T> loopevol;

    private:
      std::unordered_set<T> copy_buffer;

    public:
      Evolution(size_t maxp, size_t size, generate_t gen, ag_t ag,
                termination_t term, const env_t &E = env_t(),
                bool spinWait=true): maxp(maxp), pE(maxp), env(E),
                generate(gen), ag(ag), termination(term),
                loopevol(maxp, spinWait){
                  loopevol.disableScheduler(true);
                  input.resize(size);
```

```
                generate(input, env);
                for(const auto &e : input)
                  buffer.push_back(e->Clone());
                copy_buffer.insert(this->buffer.begin(),
                                   this->buffer.end());
        }

        ~Evolution(){
          for(const auto &e : input)
            delete e;

          for(const auto &e : this->buffer)
            if(copy_buffer.count(e)==0)
              delete e;
          input.clear();
          buffer.clear();
          copy_buffer.clear();
        }

        std::vector<T>& get_input() { return input; }
        std::vector<T>& get_buffer() { return buffer; }

        void setParEvolution(size_t pardegree){
          if(pardegree>maxp)
            error(``setParEvolution: pardegree too high,
                  it should be less than or equal to %ld\n'', maxp);
          else pE = pardegree;
        }

        env_t& getEnv() { return env; }

        int getIte() { return iter; };
        void setIte(int i) { iter = i; };

        int getId() { return id; };
        void setId(int i) { id = i; };

        int run_and_wait_end(){
          if(ff_node::run()<0) return -1;
          if (ff_node::wait()<0) return -1;
          return 0;
```

```
        }

83      protected:
          void* svc(void* task){

            while(iter>0 && !termination(input, env)){
87            ag(loopevol, input, buffer, id, pE, env);
              --iter;
            }
            loopevol.threadPause();
91
            return NULL;
          }
      };
95  }
```

A.3 Multi-Popolazione

```
1  #include <iostream>
   #include <vector>
   #include <ff/node.hpp>
   #include <ff/parallel_for.hpp>
5
   namespace ff{

     template<typename T, typename env_t=char>
9    class EvolutionE: public ff_node{
       public:
         typedef void (*mix_t) (std::vector<T> &);
         typedef bool (*compareE_t) (env_t e1, env_t e2);
13
       protected:
         size_t delta, ite_max = INT_MAX;
         env_t  env;
17       mix_t mix;
         compareE_t compareE = NULL;
         std::vector<T>* islands;
         ParallelForReduce<T> loopevol;
21
       public:
         EvolutionE(std::vector<T> &input, mix_t m, size_t d,
                    bool spinWait=true): mix(m), delta(d),
25                  islands(&input), loopevol(input.size(), spinWait){
                      assert(delta > 0 &&
                             ``delta must be greater than 0.'');
                      loopevol.disableScheduler(true);
29       }

         int getIte() { return ite_max; };
         void setIte(int i) { ite_max = i; };
33       void setCompareE(compareE_t e) { compareE = e; };

         env_t& getEnv() { return env; }

37       int run_and_wait_end(){
           if (ff_node::run()<0) return -1;
           if (ff_node::wait()<0) return -1;
```

```
    return 0;
  }

protected:
  void* svc(void* task){
    int size = islands->size();
    if(size>1)
      assert(compareE != NULL && ``compareE() must be defined.'');

    bool termination = false;

    auto E = [&](const long i, const int thid){
      if(size==1) (*islands)[i]->setIte(ite_max);
      else (*islands)[i]->setIte(delta);

      if((*islands)[i]->run_and_wait_end()<0)
        abort();
      if((*islands)[i]->getIte()>0) termination = true;
    };

    while(ite_max>0){
      loopevol.parallel_for_thid(0, size, 1, PARFOR_STATIC(0), E, size);
      if(size>1){
        if(!termination){
          ite_max = ite_max - delta;
          mix(*islands);
        }
        else ite_max = 0;

        for(int i=0; i<size; ++i)
          if(compareE((*islands)[i]->getEnv(), env))
            env = (*islands)[i]->getEnv();
      }
      else{
        ite_max = 0;
        env = (*islands)[0]->getEnv();
      }
    }
    loopevol.threadPause();

    return NULL;
```

```
81          }
      };
}
```