Scuola Superiore Sant'Anna

Classe Accademica di Scienze Sperimentali

Settore di Ingegneria

Tesi di Licenza

Algoritmo genetico parallelo sul Single-Chip Cloud Computer

Relatore

. . . . . .

Prof. *Giuseppe Lipari*

Scuola Superiore Sant'Anna

Il candidato

. . . . . .

Vincenzo Maffione

Tutor

. . . . . .

Prof. *Paolo Ancilotti*

Scuola Superiore Sant'Anna

Anno Accademico 2010-2011

Sommario

Questo lavoro si pone come obiettivi:

1. lo studio di un'architettura multicore basata sullo scambio di messaggi sviluppata da Intel, il Single Chip Cloud Computer.

2. lo sviluppo su tale architettura, utilizzando il linguaggio C++, di un framework di ottimizzazione mono-obiettivo non vincolata basato su un *algoritmo genetico parallelo*, ai fini di sperimentare il paradigma di programmazione a scambio di messaggi

Le simulazioni effettuate evidenziano un graduale miglioramento dei risultati dell'ottimizzazione all'aumentare delle unità di calcolo impiegate.

Indice

Elenco delle figure

Capitolo 1

Introduzione ed obiettivi

1.1 Nuove tendenze architetturali

La spinta verso il continuo aumento delle prestazioni dei processori che da sempre caratterizza la storia dei calcolatori elettronici ha portato negli ultimi anni allo sviluppo di nuovi modelli architetturali, basati sull'impiego di più unità di elaborazione (CPU). Ne costituiscono un esempio i comuni PC dotati di processore multi-core o le architetture SMP (*Symmetric Multi-Processing*), che sono al giorno d'oggi facilmente accessibili sul mercato.

La necessità di dirigersi verso architetture multielaboratore è dettata da limiti e difficoltà di carattere tecnologico, che rendono molto complicato oppure sconveniente l'aumento della frequenza di lavoro dei chip al fine di incrementare le prestazioni e/o migliorare dell'efficienza energetica.
Aumentare la frequenza di lavoro significa infatti incrementare la potenza termica generata dal chip durante il suo normale funzionamento, con conseguente necessità di dissipare il calore nell'ambiente. Il problema può essere affrontato efficacemente riducendo progressivamente le dimensioni dei transistor di cui i chip sono costituiti. Si parla in questo senso di ottimizzare il *processo produttivo*.

Sfortunatamente non è possibile ridurre indefinitivamente le dimensioni dei transistor, a causa del progressivo intensificarsi di effetti fisici indesiderati.
L'alternativa all'aumento della frequenza di lavoro (*frequency scaling*), adottata da tutti i produttori, è quindi lo sviluppo di processori costituiti da più CPU.

Da osservare che l'aumento delle prestazioni ed il miglioramento dell'efficienza energetica è dovuto anche allo sviluppo di microarchitetture[1] sempre più avanzate e ottimizzate. Ad esempio la tecnica dell'esecuzione fuori ordine (*out-of-order execution*) sfrutta l'*instruction-level parallelism* per raggiungere prestazioni più elevate. Un'altra tecnica è il cosiddetto *simultaneous multi-threading*, che permette di eseguire più thread in parallelo su una stessa unità di elaborazione.

Esistono vari modi con cui più CPU in uno stesso sistema di elaborazione possono comunicare tra loro, ed esistono anche diversi schemi con cui essi possono condividere risorse quali banchi di memoria RAM o memorie cache.
A titolo di esempio, i comuni processori *dual-core* sono costituiti da due unità di elaborazione che condividono la memoria centrale e tutte le risorse accessibili nel sistema. I due core possono condividere anche uno o più livelli di memoria cache.
Un processore dual-core di questo tipo è un esempio di architettura a memoria comune (*shared memory*), in quanto la memoria centrale nella quale il sistema operativo ed i programmi applicativi risiedono durante il funzionamento del sistema è condivisa tra le due unità di elaborazione.
Esistono sul mercato architetture multi-core a memoria comune con quattro, sei o otto unità di elaborazione.

Nonostante l'architettura a memoria comune sia attualmente molto utilizzata, essa si è dimostrata poco scalabile nel momento in cui il numero dei processori aumenta ulteriormente, perché tutte le unità di elaborazione devono condividere l'unità di comunicazione con la memoria centrale, il bus di memoria. Infatti, poiché il bus di memoria è utilizzabile da un core alla volta, all'aumentare del numero dei processori aumenta il tempo medio di attesa per l'accesso alla memoria.
Un utilizzo più intensivo del caching allevia questo problema, ma allo stesso tempo amplifica il problema della coerenza della cache (*cache coerency*), che deve essere gestito al fine di garantire l'integrità della memoria, e conseguentemente la correttezza delle elaborazioni effettuate.
Il costo del mantenimento della coerenza della cache, che si può ottenere ad esempio mediante l'utilizzo di tecniche di *snooping* o di *snarfing*, diventa progressiva-

[1] Per *microarchitettura* di un processore si intende lo schema logico espresso in termini di blocchi funzionali che descrive il funzionamento del processore stesso.

mente sempre più pesante all'aumentare del numero di core.

Per questi motivi, allo scopo incrementare ulteriormente il numero di unità di elaborazione incorrendo il meno possibile in problemi di scalabilità, sono stati sviluppati sistemi di elaborazione basati sull'architettura a memoria distribuita (*distribuited memory*), nella quale ogni unità di elaborazione dispone della propria memoria centrale. Non condividendo un'unica memoria centrale, le unità di elaborazione devono comunicare scambiando messaggi mediante una apposita rete di comunicazione. Per questo motivo le architetture a memoria distribuita vengono anche chiamate architetture *a scambio di messaggi* (*message passing*).
Se un'architettura a memoria distribuita viene realizzata su un singolo chip, si parla di *Network-on-Chip* (NoC).

Il Single Chip Cloud Computer (per brevità SCC) presentato da Intel nel 2010, oggetto di questo lavoro, è un esempio di Network-on-Chip.

1.2 Obiettivi della tesi

Questo lavoro si divide in due parti, che perseguono rispettivamente due obiettivi.

Il primo obiettivo è quello di studiare l'architettura hardware dell'Intel Single Chip Cloud Computer ed il relativo software di supporto per il programmatore. In modo particolare si studierà il funzionamento della libreria RCCE, disponibile al programmatore, ed il relativo paradigma di programmazione a cui bisogna attenersi. L'interesse per un processore multicore a memoria distribuita che fornisce supporto per il message passing, quale è l'SCC, nasce dalle considerazioni fatte nella sezione precedente.

Il secondo obiettivo è quello di mettere in pratica gli strumenti e le metodologie apprese nella prima parte del lavoro per sviluppare una applicazione parallela che sfrutti il più possibile le potenzialità dell'SCC. A questo fine si è scelto di implementare un framework di ottimizzazione mono-obiettivo e non vincolato basato su un *algoritmo genetico parallelo*.

Capitolo 2

Architettura dell'SCC

2.1 Introduzione all'SCC

Il Single Chip Clourd Computer (SCC) è un microprocessore sperimentale creato nei laboratori Intel, ed alla data attuale è il processore che contiene il più alto numero di core (48) conformi alla Intel Architecture (IA) mai integrati su un singolo chip.
Nel settore dei supercomputer esistono da anni sistemi costituiti da centinaia o anche migliaia di core, ma in questi ultimi le singole unità di elaborazione implementano un instruction set molto limitato, non paragonabile alla complissità della Intel Architecture. In questo senso, l'SCC rappresenta un passo in avanti nel settore, in quanto la disponibilità di un instruction set complesso permette l'utilizzo di un sistema operativo moderno completo.
L'SCC incorpora diverse tecnologie destinate ad essere impiegate su processori multi-core con un numero di core anche superiore alle 100 unità, come ad esempio la tecnologia che realizza il network-on-chip, le tecnologie avanzate di gestione dell'alimentazione o il supporto per il message passing.
La nuova architettura multicore comprende alcune innovazioni che favoriscono la scalabilità in termini di efficienza energetica: tra queste, un miglior meccanismo di comunicazione core-to-core e tecniche software che consentono di configurare dinamicamente tensione e frequenza di lavoro regolando consumi di potenza tra i 25W ed i 125W.
La piattaforma rappresenta l'ultimo risultato raggiunto dal progetto Intel *Tera-scale*

Computing Research Program. La ricerca è stata co-promossa dai laboratori Intel a Bangalore (India), i laboratori Intel a Braunschweig (Germania) e ricercatori dei laboratori Intel negli Stati Uniti.
Il nome Single-chip Cloud Computer riflette il fatto che l'architettura ricorda molto da vicino un cluster scalabile di computer (una *nuvola* di computer), con la peculiarità di essere integrato su un singolo wafer di silicio.
A grandi linee, il sistema comprende:

1. 24 *tiles* (ossia mattonelle) che compongono il cluster: ogni tiles contiene due core IA
2. Una rete mesh composta da 24 router con picco di banda sul taglio pari a 256 GB/s
3. 4 DDR3 memory controllers integrati
4. supporto hardware per lo scambio dei messaggi

Ogni core può caricare il proprio sistema operativo, comprensivo di stack di rete, e agire esattamente come agirebbe il nodo di una tradizionale rete di calcolatori a commutazione di pacchetto.
Uno degli aspetti più importanti della rete di comunicazione dell'SCC è che fornisce supporto a dei modelli di programmazione basati sullo scambio di messaggi che si sono dimostrati essere adatti per cluster composti da migliaia di processori.
Benchè ogni core abbia due livelli di cache, non viene fornito alcun supporto per la cache coherency, in modo da rendere più semplice l'architettura, ridurre i consumi, ed incoraggiare la ricerca sul calcolo distribuito on-chip basato su sistemi a memoria distribuita.

Un altro aspetto caratterizzante l'architettura è l'attenzione alla gestione dei consumi. Le applicazioni hanno la possibilità di attivare/disattivare ogni core e regolare le performance in ogni istante, rendendo possibile la minimizzazione dei consumi, pur garantendo le prestazioni necessarie.
É possibile regolare regolare la tensione di alimentazione e la frequenza di lavoro sia della mesh che di alcuni sottoinsiemi di core. Ogni tile ha la propria frequenza

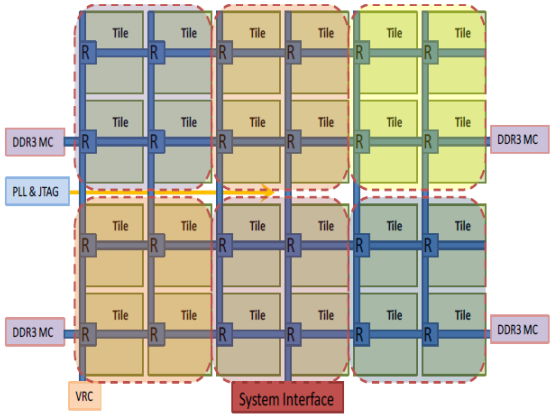

Figura 2.1: Schema a blocchi dell'SCC

di lavoro, e gruppi di 4 tiles possono funzionare ognuno con la propria tensione di alimentazione.

2.2 Architettura

Uno schema a blocchi dell'SCC è riportato in figura 2.1.
I 48 core IA presenti nell'SCC sono microprocessori della famiglia P54C, uscita sul mercato nel 1994. Il processore P54C è relativamente semplice (rispetto ai core attuali): è dotato di due pipeline a 5 stadi, istruzioni floating-point, controllore avanzato delle interruzioni APIC, ma non ha un motore di esecuzione fuori ordine e non implementa le estensioni MMX.
I 48 core sono piazzati in una formazione a mattonelle (*tiles*), con due core per ogni mattonella.
Le *tiles* sono disposte in modo tale da formare una griglia bidimensionale (2D mesh) di dimensioni 6x4. Le comunicazioni avvengono per mezzo di appositi router presenti all'interno di ogni tile.
I quattro memory controller integrati sul chip permettono di indirizzare fino ad un massimo di 64GB di memoria. Più precisamente, ogni memory controller può indirizzare 2 banchi di memoria DIMM[1], ognuno conentente 8GB di memoria.

[1] *dual in-line memory module*

Sono inoltre presenti, come supporto hardware allo scambio di messaggi, dei banchi di SRAM locali ad ogni tile. Questo nuovo tipo di memoria, unito ad una nuova istruzione per la gestione della cache L1 interna al microprocessore P54C, facilitano la gestione della memoria dell'SCC.
Il *voltage regulator controller* (VRC) permette a qualsiasi core o l'interfaccia di sistema (SIF) di regolare la tensione di alimentazione in una qualsiasi delle regioni tratteggiate mostrate in figura 2.1, oltre alla tensione di alimentazione dell'intera griglia di router. Questo metodo consente alle applciazioni un controllo totale del consumo di potenza.
L'interfaccia esterna di sistema, o SIF (*system interface controller*) permette di mettere in comunicazione un router di bordo nella rete mesh con l'esterno (precisamente, con un FPGA la cui funzione è descritta qui di seguito).

La piattaforma di sviluppo su cui il processore SCC è montato è costituita inoltre da un FPGA ed un *board management microcontroller* (BMC).
L'FPGA fa le veci di un *chipset*, fornendo tra le altre cose interfacce ethernet, interfacce SATA, un controllore globale delle interruzioni e registri contatori su cui è possibile fare incrementi in maniera atomica.
Da un punto di vista operativo l'SCC è controllato da un *board management microcontroller* (BMC) che inizializza ed arresta le funzionalità critiche della piattaforma, e che si interfaccia, tramite PCI-Express, ad un comune PC che agisce da *Management Console* (MCPC).
Tramite l'MCPC è dunque possibile caricare su ogni core un sistema operativo basato sul kernel Linux e caricare sui vari core una applicazione.
La piattaforma di ricerca è rappresentata in figura 2.2.

2.3 Descrizione del tile

Ogni tile è composto dai seguenti elementi:

1. Due IA core PC54C, con associati una cache L1 interna ed una cache L2 esterna

2. Un crossbar router a 5 porte, che interfaccia il tile con la mesh e quindi con gli altri core

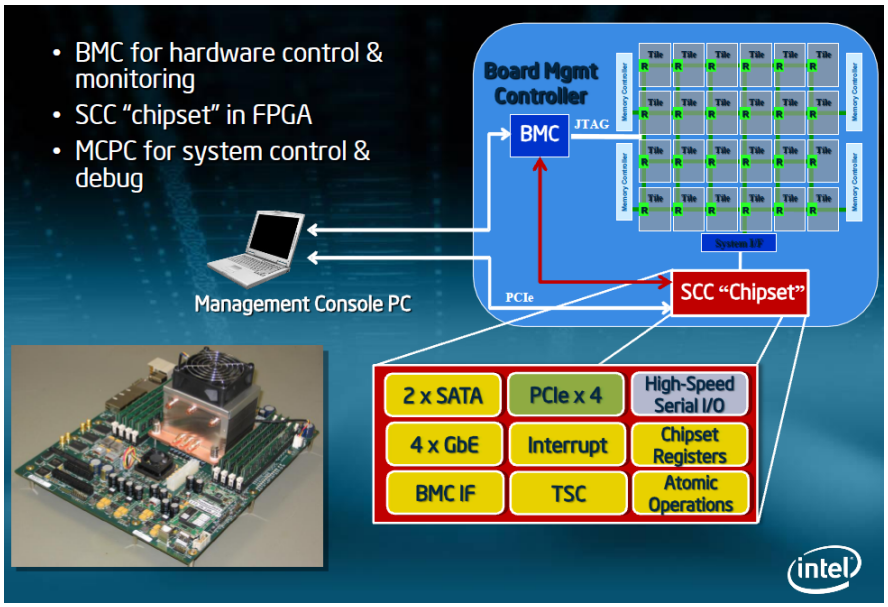

Figura 2.2: Piattaforma di sviluppo dell'SCC

3. Un generatore di traffico (TG) per testare la mesh, non accessibile via software,

4. Una *mes interface unit* (MIU) che gestisce tutti gli accessi alla memoria e le operazioni di scambio dei messaggi. Si noti che la MIU è l'unica interfaccia che i due core interni al tile hanno con il router, e quindi con l'esterno. Allo stesso modo, i router costituiscono l'unica interfaccia tra i vari tiles.

5. Delle *memory lookup table* (LUT) che permettono la traduzione degli indirizzi fisici di un core in indirizzi di sistema (ossia indirizzi globali nell'SCC)

6. Un message-passing buffer, che supporta lo scambio di messaggi.

7. Ciruciterie per la generazione del clock e circuiterie di sincronizzazione per porre in comunicazione porzioni di circuiti che lavorano a frequenze differenti (GCU e CCF)

Lo schema a blocchi di un tile è riportato in figura 2.3.

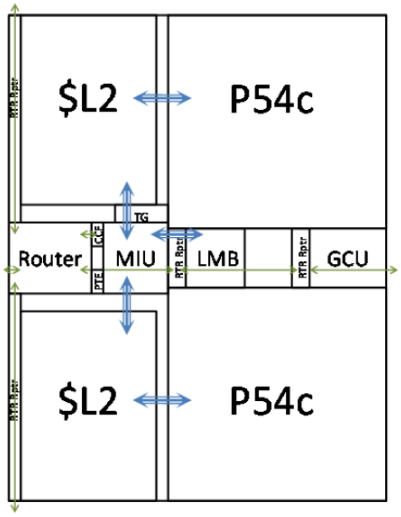

Figura 2.3: Schema a blocchi di un tile

2.4 Descrizione funzionale dei componenti dell'SCC

2.4.1 Message passing buffer (MPB)

In aggiunta alle tradizionali memorie cache, ogni tile è fornito di un buffer locale, il *message passing buffer*, capace di operazioni veloci di lettura/scrittura. Questo buffer, grande 16KB, fornisce l'equivalente di 512 linee di memoria cache da 32 bytes. Tutti i core e la SIF possono scrivere o leggere in uno qualasiasi degli MPB. Sebbene l'MPB possa essere usato in qualsiasi modo, il suo principale utilizzo riguarda il meccanismo per lo scambio di messaggi, come verrà illustrato nel seguito.

2.4.2 Core P54C

Il core, come già accennato, è un Pentium PC54. Il design originale è stato tuttavia leggermente alterato per aumentare fino a 16KB le dimensioni della data cache e della instruction cache di primo livello (la singola linea ha dimensione pari a 32 bytes). Le cache sono di tipo 4-way set associative con politica di rimpiazzo pseudo-LRU. In più, l'originale interfaccia tra bus di sistema e controllore della cache (M-unit) è stata integrata all'interno del core.

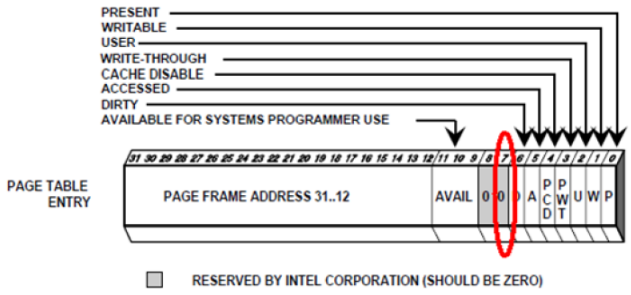

Figura 2.4: Entrata della tabella delle pagine del P54C

L'instruction set del P54C è stato esteso con una nuova istruzione (`CL1INVMB`) ed un nuovo tipo di memoria (MPBT, ossia *message passing buffer type*) allo scopo di facilitare il meccanismo di scambio dei messaggi.
L'istruzione `CL1INVMB` è stata aggiunta per invalidare tutte le linee della cache L1 che contengono dati di tipo MPBT. A questo scopo la cache L1 è stata estesa aggiungendo ad ogni linea un bit che specifica se la stessa contiene dati di tipo MPBT. Questo bit viene settato quando la linea di cache viene caricata con un dato di tipo MPBT. Per stabilire se un dato sia o meno di tipo MPBT, bisogna considerare l'entrata della tabella della pagine relativamente alla pagina che contiene il dato. Questa entrata, oltre che in memoria virtuale, è presente anche nel TLB (*Translation lookaside buffers*). Il bit MPBT viene invece resettato quando il dato viene espulso dalla cache oppure quando vengono eseguite le istruzioni `CL1INVMB`, `INVD` o `WBINVD`. Di conseguenza, un nuovo bit è stato aggiunto in ogni entrata del TLB. Questo bit, presente in posizione 7 (è uno dei bit riservati, quindi precedentemente inutilizzati), insieme al bit PCD (*page cache disable*) ed al bit PWT (*page write through*) determina il tipo di memoria della pagina in questione (e quindi di un qualsiasi dato in essa presente). Le entrate delle tabelle delle pagine devono essere impostate dal sistema operativo. Dati di questo tipo sono gestiti dalla cache L2 come non-cacheable: la M-unit del core assicura che i dati di tipo MPBT non siano mai caricati nella cache L2. In altre parole per dati di tipo MPBT la cache L2 viene esempre bypassata. Inoltre, i dati di tipo MPBT vengono trasferiti tra cache L1 e memoria condivisa (MPB oppure memoria condivisa off-chip) con la granularità di 32 bytes. Ciò non vale per dati non di tipo MPBT.

La struttura dell'entrata della tabella delle pagine è riportata in figura 2.4; Il tipo MPBT viene utilizzato principalmente per quella porzione di spazio di indirizzamento fisico che corrisponde all'MPB. In generale, il sistema operativo può inizializzare la tabella delle pagine in modo dale che anche pagine fisiche di memoria off-chip siano etichettate come di tipo MPBT. Poiché tuttavia l'SCC non implementa protocolli di cache coherency (snooping, snarfing, ecc.), conviene far sì che una qualsiasi porzione di memoria condivisa tra più core sia etichettata come di tipo MPBT, in modo tale da ottenere la coerenza della cache via software, nel modo descritto di seguito. Per accelerare il trasferimento di messaggi tra i core è stato inoltre aggiunto un *write combine buffer* alla M-unit, che agisce sui dati di tipo MPBT. Il write combine buffer trattiene nella M-unit i dati aggiornati da scrivere in memoria fino a quando non si riempie una intera linea di cache, oppure fino a quando non si tenta di scrivere un'altra linea. Quando una di queste due situazioni si verifica, esso viene svuotato ed il comando di scrittura di una intera linea di cache viene inviato in memoria.

Quando la `CL1INVMB` viene seguita, essa pone a zero tutti i bit MPBT della cache in un ciclo di clock. Ogni linea di tipo MPBT viene invalidata. Se una linea da invalidare era stata modificata, il dato modificato non va ad aggiornare la memoria, ma viene perso. É responsabilità del software assicurare che i dati di tipo MPBT presenti in cache non vadano persi. Nel peggiore dei casi, si può eseguire l'istruzione `WBINVD`, la quale, prima di invalidare l'intera cache L1, si preoccupa di effettuare l'operazione di *write back*, ossia la propagazione verso l'esterno dei dati aggiornati (assicura la cache coerency).

Per quanto riguarda il message passing, l'istruzione `CL1INVMB` può essere usata per mantenere coerente la cache L1 (visto e considerato che la cache L2 viene comunque bypassata) con il contenuto dalla memoria condivisa (MPB oppure memoria condivisa off-chip). É sufficiente infatti eseguire la `CL1INVMB` prima di leggere dalla memoria condivisa per causare un read miss e dunque essere sicuri di leggere il contenuto attuale dell'MPB e non una copia non aggiornata dei dati in memoria condivisa (*stale data*). Analogamente, è sufficiente eseguire la `CL1INVMB` prima di scrivere nell'MPB per causare un write miss e forzare il P54C a emettere effettivamente una richiesta di scrittura verso l'esterno. Infatti la politica in caso di write

miss della cache L1 è *no write allocate*. In altre parole, solo le read miss comportano il caricamento del dato in cache. Se non eseguissimo la `CL1INVMB`, il dato da modificare con l'operazione di scrittura potrebbere essere già presente in cache, ed in virtù della politica write-back la scrittura aggiornerebbe solamente la cache, non arrivando in memoria condivisa.

2.4.3 Cache L2

Ogni core ha la propria cache di secondo livello (256KB) con il controllore associato. Durante un *cache miss* di quest'ultima, il controllore invia l'indirizzo richiesto alla MIU, per la decodifica dell'indirizzo (si veda il paragrafo dedicato alle lookup tables, o LUT) ed il successivo fetching in memoria. Ogni core può avere in ogni istante solamente una richiesta pendente verso la memoria. Sarà quindi costretto ad attendere a causa di una *read miss* fino a quando i dati non arrivano dalla memoria. In seguito ad una *write miss*, invece, il core può continuare ad operare a patto che non si verifichi un altra write miss oppure una read miss. All'arrivo del dato richiesto, il core può tornare ad operare normalmente.
Nonostante queste limitazioni, dovute all'impiego del P54C, il resto dell'hardware (la mesh ed il sistema della memoria) è capace di supportare più richieste pendenti.
La cache L2 è di tipo 4-way set associative con politica di rimpiazzo pseudo-LRU. Può essere usata solo in modalità *write-back* e non supporta la modalità *write-allocate*.
Nell'SCC, le istruzioni `INVD` e `WBINVD` eseguite da un core non hanno alcun effetto sulla cache L2 del core stesso.

2.4.4 Tabelle di lookup per la traduzione degli indirizzi

Ogni core dispone di una *lookup table* (LUT) che è formata da un insieme di registri di configurazione (che sono a loro volta mappati in memoria tramite loro stessi). Questa tabella mappa gli indirizzi fisici del core nello spazio di indirizzi di sistema dell'SCC (indirizzi globali). Una LUT contiene 256 entrate, una per ogni porzione dello spazio di indirizzamento fisico del processore, che è di 4GB (gli indirizzi del core sono su 32 bit). Di conseguenza ogni entrata della LUT mappa 16MB dello spazio fisico di indirizzamento del processore. Ogni entrata della LUT può puntare ad una qualsiasi locazione di memoria: le memoria privata del core (la

RAM off-chip accedibile tramite i 4 memory controller), il message passing buffer, i registri di configurazione del core, l'interfaccia di sistema (SIF) o il controllore dell'alimentazione. Il LUT può essere programmato dalla Management Console per mezzo di scritture che attraverso la SIF. Normalmente, le sue entrate sono inizializzate con una configurazione opportuna durante il processo di bootstrap del sistema. Dopo il boot, una generica LUT può essere dinamicamente modificata da qualsiasi core che sia capace di indirizzarla attraverso la propria LUT. Quando si verifica una chache miss della cache L2, la MIU consulta la LUT per determinare dove la richiesta di memoria debba essere spedita.

2.4.5 Mesh Interface Unit (MIU)

La Mesh Interface Unit (MIU) contiene i seguenti blocchi funzionali:

1. Pacchettizzatore e depacchettizzatore
2. Interprete dei comandi provenienti dall'esterno, decodificatore degli indirizzi
3. Alcuni registri locali di configurazione
4. Logica per il controllo di flusso a livello di collegamento (livello data link)
5. Un arbitro per coordinare richieste diverse provenienti da agenti diversi

Il pacchettizzatore/depacchettizzatore traduce i dati da/verso gli agenti alla/dalla mesh.
Più in dettaglio, al verificarsi di una cache miss L2, la MIU decodifica l'indirizzo del dato richiesto, utilizzando la LUT per tradurre l'indirizzo locale del core in un indirizzo di sistema. Dunque inserisce la richiesta in una coda appropriata. Ci sono tre code di richieste:

1. Richieste per l'accesso al router, e quindi alla memoria off-chip.
2. Accesso al message passing buffer
3. Accesso ai registri locali di configurazione

Per quanto riguarda il traffico proveniente dal router, la MIU è responsabile di dirottare i dati verso la destinazione locale appropriata. Il controllo di flusso a livello data link è gestito attraverso un protocollo credit-based.

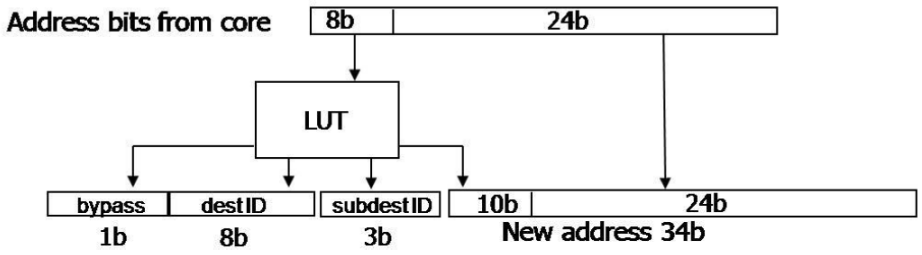

Figura 2.5: Schema di traduzione degli indirizzi

L'arbitro opera secondo una politica round robin.
Il core lavora con indirizzi fisici a 32-bit. Gli 8 bit più significativi sono usati direttamente come indice nella LUT in concomitanza di una cache miss. Il LUT emette in uscita 22 bit: 10 bit di estensione per l'indirizzo di sistema, 8 bit di `tileID`, 3 bit di `sub-destinationID` e un bit di *bypass*. Il campo `tileID` è un identificatore che specifica il tile destinazione della richiesta di accesso in memoria effettuata dal core. Il campo `sub-destinationID`, che va interpretato in base al valore del campo `tileID`, specifica su quale porta del router destinatario il pacchetto inviato dovrà essere rediretto e/o qual è il dispositivo destinatario: potrebbe trattarsi di un MPB, di un registro di configurazione, di un memory controller, del VRC (voltage regulator controller) oppure dell'interfaccia di sistema (SIF). L'indirizzo di sistema è dunque a 46-bit: il bit di bypass + 3 bit di `sub-destinationID` + 8 bit di `tileID` + 10 bit di estensione dell'indirizzo + i 24 bit meno significativi dell'indirizzo a 32-bit del core.
Il `sub-destinationID` definisce la porta dalla quale il pacchetto lascia il router (porta est, ovest, nord, sud).
Il tile specificato dal campo `tileID` (tile destinatario) riceve il pachetto inviato dal router e utilizzando il campo `sub-destinationID` (inviato anch'esso nel pacchetto) decide quale destinazione prendere. Il bit di bypass specifica accessi locali all'MPB.
I 34 bit meno significativi dell'indirizzo di sistema tradotto vengono spediti al tile destinatario.

Il campo `tileID` è codificato in maniera tale da permettere ai router di effettuare il routing dei pacchetti in modo corretto.

2.4.6 Registri di configurazione

I registri di configurazione presenti in ciascun tile forniscono alle applicazioni la possibilità di controllare i modi di operazione delle varie unità hardware presenti nel tile stesso. Questi registri controllano l'abilitazione del reset locale, le impostazioni del clock locale, la configurazione iniziale dei core, la gestione delle interruzioni e la configurazione della cache L2. Ogni core, cache controller della L2 o unità di gestione del clock ha un registro di configurazione dedicato che è accessibile in scrittura da qualsiasi altro core e dalla SIF. In più sono presenti dei *test-and-set register* per rendere possibili i protocolli di comunicazione (ad esempio un protocollo basato sullo scambio di messaggi) in ambito multi-processore. Più in dettaglio, ogni core possiede il proprio test-and-set register. Il valore iniziale di tali registri è 1. Un core acquisice il lock leggendo un test-and-set register ed ottenendo uno. Qualsiasi core (non necessariamente quello che ha acquisito il lock) può rilasciare il lock scrivendo un qualsiasi valore nel test-and-set register. Per convenzione, all'atto del rilascio viene scritto il valore 0.
Al reset, i registri di configurazione sono settati con dei valori ben noti e sicuri. Per modificare i registri di configurazione, è necessario seguire il paradigma read-modify-write sull'intero registro a 32 bit, per essere sicuri che l'operazione avvenga correttamente (non sono supportati accessi a porzioni di bit inferiori a 32-bit).
Il registro di configurazione di un bit contiene anche i due bit di interrupt: il bit `INTR` che segnala le interruzioni mascherabili ed il bit `NMI` che segnala le interruzioni non mascherabili. I due bit del registro sono connessi direttamente ai rispettivi pin di interrupt del core.

2.4.7 Voltage regulator controller

Il *Voltage regulator controller* (VRC) è gestito attraverso delle locazioni di memoria di sistema accessibili da un qualunque core abbia mappato tali locazioni nella propria LUT.
Una scrittura in queste locazioni si traduce in un comando per il VRC, che viene spedito sulla mesh ed eseguito. Il VRC accetta il comando, modifica la tensione

di alimentazione, ed infine invia un riscontro al tile che ha generato il comando, cosicché quest'ultimo possa essere sicuro che il comando sia stato eseguito con successo.
In fase di inizializzazione del sistema, il VRC deve ricevere i comandi per fornire l'alimentazione ai tiles, in modo tale che questi possano essere resettati. Successivamente, il VRC può ricevere richieste addizionali: togliere l'alimentazione ad un sottoinsieme di tiles, passare in modalità risparmio energetico, passare in modalità alte prestazioni, etc.).

2.4.8 Rete mesh

La rete mesh di cui l'SCC è dotato, è formata da 24 router che commutano pacchetti in una configurazione 6x4 (conforme alla formazione dei tiles). La rete è dotata di una sua sorgente di alimentazione e di un suo generatore di clock, separati da quelli del resto del chip. Ciò permette di trovare un compromesso ottimale tra consumi e prestazioni.
Il router utilizzato (RXB) è un router di nuova generazione pensato per le future strutture mesh bidimensionali da utilizzarsi per processori multicore analoghi all'SCC. Le sue caratteristiche sono le seguenti:

1. Larghezza dei collegamenti: 16 bit per la linea dati + 2 bit per segnali di controllo
2. Frequenza: 2 Ghz
3. Latenza: 4 cicli, includendo l'attraversamento del link
4. Diverse classi di messaggi: una classe di messaggio per le richieste ed una classe di messaggi per e risposte
5. Più canali virtuali (VC): 1 VC riservato per ogni classe di messaggi + 6 VC a disposizione per un totale di 8 VC
6. Gestione dinamica dell'alimentazione: modalità di sleep, controllo della tensione, clock gating[2], ecc..

[2]Il *clock gating* è una tecnica di progetto di circuiti digitali in base alla quale si cerca di fornire il clock solo a quelle porzioni di circuito che in un dato istante stanno effettivamente lavorando, riducendo così i consumi globali medi

Gli agenti della mesh commuicano tra di loro tramite scambio di pacchetti. Un pacchetto corrisponde di uno o più *flit* (fino ad un massimo di tre flit). Un pacchetto dati è composto da tre flit, con l'header nel primo flit, il corpo nel secondo e la coda nel terzo. I flit di controllo (che non sono pacchetti contenenti dati utili) sono utilizzati per scambiare informazioni di controllo quali i crediti (utilizzati nel protocollo di controllo di flusso).
Il controllo dell'errore viene effettuato end-to-end, principalemente attraverso l'uso di diversi bit di parità, applicati alle informazioni di instradamento, alle informazioni di comando ed ai dati.
I percorsi seguiti dai pacchetti vengono instradati secondo una semplice politica x-y: un pacchetto si sposta prima orizzontalmente fino all'ascissa di destinazione e successivamente si muove verticalmente.

2.5 Gestione dinamica della frequenza di funzionamento e della tensione di alimentazione

I core dell'SCC sono divisi in sei *voltage island* (o *voltage domain*), ognuna contenente una matrice 2x2 di tiles. Ogni isola ha dunque in totale di otto P54C core. Ciascuna isola possiede una propria sorgente di alimentazione. Il clock è invece fornito ad un livello più piccolo di granularità, in modo che ogni tile possa operare ad una sua propria frequenza. Le isole di tensione e di frequenza rendono possibile lo spegnimento o la riduzione dei consumi (e quindi delle prestazioni) di sottoinsiemi dell'SCC, in modo da minimizzare il consumo di energia. La regolazione può avvenire sotto il controllo dell'applicazione che può impostare il livello di prestazione voluto in specifici gruppi di tiles.

Sono dunque possibili diversi paradigmi per il controllo delle performance. Ad esempio:

- un core controlla tutti gli altri
- ogni core controlla sé stesso
- ogni core controlloa il quadrante in cui si trova

Come già accennato, la mesh ha un proprio generatore di clock e una propria sorgente di alimentazione, condivise da tutti i router nella griglia. Per questo motivo, qualsiasi comunicazione tra un router e l'hardware all'interno del tile (o altre periferiche connesse all'SCC) richiedono dell'hardware di raccordo (level shifter e asynchronous clock transition): queste ultime funzionalità sono fornite dall'unità CCF (clock crossing FIFO). I consumi della mesh possono dunque essere controllati indipendentemente da quelli dei core, e viceversa. Si può dunque pensare all'intera mesh come un'unico dominio di frequenza/tensione.

Capitolo 3

Programmazione con l'SCC

3.1 Introduzione

L'SCC fornisce strumenti di basso livello per supportare diversi paradigmi di programmazione. Il paradigma tipicamente utilizzato si basa su una qualche forma di message passing. Come già accennato, i messaggi vengono trasferiti nel chip utilizzando i message passing buffer (MPB). Come descritto nel capitolo precedente, l'SCC supporta diverse configurazioni della memoria off-chip (la RAM esterna accedibile tramite i quattro memory controller), esponendo i registri di configurazione delle tabelle di lookup (LUT). In ogni istante, l'applicazione può modificare il modo in cui gli indirizzi fisici di un dato core vengono tradotti (mappati) in indirizzi di sistema.
Gli indirizzi fisici di ogni core possono essere mappati sugli indirizzi di sistema in modo tale da far sì che tutta, parte, o nessuna parte del loro spazio di indirizzamento si riferisca ad una porzione di memoria condivisa tra determinati insiemi di core.
La suddivisione della memoria off-chip tra memoria privata a ciascun core e memoria condivisa tra gruppi di core è programmabile in modo dinamico, in modo da rendere il sistema flessibile nel partizionare compiti tra i core. La memoria condivisa può essere usata per scambiare dati fra i core, o per memorizzare dati (ad esempio la si potrebbe utilizzare per implementare un database in-memory). Può essere convieniente disabilitare il caching per la porzione di memoria condivisa, oppure gestire la coerenza della cache via software.
Tutti gli accessi di I/O sono rediretti sull'interfaccia di sistema (SIF) e dunque

all'FPGA off-chip.

Come accennato nel capitolo precedente, le interruzioni vengono segnalate ad un determinato core settando e resettando un bit appropriato nel registro di configurazione di quel core. Il software può generare interruzioni non mascherabili, interruzioni mascherabili oppure interruzioni per la gestione del sistema (*systema management interrupt*). Il modo in cui un core processa le interruzioni è configurato nella Local Vector Table (LVT) del Local APIC di cui il P54C è dotato.
L'SCC dispone di diversi metodi di reset della logica dei core, a vari livelli di granularità: a titolo di accenno, esiste l'*hard reset*, il *soft reset*, il *direct single core reset* ed il *direct single L2 cache reset*.

In fase di inizializzazione della piattaforma su cui è montato l'SCC, su ogni core viene caricato un sistema operativo basato sul kernel di Linux. Ogni core dispone della sua copia del sistema operativo, presente nella sua memoria privata. A livello operativo, i programmi applicativi con cui si vogliono sperimentare le potenzialità dell'SCC possono essere caricati sfruttando lo stack di rete attivo su tutti i core. Attraverso l'interfaccia di sistema (SIF) vengono stabilite delle connessioni ssh tra l'MCPC ed i core, in modo da poter lanciare le applicazioni semplicemente eseguendo un comando remoto dall'MCPC.

3.2 La libreria RCCE

Per utilizzare l'SCC allo scopo di effettuare ricerche sulle architetture a scambio di messaggi, è stata sviluppata la libreria RCCE, una libreria leggera e di dimensioni contenute che implementa le funzionalità minimali per consentire lo scambio di messaggi tra i core. La libreria utilizza gli MPB presenti nei tiles (si ricordi che si tratta di memoria on-chip) per trasferire dati tra core differenti.
L'MBP è un'area di memoria condivisa, e dunque in teoria potrebbe essere utilizzata in un modo qualsiasi dalle applicazioni. Poiché tuttavia ogni MPB è grande solo 16KB[1], è conveniente utilizzarlo unicamente come buffer per lo scambio di mes-

[1] si consideri inoltre che i 16KB sono dedicati complessivamente ad un tile, e quindi andrebbero suddivisi tra i due core

saggi. La dimensione dell'MPB è infatti troppo piccola perché esso possa essere utilizzato per memorizzare le strutture dati di un'applicazione di medie dimensioni. RCCE è una libreria di basso livello, che era inizialmente stata concepita per lavorare sull'SCC senza il supporto del sistema operativo (si parla di *bare metal mode*). Di conseguenza, essa è stata sviluppata senza fare affidamento ai tipici servizi che un sistema operativo offre.
Solo successivamente è stato possibile effettuare il porting di Linux sull'SCC, e da quel momento in poi si è in generale preferito lavorare con Linux.

Per questi motivi, le scelte di progetto effettuate per RCCE hanno comportato una notevole riduzione degli overhead. Innanzitutto, senza sistema operativo non esiste il concetto di unità di schedulazione, e dunque RCCE è pensata per eseguire una singola applicazione, senza necessità di richiedere servizi al sistema operativo portando il processore in modalità sistema. Ciò non rende possibile effettuare comunicazioni asincrone, poichè un'implementazione di queste ultime richiederebbe l'utilizzo di più thread. Di conseguenza le operazioni di invio e ricezione sono necessariamente bloccanti. Se da un lato ciò costituisce una limitazione, dall'altro semplifica enormemente la gestione dei messaggi, dal momento che in ogni istante ci può essere solamente una comunicazione in attesa di completamento tra ogni coppia di core. Ciò comporta che:

- non ci sono code di messaggi
- sono sufficienti due variabili di sincronizzazione per ogni coppia di core comunicanti
- l'MPB può essere interamente dedicato al payload della comunicazione (al netto dello spazio necessario per le variabili di sincronizzazione)

Il modello di programmazione fornito da RCCE è molto semplice: si basa sul paradigma *Single Program Multiple Data* (SPMD). Di conseguenza, lo stesso eseguibile viene caricato su tutti i core. Nel codice sorgente, i costrutti condizionali possono essere utilizzati per far eseguire parti di codice solo a determinate unità di esecuzione (ad ogni unità di esecuzione è associato un identificatore, utilizzabile nel codice sorgente). In questo modo, pur utilizzando un solo programma (*Single Program*), è possibile assegnare compiti diversi ad unità di esecuzione diverse, che in generale lavoreranno su dati differenti (*Multiple Data*).

Ogni core esegue sempre e solo un'unica unità di schedulazione (o unita di esecuzione). Un'applicazione è costituita da un'insieme di unità di esecuzione che cooperano. Da un punto di vista logico, le unità di esecuzione iniziano ad eseguire il codice applicativo nello stesso istante e terminano nello stesso istante. In altre parole, l'inizio e la fine sono punti di sincronizzazione (barriere).
La disponibilità dei servizi del sistema operativo consentirebbe di superare tutte queste limitazioni, ma allo stesso tempo introdurrebbe overhead temporale e spaziale[2].

RCCE spedisce un messaggio da un core all'altro trasferendo dati dalla data cache L1 del core sorgente alla data cache L1 del core destinatario, utilizzando l'MPB del core sorgente come memoria tampone. Di conseguenza il trasferimento vero e proprio non fa utilizzo della memoria off-chip: questo è un grande vantaggio, perchè il trasferimento risulta molto efficiente. Ovviamente in questa analisi non si tiene conto del costo necessario per trasferire eventualmente il messaggio ricevuto dalla cache L1 ai livelli più bassi della gerarchia (cache L2 e memoria privata off-chip). In ogni caso, essendo la data cache L1 grande 16KB, per messaggi sufficientemente piccoli (ad esempio fino a 4KB) non dovrebbe essere necessario ricorrere troppo alla memoria off-chip, se le elaborazioni da effettuare non sono troppo costose in termini di memoria.
I 16 KB di ogni MPB vengono equamente suddivisi tra i due core su ogni tile. Nel seguito, per semplicità, per *MPB di un core* si intenderà la porzione di 8KB di MPB del tile a cui il core appartiene dedicata al core stesso. Per semplificare la gestione della comunicazione, si adotta un modello simmetrico di allocazione dell'MPB. In questo modello ogni operazione di allocazione di memoria sull'MPB (di una variabile condivisa) è eseguita da tutti i core. Quando si vuole allocare, da un punto di vista logico, una variabile condivisa in un determinato MPB, una copia di tale variabile viene allocata da ogni core all'interno del proprio MPB utilizzando lo stesso ed identico offset locale per tutti gli MPB. Anche se lo spazio allocato per la variabile verrà effettivamente utilizzato solo in un MPB, tutti gli altri core allocano lo stesso spazio nella stessa posizione relativa nell'MPB, senza però utilizzarlo.
Di conseguenza, tutti i core hanno una visione identica della posizione nell'MPB di tutte le variabili condivise. Ogni core ha tuttavia mappato nel proprio spazio di indirizzi fisici l'MPB ad un indirizzo diverso, che viene calcolato dall'RCCE

[2] si ricordi che le dimensioni dell'MPB sono critiche

in fase di inizializzazione. Un core `C1` che necessiti di accedere ad una variabile condivisa logicamente allocata nell'MPB di un altro core `C2` può farlo utilizzando l'indirizzo della copia di tale variabile allocata nel proprio MPB, traslato di una quantità pari alla differenza tra l'indirizzo fisico dell'MPB del core `C2` e l'indirizzo fisico del proprio MPB[3]. Questo metodo di calcolo degli indirizzi costituisce il vantaggio del modello di allocazione simmetrica.

Questo modello di allocazione può dunque apparire molto restrittivo, ma in pratica non costituisce un problema, considerando che l'MPB viene usato solamente come buffer di comunicazione. Più precisamente nell'MPB vengono memorizzati due tipi di dati: il messaggio da inviare (che non ha nessun overhead quali header, checksum, ecc..) e flag di sincronizzazione. I flag di sincronizzazione sono variabili booleane che servono a coordinare le operazioni di invio e ricezione. Dal momento che tutti i core in una applicazione che utilizza RCCE normalmente partecipane nel message passing, è effettivamente necessario che tutti i core allochino il payload dei messaggi ed i flag.

RCCE fornisce al programmatore due tipi di interfacce. L'interfaccia *non-gory* è di più alto livello e fornisce le funzioni `RCCE_send()` e `RCCE_recv()` che vanno chiamate rispettivamente dal trasmettitore e dal ricevitore e che complessivamente trasferiscono un messaggio di dimensione arbitraria dallo spazio di memoria (che non sia mappato nell'MPB) del core trasmettente allo spazio di memoria (che non sia mappato nell'MPB) del core ricevente, senza esporre al programmatore i flag di sincronizzazione. Queste due funzioni sono implementate attraverso due funzioni analoghe di più basso livello, che costituiscono l'altra interfaccia (l'interfaccia *gory*), ossia le funzioni `RCCE_put` e `RCCE_get` che invece trasferiscono un messaggio multiplo di 32 bytes (dimensione della linea di cache) e non più grande dello spazio disponibile nel MPB dei core[4], ossia lo spazio non utilizzato per allocare variabili condivise.
Le suddette funzioni specificano l'indirizzo del messaggio da trasferire (o dell'indirizzo in cui copiare il messaggio ricevuto), l'identificatore del core destinatario (o

[3]Naturalmente tutti gli indirizzi fisici si riferiscono al core `C2`, e vengono dunque tradotti correttamente con la sua LUT.

[4]Poichè si utilizza il modello di allocazione simmetrica gli MPB di tutti i core hanno esattamente lo stesso spazio disponibile

quello del core sorgente) ed il numero di bytes da inviare (o da ricevere).
Se l'utente vuole utilizzare l'interfaccia non-gory, RCCE alloca in ogni MPB due array di flag di sincronizzazione, l'array `sent` e l'array `ready`, aventi ciascuno tanti elementi quante sono le unità di esecuzione dell'applicazione. L'utilizzo di questo flag è specificato nella sezione successiva.

Oltre alle funzioni necessarie per effettuare la comunicazione, RCCE fornisce altre funzioni di supporto (i.e. split del dominio di comunicazione, timer e altre funzioni di utilità).

3.2.1 Implementazione delle funzioni di comunicazione non-gory

All'invocazione della `RCCE_send()`, il messaggio viene spezzato in blocchi di dimensione pari allo spazio disponibile negli MPB e viene inviato blocco per blocco tramite la `RCCE_put()`. Se l'ultimo blocco non dovesse avere una dimensione multipla di 32 bytes, esso viene completato con del padding. Prima di effettuare la copia del blocco dalla memoria del trasmettitore (e quindi, dalla sua data cache L1) all'MPB dello stesso, la `RCCE_put()` invalida le linee di cache L1 che mappano l'MPB (utilizzando la `CL1INVMB`), per quanto detto nel capitolo precedente. Successivamente, la `RCCE_send()` segnala al core ricevente di aver effettuato l'operazione di copia settando uno dei flag di sincronizzazione allocati nell'MPB del core ricevente (precisamente, l'elemento nell'array di flag `sent` relativo al core trasmettitore). Il ricevente, parallelamente a tutto ciò, avendo invocato la `RCCE_recv()` si era posto in attesa (attiva) che il flag `sent` relativo al core trasmettitore venisse settato. Quando ciò avviene, può copiare nel proprio spazio di memoria il blocco scritto dal trasmettitore nell'MPB di quest'ultimo, dopo aver invalidato a sua volta le linee della data cache L1 che mappano l'MPB (utilizzando la `CL1INVMB`), per essere sicuro di accedere effettivamente all'MPB del trasmettitore. Fatto ciò, segnala al core trasmettitore di aver terminato l'operazione ed essere pronto a ricevere nuovi dati settando uno dei flag di sincronizzazione allocati nell'MPB del core trasmettitore (precisamente l'elemento nell'array di flag `ready` relativo al core ricevente). Il trasmettitore, dopo aver scritto il blocco nel proprio MPB, si era posto in attesa (attiva) che il flag `ready` relativo al core ricevente venisse settato da quest'ultimo. Quando ciò avviene, può ripetere il procedimento inviando un altro

blocco.

3.2.2 Implementazione dei flag di sincronizzazione

Essendo i flag variabili booleane, ogni flag può essere implementato semplicemente con un bit. Tuttavia le scritture e le letture dei flag devono essere atomiche perchè il protocollo descritto nella sottosezione precedente funzioni correttamente. A causa del write combine buffer, le scritture nell'MPB devono essere effettuate scrivendo un'intera cache line per volta (con più operazioni di scrittura, ma non si deve scrivere solo una parte incompleta di linea di cache).
Di conseguenza, il modo più semplice di implementare i flag in RCCE è quello di memorizzare ogni flag in una sua linea di cache dedicata, in modo da evitare di accedere più linee cache per assicurare che il write combine buffer si svuoti.
Ogni core ha bisogno dei suoi due array di flag. Nonostante il modello di memoria simmetrico preveda in generale che ci siano copie di variabili condivise inutilizzate, nel caso particolare di questi due array tutte le copie effettivamente allocate vengono anche effettivamente utilizzate. Infatti, durante l'inizializzazione di RCCE, ogni core alloca un solo array `ready` ed un solo array `sent` (e non N coppie di array). Grazie al modello simmetrico i due array sono allocati esattamente allo stesso offset in ciascun MPB. Ogni core può dunque calcolare con facilità l'indirizzo di un particolare flag (in uno dei due array di flag) associato ad un particolare core, come specificato nella sezione 3.2.

In ogni caso, se un applicazione utilizza 48 core, sono necessari almeno 96 flag per core (ci potrebbero essere altri flag necessari per altre funzionalità di RCCE), che in totale occupano il 37.5% dell'intero MPB[5].
Per evitare questo spreco eccessivo di spazio, si può optare per un'implementazione alternativa, che impiega effettivamente un solo bit per ogni flag: in questo modo una linea di cache (32 bytes) può contenere fino a 256 flag. Aggiornare un flag richiede in questo caso la copiatura dell'intera lina di cache in cui tale flag è allocato dall'MPB in memoria privata, aggiornare il bit associato al flag, e riscrivere la linea di cache aggiornata nell'MPB. Questa operazione deve essere eseguita in mutua esclusione con gli altri core. Ciò si può ottenere sfruttando i

[5] 96*32 bytes = 3072 bytes

registri test-and-set di cui ogni core è fornito. Questa implementazione comporta un maggiore overhead, dovuto alle operazioni di acquisizione/rilascio del lock e le manipolazioni di bit all'interno delle linee di cache contenenti i flag. Si ottiene però un vantaggio importante perchè il numero di sincronizzazioni necessarie per il trasferimento di un messaggio nei due casi, a parità di dimensione del messaggio, è mediamente minore in quest'ultimo caso (essendoci nell'MPB più spazio per il payload).

Capitolo 4

Algoritmo genetico parallelo

Oggetto di questo lavoro è l'implementazione, sotto forma di framework, di un algoritmo genetico parallelo capace di sfruttare le potenzialità offerte dall'SCC. Il progetto è stato sviluppato utilizzando il linguaggio C++.
Un aspetto importante dell'applicazione, che sfrutta a pieno le caratteristiche del C++, è la sua genericità rispetto alla funzione obiettivo da ottimizzare. Mediante l'uso dei *template* C++, infatti, è possibile specificare (sempre utilizzando il linguaggio C++) una funzione obiettivo avente in ingresso un oggetto di tipo arbitrario, ed in uscita uno tra i tipi primitivi numerici del C++ (float, double, int, ecc.).

4.1 Algoritmi genetici

In questa sezione viene brevemente descritto l'algoritmo genetico sequenziale su cui si basa l'algoritmo genetico parallelo implementato nel framework.
Un algoritmo genetico è un metodo stocastico di ottimizzazione che prende spunto da alcuni meccanismi biologici che gli organismi utilizzano per *adattarsi* ai cambiamenti dell'ambiente in cui vivono, e quindi per evolversi evitando l'estinzione.

I meccanismi biologici presi in considerazione sono sostanzialmente tre:

- Mutazione: a causa di errori nella copia dei filamenti di DNA, le cellule degli organismi subiscono delle mutazioni che possono cambiare le caratteristiche dell'organismo stesso.

- Ricombinazione (*crossover*): quando gli organismi di una certa specie si riproducono in coppia i loro geni si mescolano nel nuovo organismo generato, combinandone le caratteristiche. Questo può favorire o sfavorire la perpetuazione della specie.

- Selezione: non tutti gli organismi di una certa specie riescono a riprodursi. Tipicamente riescono nell'intento solo gli individui più forti.

L'idea alla base dell'algoritmo genetico è quella di definire opportunamente ed applicare iterativamente questi tre meccanismi evolutivi (chiamati *operatori*) ad una popolazione di N_p individui appartenenti allo spazio di ricerca su cui si vuole effettuare l'ottimizzazione. Gli individui sono anche detti *cromosomi*.
Il valore della funzione obiettivo calcolata su un certo individuo è anche detta *fitness* dell'individuo.
La scelta dei tre operatori condiziona fortemente le prestazioni della ricerca.

Gli algoritmi genetici sono nati per risolvere problemi in cui il cromosoma è una stringa di bit, anche se successivamente sono stati adattati a risolvere problemi in cui il cromosoma è un vettore di numeri reali. Il cromosoma è comunque costituito da n componenti (siano esse bit oppure numeri reali) che vengono chiamate *geni*.

L'algoritmo genetico che verrà usato è così schematizzato:

1. Vengono creati gli N_p elementi costituenti la popolazione iniziale, in modo casuale oppure inizializzandoli opportunamente.

2. La popolazione corrente viene utilizzata per creare la nuova popolazione, nel modo seguente:

   (a) Viene calcolato il valore della funzione obiettivo per tutti i membri della popolazione corrente.

   (b) *Fitness scaling*: i valori calcolati al punto precedente vengono *scalati*, ossia rimappati in modo non lineare, per esigenze dell'operatore di selezione.

   (c) Alcuni membri della popolazione corrente vengono selezionati, in base al valore scalato della funzione fitness, per partecipare alla riproduzione.

(d) Gli N_e individui migliori (sempre in base alla loro fitness), detti *elite children*, vengono automaticamente promossi a membri della popolazione successiva.

(e) A partire dagli individui selezionati al punto c (i *genitori*) viene creata la nuova popolazione. I nuovi individui vengono creati per mutazione (*mutation children*) oppure per crossover (*crossover children*).

(f) La popolazione corrente viene rimpiazzata dagli individui prodotti al punto precedente e dagli *elite children*.

3. Se una delle condizioni di terminazione è soddisfatta, l'algoritmo si ferma, altrimenti si torna al punto 2.

Si analizzano ora nei dettagli le varie componenti dell'algoritmo.

4.1.1 Fitness scaling

La rimappatura delle fitness della popolazione è necessaria per evitare problemi nella successiva fase di selezione. In questa fase, infatti, la probabilità che un individuo ha di essere selezionato sarà in proporzionale al valore della funzione fitness.
Se però i valori della funzione fitness sulla popolazione corrente variano troppo fra di loro, ad esempio se c'è un piccolo gruppo di individui con fitness molto migliore degli altri, si creano degli squilibri, in quanto si riproducono quasi esclusivamente gli individui di quel gruppo. L'informazione genetica contenuta negli altri individui viene quasi sicuramente persa, portando l'algoritmo a convergere prematuramente. Questo comporta una ricerca molto limitata nello spazio delle soluzioni, con conseguenti scarsi risultati finali.

D'altro canto, se i valori di fitness variano troppo poco all'interno popolazione, tutti gli individui tenderanno a partecipare alla riproduzione, e l'algoritmo si evolverà troppo lentamente.
La strategia scelta è la seguente: si ordinano gli individui in base alla fitness, e poi si assegna al k-esimo individuo nella lista ordinata un nuovo valore di fitness proporzionale a $\frac{1}{\sqrt{k}}$. Il coefficiente di proporzionalità viene calcolato in modo che la somma di tutti i nuovi valori di fitness sia uguale al numero totale di genitori necessari per la riproduzione.

Agendo in questo modo si evitano i problemi di cui sopra.

4.1.2 Selezione

I valori scalati della funzione fitness vengono ora utilizzati per selezionare gli individui che partecipano alla riproduzione. Ogni individuo può partecipare più volte alla riproduzione. Escludendo gli N_e individui migliori che vengono promossi a *elite children*, gli altri $N_p - N_e$ individui della nuova popolazione saranno in parte prodotti per crossover, e nella restante parte prodotti per mutazione. Il parametro $C_f \in [0, 1]$ (*crossover fraction*) specifica appunto qual è la frazione di nuovi individui da produrre per crossover.
Per effettuare il crossover sono necessari due genitori, mentre per la mutazione viene utilizzato un solo genitore.
La scelta dei genitori viene effettuata considerando idealmente una linea divisa in segmenti. Ogni segmento corrisponde ad un individuo, e la sua lunghezza è proporzionale al relativo valore scalato della funzione fitness. Scelto un passo, l'algoritmo si muove iterativamente lungo la linea selezionando come genitore l'individuo corrispondente al segmento su cui si ferma.

4.1.3 Crossover

Il crossover è un operatore che associa a due individui genitori un nuovo individuo figlio (*crossover children*) che *eredita* i geni di entrambi, secondo un certo criterio. Nel caso in cui il cromosoma sia una stringa di bit, la procedura fa uso di un vettore binario $\boldsymbol{r}$ di dimensione n generato casualmente. Se $\boldsymbol{p^{(1)}}$ e $\boldsymbol{p^{(2)}}$ sono gli individui da ricombinare, allora il nuovo individuo $\boldsymbol{c}$ sarà così definito:

$$c_j = \begin{cases} p_j^{(1)} & \text{se } r_j = 1 \\ p_j^{(2)} & \text{se } r_j = 0 \end{cases} \tag{4.1}$$

Nel caso in cui, invece, il cromosoma sia un vettore di numeri reali, la procedura calcola l'individuo figlio come combinazione convessa casuale dei due genitori.

4.1.4 Mutazione

La mutazione è un operatore che associa ad un individuo genitore un nuovo individuo (*mutation children*) che si ottiene dal primo variando in modo casuale i

suoi geni. La mutazione favorisce la diversità tra gli individui della popolazione e permette all'algoritmo genetico di cercare nello spazio delle soluzioni in modo più vasto.

L'operatore di mutazione qui utilizzato e di tipo *gaussiano*, nel senso che esso muta l'individuo genitore aggiungendo alle sue componenti un rumore gaussiano a media nulla.

La deviazione standard σ_g del rumore alla generazione g decresce con il procedere delle generazioni secondo la relazione ricorsiva:

$$\sigma_g = \sigma_{g-1} \cdot \left(1 - \rho \cdot \frac{g}{G_{max}}\right) \tag{4.2}$$

con G_{max} numero massimo di generazioni (si veda il paragrafo successivo), $\rho \in [0, 1]$ fattore di restringimento.

Il valore di partenza σ_1 della deviazione standard è dato da:

$$\sigma_1 = I^u - I^\ell \tag{4.3}$$

dove $[I^u - I^\ell]$ è l'intervallo in cui vengono generati casualmente i valori per la popolazione iniziale.

Questo operatore di mutazione può essere utilizzato solamente nel caso in cui il problema di ottimizzazione sia non vincolato, altrimenti non è possibile garantire a priori che la mutazione generi ancora un individuo ammissibile. Se l'individuo non è ammissibile va scartato, ma scartarne troppi potrebbe compromettere le prestazioni dell'algoritmo.

Per questo motivo nel caso di problemi vincolati è conveniente utilizzare un operatore di mutazione alternativo progettato apposta per problemi vincolati.

A partire dall'individuo genitore l'operatore alternativo sceglie casualmente una direzione che si adatta a quella dell'ultima generazione (che ha migliorato o meno il valore di fitness). Lungo la direzione scelta, percorre un passo casuale che garantisce il rispetto dei vincoli linari imposti.

4.1.5 Condizioni di terminazione

Ponendo $N_e > 0$ si ha la garanzia che il valore della funzione fitness calcolato sul miglior individuo sia una funzione non crescente rispetto alla successione discreta delle iterazioni dell'algoritmo[1].

[1]Tipicamente sarà $N_e \in \{1, 2, 3\}$.

Poiché l'algoritmo può decrescere(anche lentamente) per miloni di iterazioni, è necessario specificare delle condizioni al verificarsi delle quali conviene (quasi) sicuramente fermare l'esecuzione, perché procedere ulteriormente non migliorerebbe significativamente il risultato.

In particolare, l'algoritmo termina non appena si verifica una delle seguenti condizioni:

- Sono state eseguite $G_{max} > 0$ iterazioni. Il parametro G_{max} è solitamente dell'ordine delle centinaia o delle migliaia.
- La media esponenziale pesata della deviazione standard della fitness degli individui scende sotto una certa soglia.

4.2 Parallelizzazione dell'algoritmo genetico

Esistono diversi modi di parallelizzare un algoritmo genetico. Una tassonomia degli algoritmi genetici paralleli è descritta in [1].
Il metodo più semplice è il modello master/slave. In questo modello esiste una sola popolazione, ed un nodo di calcolo viene scelto come master, mentre tutti gli altri hanno il ruolo di slave. Il master effettua tutte le operazioni richieste dall'algoritmo genetico escluso il calcolo della funzione obiettivo, che è solitamente il compito più costosto in termini di tempo di calcolo. Ad ogni iterazione, dunque, il master invia un sottoinsieme della popolazione ad ogni altro nodo di calcolo ed attende che tutti i nodi terminino restituendogli i valori delle fitness appena calcolati, per poi andare avanti con l'applicazione degli operatori genetici e dunque l'evoluzione alla generazione successiva. Questo modello ha il vantaggio di lavorare su un'unica popolazione, gestita dal master, e dunque non differisce minimamente come comportamento dalla versione sequenziale, se non per il fatto che il parallelismo può produrre una riduzione del tempo di calcolo. Lo svantaggio è il grosso overhead di comunicazione, giacché è necessario che il master invii ad ogni iterazione tutta la popolazione e riceva tutti valori delle funzioni obiettivo. Considerando che gli

individui potrebbero essere oggetti molto grandi, questo modello può comportare un overhead temporale troppo grande su alcune architetture [2].
Questo modello è stato scartato perché più adatto ad una architettura basata su memoria condivisa che quindi, per via del bottleneck del bus di memoria, permette solo un limitato livello di parallelismo (*coarse-grained parallelism*).

Un modello più adatto all'architettura a memoria distribuita ibrida propria dell'SCC è invece quello basato sulle sottopopolazioni e la migrazione (*static subpopulation with migration*). In questo modello ogni nodo di calcolo gestisce la sua popolazione[3] e ad essa applica l'algoritmo genetico sequenziale descritto nella sezione 4.1, con alcune importanti differenze.
La prima differenza è l'aggiunta di una ulteriore fase dell'algoritmo, detta fase di migrazione, che viene eseguita ogni M_p iterazioni. Durante la fase di migrazione, la normale evoluzione degli individui si blocca temporaneamente, ed i nodi di calcolo si scambiano i loro individui migliori (il *materiale genetico dominante*), mescolando i propri individui con quelli ricevuti dagli altri. Avvenuto lo scambio, l'evoluzione riprende normalmente.
In poche parole, dunque, il calcolo eseguito da un generico nodo si può pensare suddiviso in due fasi che si alternano ciclicamente. Nella fase sequenziale il nodo esegue l'algoritmo genetico sequenziale standard in modo completamente isolato dagli altri nodi, come se fosse l'unico nodo nel sistema. Nella fase di migrazione, invece, i nodi scambiano tra di loro i risultati trovati fino a quel punto.
La seconda differenza riguarda le condizioni di terminazione. É stato adottato un algoritmo distribuito per decidere di arrestare l'esecuzione[4] prima che sia stato eseguito il massimo numero di migrazioni. Quest'algoritmo è presentato nella sezione 4.4.

Mentre il modello master/slave sfrutta il parallelismo senza modificare minima-

[2] Si consideri inoltre che, in generale, l'oggetti da inviare vanno serializzati e quindi la loro dimensione cresce ulteriormente

[3] che si può interpretare come sottopopolazione dellapopolazione totale gestita complessivamente da tutti i nodi di calcolo

[4] Si osservi che questo problema non esiste nel modello master/slave, in quanto è il master a decidere quando fermare l'esecuzione

mente l'algoritmo genetico sequenziale[5], il modello con sottopopolazioni e migrazione modifica l'algoritmo, perchè l'evoluzione, e quindi l'applicazione degli operatori genetici, non avviene più globalmente, ma localmente ai singoli nodi di calcolo.
In altre parole, il modello *static subpopulation with migration* scelto per l'implementazione è un vero e proprio algoritmo distribuito, più che algoritmo parallelo, e può essere adattato ad un qualsiasi tipo di sistema di calcolo distribuito (ad esempio una tradizionale rete di calcolatori).

4.3 Schema di migrazione

Un aspetto che bisogna precisare una volta scelto il modello di parallelizzazione basato su migrazione è proprio lo schema di migrazione. In altri termini è necessario descrivere dove sono diretti i flussi di migrazione, ossia lo schema di scambio di messaggi. Lo schema deve essere compatibili con il modello di scambio di messaggi disponibile (in questo caso la libreria RCCE).
La soluzione adottata prevede la ricerca, nel grafo dei nodi di calcolo, di un ciclo orientato contenente tutti i nodi di calcolo disponibili. Un tale ciclo è noto nella teoria dei grafi come *ciclo hamiltoniano*. In altri termini lo schema di migrazione si basa su un unico flusso circolare unidirezionale di individui. É possibile scegliere schemi più complessi, con più flussi che coinvolgono uno stesso nodo. Bisogna tuttavia prestare attenzione alla possibilità che si verifichino situazioni di stallo, cosa che non può mai avvenire con un unico flusso circolare.

4.3.1 Schema di comunicazione

Come scelta progettuale, la fase di migrazione non prevede lo scambio di informazioni tra tutte le coppie di nodi possibili. Ogni nodo, infatti, propaga la frazione M_f (che l'utente è libero di specificare) della propria popolazione composta dagli individui migliori al nodo successivo nel ciclo. Se N è il numero di nodi di calcolo del sistema, dunque, sono necessarie N fasi di migrazione perchè il materiale genetico scambiato si propaghi lungo tutto il ciclo. Essendo il numero iterazioni

[5]Purchè l'algoritmo sia implementato in maniera riproducibile, infatti, l'esecuzione su una macchina sequenziale da gli stessi ed identici risultati che da su una macchina parallela

(e quindi il numero di migrazioni) tipicamente molto alto (i.e. 10000), ciò non costituisce in realtà un problema.
L'utilizzo di uno schema circolare molteplici vantaggi, primo fra tutti la semplicità. Considerato inoltre che la libreria RCCE fornisce primitive di comunicazione bloccanti, il flusso circolare permette di gestire in modo semplice, elegante ed efficiente la circolazione delle informazioni, cosa che non avverrebbe in uno schema più complicato.
Più precisamente, durante una fase di migrazione ogni nodo esegue esattamente una operazione di `send` sul nodo successivo nel ciclo, ed un'operazione di `receive` sul nodo precedente. L'ordine delle operazioni dipende dalla posizione del nodo nel cammino[6]. A questo scopo, effettuiamo una colorazione del grafo con due colori, rosso e nero. Ad ogni nodo di calcolo associamo un identificatore, che è un numero intero appartenente al range $\{0, N-1\}$, dove N è il numero totale di nodi. A livello implementativo, l'identificatore di un core (nodo) corrisponderà esattamente all'identificatore (rank) che RCCE assegna all'unità di esecuzione supportata dal core. Ad ogni nodo di calcolo è inoltre associato un numero d'ordine, anch'esso compreso tra 0 ed $N-1$, che specifica l'ordine del nodo nel ciclo hamiltoniano[7]. Il numero d'ordine viene calcolato dall'algoritmo di ricerca del ciclo hamiltoniano descritto successivamente. Un nodo viene colorato di rosso se il suo numero d'ordine è pari, altrimenti viene colorato di nero. I nodi colorati di rosso eseguono prima la `send` sul nodo successivo e poi la `receive` sul nodo precedente. Viceversa, i nodi colorati di nero eseguono prima la `receive` sul nodo precedente e poi la `send` sul nodo successivo. In questo modo, se i nodi sono in numero pari, la fase di migrazione avviene in due passi. Nel primo passo i nodi rossi inviano i propri individui migliori ai nodi neri loro successivi, per un totale di N/2 trasferimenti che avvengono in parallelo (minimizzando le attese). Nel secondo passo i nodi neri inviano i propri individui migliori ai nodi rosse loro successivi, per un totale di N/2 trasferimenti paralleli. La situazione è mostrata nelle figure 4.1 e 4.2.
Se invece il numero di nodi è dispari cambia solo il comportamento dell'ultimo nodo del ciclo (quello con numero d'ordine $N-1$), che sarà un nodo rosso, e del

[6] Se un nodo effettua prima la `receive` che la `send`, non invia al nodo successivo ciò che ha appena ricevuto dal nodo precedente.

[7] Trattandosi un ciclo non avrebbe tanto senso parlare di un *primo* nodo, ma se ne sceglierà uno per convenzione.

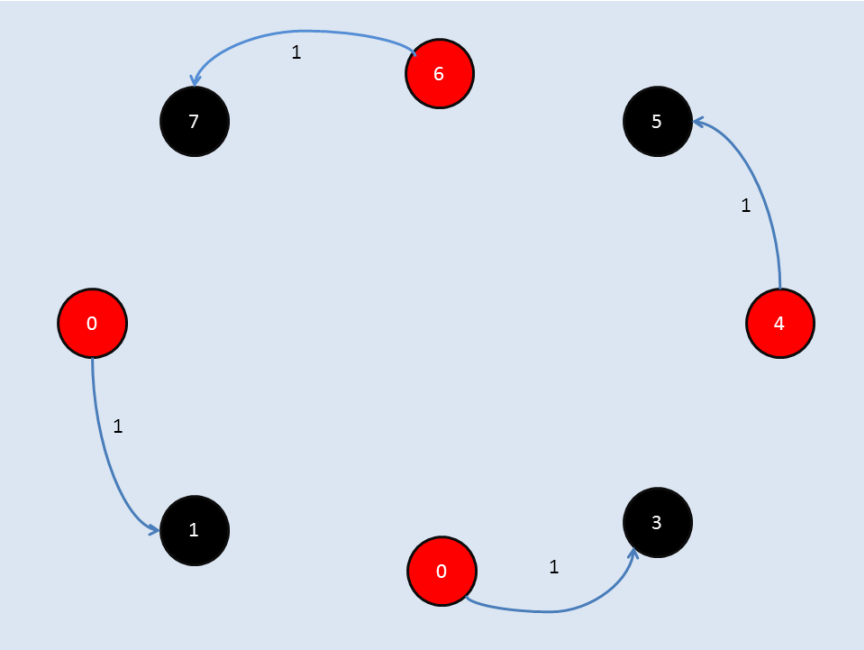

Figura 4.1: Primo passo di migrazione nel caso di numero pari di nodi. I numeri indicati sui nodi sono i numeri d'ordine.

penultimo, che sarà un nodo nero. Durante il primo passo, infatti, l'ultimo nodo del ciclo eseguirà la `send` sul primo nodo del ciclo (anch'esso rosso), ma dovrà attendere che quest'ultimo termini la `send` a sua volta invocata sul secondo nodo del ciclo. Il primo passo dell'ultimo nodo sarà di fatto temporalmente coincidente con il secondo passo del primo nodo (e quindi di tutti i primi N-2 nodi). Analogamente, nel suo secondo passo, il penultimo nodo dovrà attendere che avvenga il trasferimento tra l'ultimo nodo ed il primo, per poi effettuare il trasferimento con l'ultimo. In altri termini, pur non cambiando nulla dal punto di vista logico, nel caso di nodi pari la fase di migrazione avviene approssimativamente nel tempo necessario ad effettuare due trasferimenti tra nodi successivi (supponendo momentaneamente che i trasferimenti abbiano tutti lo stesso costo), mentre nel caso di nodi dispari è necessario un tempo pari a tre trasferimenti tra nodi successivi, perchè non tutti i trasferimenti possono avvenire in parallelo.
Si noti che la scelta di uno schema di migrazione non circolare comporterebbe difficoltà nello schema delle chiamate bloccanti, con conseguente perdita di parallelismo delle comunicazioni (un nodo potrebbe essere costretto ad aspettare che il

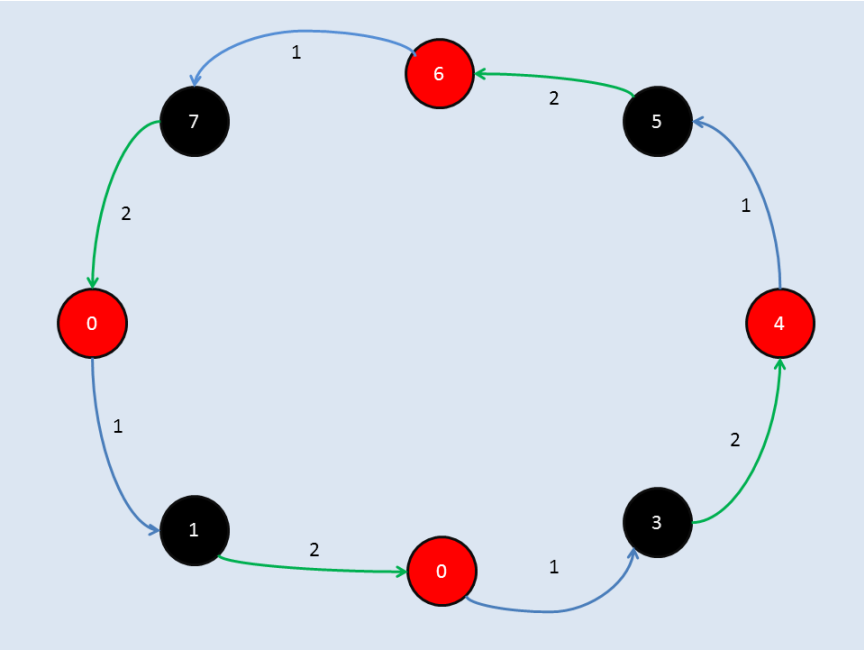

Figura 4.2: Secondo passo di migrazione nel caso di numero pari di nodi. I numeri indicati sui nodi sono i numeri d'ordine.

suo corrispondente esegua un'altra cominicazione con un nodo differente).
Nella sottosezione seguente si descriverà come scegliere un ciclo hamiltoniano *bilanciato*, ossia tale che il costo per il trasferimento di un messaggio tra due core consecutivi nel ciclo sia approssimativamente uguale per tutte le coppie di nodi consecutivi.
Nel caso dell'SCC, si assumerà come costo di trasferimento tra due core il numero di hop che separano il core ricevente dal core trasmittente, poiché tutti i messaggi hanno la stessa dimensione. Si ricorda che il routing effettuato dalla mesh è un semplice routing x-y.

4.3.2 Ricerca di cicli hamiltoniani

La ricerca di un ciclo hamiltoniano in un grafo, in generale, è un problema intrattabile. In questo caso, in realtà, la soluzione del problema è di per sè immediata, in quanto ogni core dell'SCC può comunicare con tutti gli altri core. Si tratta di un grafo completamente connesso, e per questo motivo ogni permutazione dei nodi di calcolo è un ciclo hamiltoniano.

Ciononostante non è conveniente scegliere un ciclo in modo causale, ma conviene sfruttare la topologia e le simmetrie della mesh, cercando per quanto possibile di far prevalere comunicazione tra nodi vicini, possibilmente all'interno dello stesso tile.
In generale, l'applicazione può avere a disposizione un sottoinsieme qualsiasi dei 48 core, ed è dunque necessario delineare una procedura generale per la ricerca di un cammino hamiltoniano favorevole.
La scelta che è stata fatta per eseguire in modo relativamente semplice la ricerca è la seguente:

- trattare separatamente alcuni casi particolarmente favorevoli per cui è possibile trovare la soluzione ottima in modo semplice
- nel caso generale utilizzare un'euristica semplice, ma che in pratica porta a risultati accettabili

Per non appesantire troppo l'implementazione, non sono stati trattati separatamente alcuni casi favorevoli leggermente più complicati [8].
Definiamo la *matrice di disponibilità* dell'SCC come una matrice avente 8 righe e 6 colonne che ricalca la struttura della mesh. Ogni elemento della matrice di disponibilità corrisponde al core che nella mesh ha la stessa posizione. Se il core in posizione (i, j) nella mesh è disponibile per la computazione, allora l'elemento (i, j) della matrice di disponibilità contiene l'identificatore (rank) di tale core. In caso contrario, l'elemento (i, j) contiene l'elemento -1. Gli indici di riga e colonna sono numerati a partire da 0. Chiameremo *disponibile* un elemento della matrice (i, j) se quell'elemento è maggiore di -1 (e quindi se il core corrispondente è disponibile per supportare un'unità di esecuzione dell'applicazione).
Viene fatto un primo semplice test per verificare se la matrice di disponibilità contiene un'unica sottomatrice rettangolare (di dimensioni qualsiasi) contenente solo elementi disponibili. Se il test ha risultato positivo, si cercano cammini ottimali, altrimenti si applica l'algoritmo generale [9].

[8] Sarebbe comunque semplice estendere il lavoro per trattare anche questi casi

[9] Si potrebbe estendere il test a casi in cui ci sono più sottomatrici quadrate di soli elementi disponibili

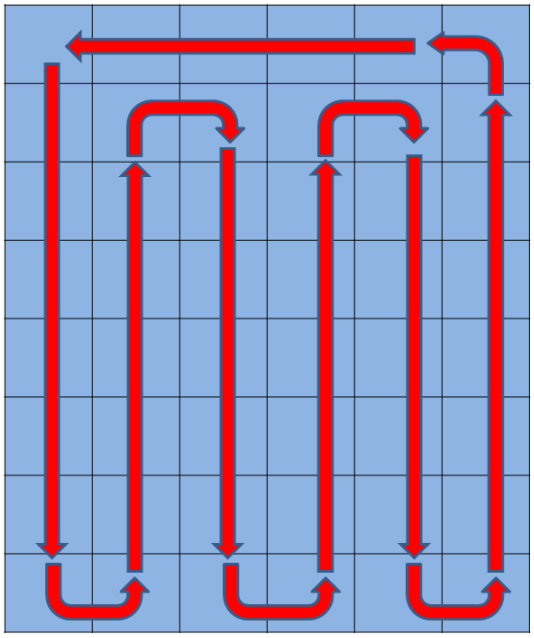

Figura 4.3: Percorso seguito dall'algoritmo generale

L'algoritmo generale consiste nello scandire la matrice di disponibilità per colonne nel modo indicato in figura 4.3 e selezionare solamente gli elementi disponibili, aggiungendoli al cammino man mano che si incontrano.

Supponiamo ora che il test precedente abbia dato risultato positivo. Nel caso in cui la sottomatrice individuata sia una matrice riga (colonna), la soluzione ottima, comunque poco bilanciata, prevede che l'ordine del ciclo segua esattamente la riga (colonna) individuata.
Poniamoci adesso nel caso più generale in cui la sottomatrice individuata abbia un numero di righe e colonne strettamente maggiore dell'unità. Di stinguiamo i casi seguenti:

1. Se la sottomatrice ha un numero dispari di righe ed un numero dispari di colonne, non è sempre possibile trovare un ciclo perfettamente bilanciato, ma si è costretti ad accettare la presenza di un trasferimento che ha costo maggiore degli altri[10]. Il ciclo scelto è evidenziato in figura 4.4, nel caso di

[10]Potrebbe comunque verificarsi il caso favorevole in cui tale comunicazione ha lo stesso costo delle altre, in quanto effettuata tra due tiles adiacienti

una sottomatrice 5x7.

2. Se la sottomatrice ha un numero di righe pari e maggiore di quattro, ha un numero di righe dispari, e se il più piccolo indice di riga corrispondente a elementi disponibili nella matrice è pari, si può seguire il percorso indicato in figura 4.5, che ha il vantaggio di comportare solo comunicazioni intra-tile nel primo passo della fase di migrazione e solo comunicazioni inter-tile (dallo stesso costo) nel secondo passo[11]. Si ottiene così una soluzione perfettamente bilanciata.

3. Se la sottomatrice ha un numero di righe multiplo di quattro, un numero di colonne maggiore di due, e se il più piccolo indice di riga corrispondente a elementi disponibili nella matrice è pari, allora si può seguire il percorso rappresentato in figura 4.6, che ha le stesse proprietà del ciclo scelto al punto precedente.

4. Se non valgono i casi 2 e 3, e la sottomatrice ha un numero pari di colonne ed un numero di righe maggiore o uguale di due, è possibile seguire un ciclo del tipo rappresentato in figura 4.7.

5. Se non valgono i casi 2 e 3, e la sottomatrice ha un numero pari di righe ed un numero di colonne maggiore o uguale di due, è possibile seguire un ciclo del tipo rappresentato in figura 4.8.

4.4 Algoritmo distribuito di terminazione

Come ultima questione, bisogna stabilire in che modo i nodi di calcolo devono coordinarsi per stabilire di fermare l'esecuzione prima che il numero massimo di iterazioni sia stato raggiunto. Nel caso sequenziale, quando l'algoritmo converge ad una soluzione (ossia quando gli individui tendono ad uniformarsi completamente), è sufficiente interrompere l'evoluzione e terminare l'esecuzione.
Nel caso parallelo, a causa della migrazione, ci sono tante popolazioni (una per ogni nodo), ed ogni nodo valuta indipendentemente dagli altri la condizione di convergenza. Di conseguenza, i nodi possono decidere di essere arrivati a convergenza in iterazioni differenti, in modo sostanzialmente non prevedibile.

[11] Si osservi che in questo caso il numero di nodi è pari

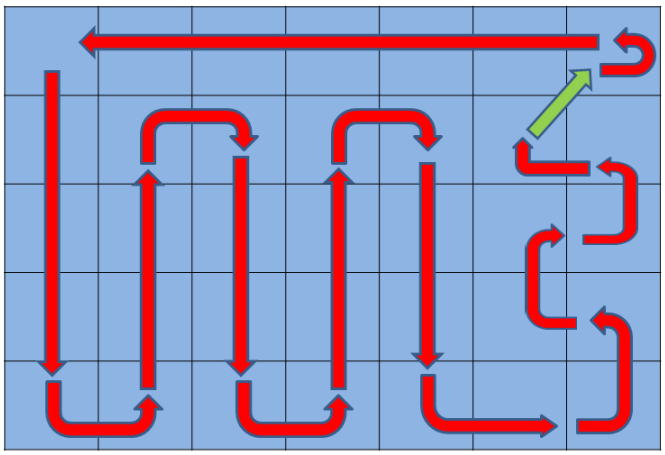

Figura 4.4: Ciclo scelto nel caso di sottomatrice con numero dispari di elementi. In verde è evidenziata la comunicazione in generale più lenta rispetto alle altre.

Inoltre, se un nodo decide che non ha più senso continuare l'evoluzione, non può semplicemente terminare l'esecuzione, perchè i nodi adiacenti (i quali potrebbero non aver ancora deciso di fermarsi) effettuano delle chiamate bloccanti su di esso: si arriverebbe così ad una situazione di stallo. Detto in altri termini, una fase di migrazione deve essere eseguita da tutti i nodi oppure da nessun nodo, altrimenti si raggiunge sicuramente una situazione di stallo.
Infine, quando tutti hanno terminato, è necessario raccogliere le soluzioni locali a ciascun nodo, cosa che può essere fatta da un solo nodo designato appositamente. In ogni caso, dunque è necessario che i tutti i nodi raggiungano un punto di sincronizzazione finale.

Per questi motivi, l'algoritmo di terminazione distribuita è stato concepito in maniera tale che l'evoluzione termini per tutti i nodi esattamente alla stessa iterazione, dopo che tutti i nodi, tramite un semplice protocollo di segnalazione, abbiano deciso di comune accordo di voler terminare.
Allo scopo di non appesantire le comunicazioni, il protocollo di segnalazione agisce

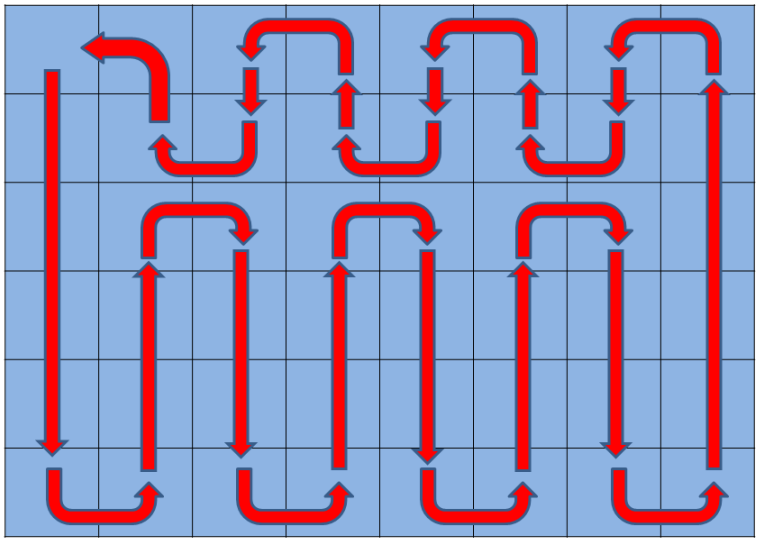

Figura 4.5: Ciclo scelto nel caso 2

solamente durante le fasi di migrazione[12]. Più in dettaglio, un byte aggiuntivo viene posto in coda al messaggio contentente il materiale genetico da propagare verso i nodi successivi. Questo byte contiene l'informazione necessaria al nodo per decidere come agire.
I segnali si trasferiscono nel ciclo muovendosi alla velocità di due nodi per migrazione. Infatti il nodo rosso avente numero d'ordine i invia nel primo passo di una fase di migrazione un messaggio al nodo nero avente numero d'ordine $i + 1$, il quale durante il secondo passo della stessa fase di migrazione invia un altro messaggio (il cui byte di segnalazione tiene conto del messaggio appena ricevuto dal nodo i) al nodo rosso avente numero d'ordine $i + 2$.

Quando nessun nodo è ancora arrivato a convergenza, il byte del protocollo di segnalazione viene impostato con un segnale (K) che indica al nodo successivo di continuare tranquillamente l'evoluzione.

[12]Infatti, anche se protocollo di segnalazione ha bisogno di trasmettere un solo byte, la trasmissione di un messaggio da un core all'altro comporta comunque un overhead dovuto alla sincronizzazione, soprattutto a causa delle attese attive

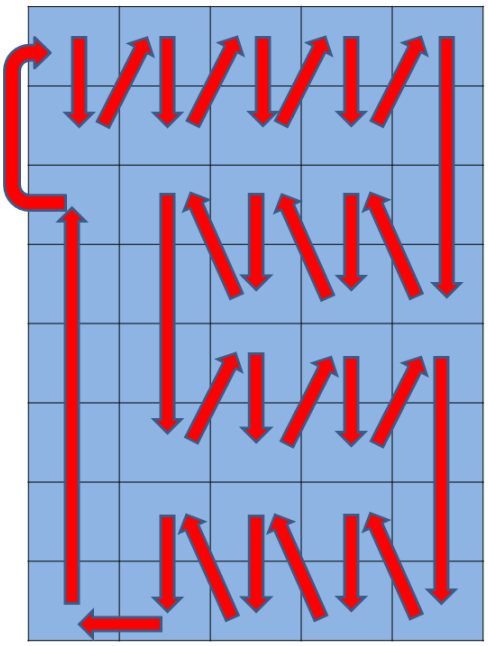

Figura 4.6: Ciclo scelto nel caso 3

Quando un nodo converge, per esso comincia la prima fase dell'algoritmo di terminazione. Il nodo arrivato a convergenza segnala al nodo successivo la sua volontà di fermarsi, inviandogli il segnale S, ma solamente se anche il nodo precedente gli ha a sua volta già inviato (in passato) tale segnale, avendo espresso la stessa volontà. Perché questa condizione si verifichi prima o poi per tutti i nodi, è chiaramente necessario prevedere l'esistenza di un nodo coordinatore, che non appena arrivato a convergenza invii S al nodo successivo il segnale S senza attendere oltre. In questo modo, quando il nodo coordinatore riceve il segnale S, è sicuro che tutti i nodi nel sistema sono arrivati a convergenza ed hanno deciso di essere pronti a fermarsi.

A questo punto comincia la seconda fase dell'algoritmo, la fase di *countdown*. Questa fase, inaugurata ancora una volta dal nodo coordinatore (l'unico che può essere sicuro che la decisione di terminare l'evoluzione sia stata presa unanimemente), ha lo scopo di far terminare tutti i nodi esattamente durante la stessa iterazione, in modo tale che non possano nascere situazioni di stallo dovute a fasi di migrazioni eseguite solo da un sottoinsieme dei nodi.
A questo scopo, il nodo coordinatore imposta un contatore locale ad un certo valore

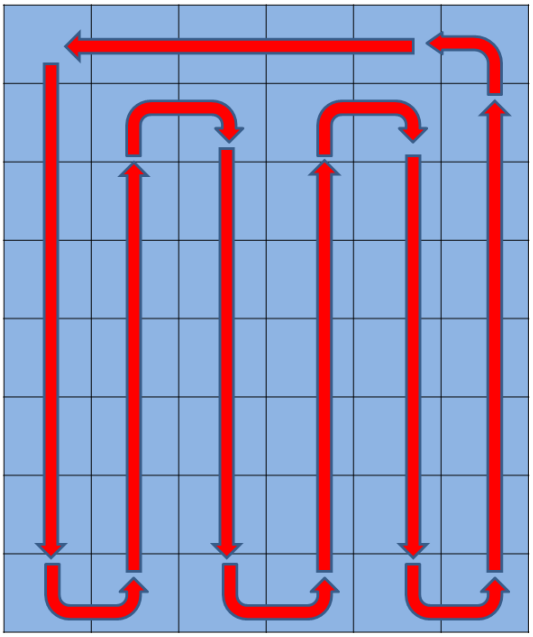

Figura 4.7: Ciclo scelto nel caso 4

(specificato di seguito) ed invia al nodo successivo il segnale `C`, che indica l'inizio della fase di countdown. Un nodo che riceve il segnale `C` lo trasmette appena possibile al suo nodo successivo, impostando a sua volta un contatore locale ad un certo valore. Ad ogni fase di migrazione successiva a quella in cui è stato impostato, il contatore viene decrementato. Il valore iniziale del contatore varia da nodo a nodo, e dipende solamente dal numero d'ordine del nodo. Questo valore deve essere calcolato in modo tale che il contatore raggiunga il valore zero per tutti i nodi esattamente nella stessa iterazione. In questo modo, quando un nodo si accorge che il proprio contatore è arrivato a zero, può terminare in tutta sicurezza l'evoluzione.
Si può verificare facilmente che, per il nodo avente numero d'ordine i il contatore deve essere impostato al valore

$$\lfloor \frac{2\lfloor \frac{N}{2} \rfloor - i}{2} \rfloor \tag{4.4}$$

Le figura 4.9 rappresenta in modo schematico la macchina a stati che esprime l'algoritmo di terminazione per quanto riguarda un nodo non coordinatore. Sugli archi che rimangono sullo stesso stato sono indicate le operazioni effettuate dal nodo

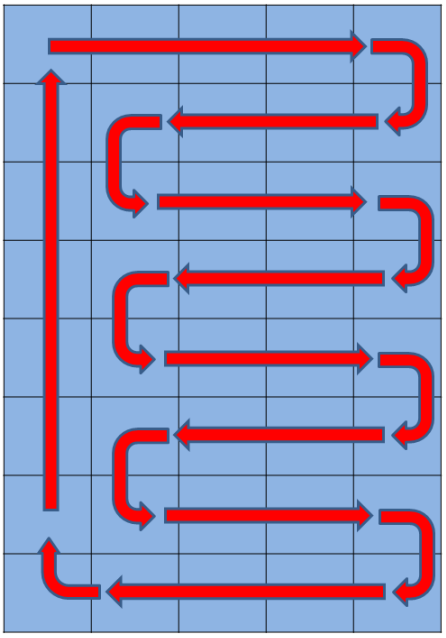

Figura 4.8: Ciclo scelto nel caso 5

quando si trova in quello stato. L'operazione `signal(X)` indica l'invio di un segnale al nodo successivo (protocollo di segnalazione). Sugli archi che collegano due stati distinti sono presenti nomi di segnali o la variabile di stato `readyToStop`. Un nome di segnale X indica l'evento 'ho ricevuto in passato il segnale X dal mio nodo precedente', mentre la variabile `readyToStop` viene settata quando un nodo arriva a convergenza.
Analogamente, la figura 4.10 rappresenta la macchina a stati che esprime l'algoritmo di terminazione relativo al nodo coordinatore.

4.5 Raccolta dei risultati

Una volta terminata l'evoluzione, bisogna raccogliere i risultati. A questo scopo un nodo di calcolo viene scelto per effettuare questo compito. Per convenzione si sceglie il nodo avente identificatore 0. Il nodo 0 invoca, in sequenza, una `receive` su ogni altro nodo. Ogni nodo con identificatore maggiore di zero, d'altra parte, effettua una `send` inviando al nodo 0 il suo individuo con fitness più alta. In questo

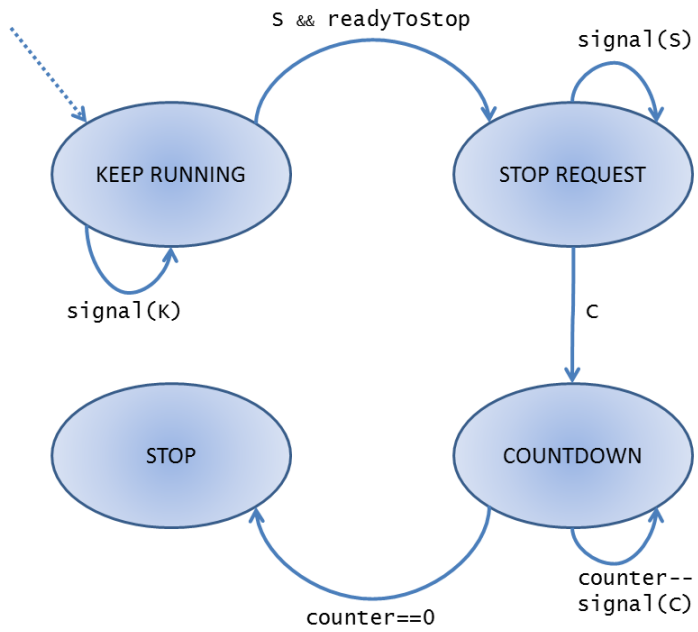

Figura 4.9: Macchina a stati per l'algoritmo di terminazione relativa ad un nodo non coordinatore

modo il nodo 0 può calcolare la soluzione globale e restituirla al chiamante (la sezione 5.1 specifica l'interfaccia offerta dal framework).

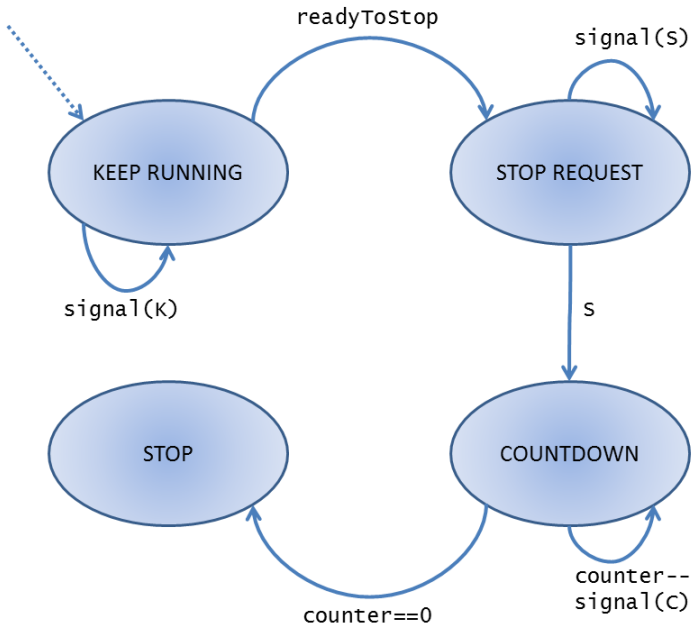

Figura 4.10: Macchina a stati per l'algoritmo di terminazione relativa al nodo coordinatore

Capitolo 5

Interfaccia ed implementazione

In questo capitolo viene presentata l'interfaccia che il framework di ottimizzazione fornisce all'utente e vengono esposti alcuni dettagli implementativi del framwork stesso.

5.1 Interfaccia

Il framework è stato implementato sull'SCC utilizzando la libreria RCCE. Poichè la libreria RCCE adotta il paradigma *Single Program Multiple Data* (SPMD), anche l'utente deve attenersi allo stesso paradigma.
Ciononostante, questo non comporta praticamente nessun disagio o difficoltà per l'utente inesperto nel momento in cui egli compila un programma che si limita ad utilizzare il framework per effettuare una qualsiasi ottimizzazione. É pertanto sufficiente seguire le istruzioni riportate di seguito.

Creato un file sorgente C++ contenente la funzione *main*, è necessario includere l'header 'ga.h'. Prima di creare gli oggetti necessari per impostare l'algoritmo, è necessario eseguire la chiamata di funzione `GAUtils::frameworkInit( &argc, &argv )`, che inizializza il framework passando a RCCE i parametri passati dalla linea di comando[1].
Una volta fatto questo, è necessario creare un oggetto della classe template `GeneticAlgorithm`. Il primo parametro template (IT) specifica la classe (o il tipo primitivo) che la fun-

[1] Una volta compilato il sorgente, si deve lanciare l'eseguibile utilizzando l'utilità `rccerun`, il cui funzionamento è descritto in [8]

zione obiettivo da minimizzare accetta in ingresso, mentre il secondo (OT) specifica il tipo di primitivo che la funzione restituisce in uscita. Il secondo parametro può essere solo uno tra `float`, `double` o `int`, in quanto la minimizzazione è mono-obiettivo (questo vincolo non impone perdita di generalità). Il costruttore della classe `GeneticAlgorithm` prende in ingresso quattro parametri:

1. Il puntatore alla funzione obiettivo da ottimizzare. La funzione deve restituire un oggetto di tipo `OT` e deve accettare in ingresso un riferimento a costante di tipo `IT` (`const IT&`).

2. Il puntatore alla funzione che implementa l'operatore di mutazione. Tale funzione non ha valore di ritorno, ed accetta due parametri in ingresso, il primo di tipo (`const IT&`) ed il secondo di tipo (`IT&`). Il primo parametro si riferisce all'individuo a cui l'operatore deve essere applicato, mentre il secondo si riferisce appunto all'individuo da creare per mutazione a partire dal primo.

3. Il puntatore alla funzione che implementa l'operatore di crossover. Tale funzione non ha valore di ritorno, ed accetta tre parametri in ingresso, i primi due di tipo (`const IT&`) ed il terzo di tipo (`IT&`). Il primi due parametri si riferiscono agli individui genitori a cui l'operatore deve essere applicato, mentre il terzo si riferisce all'individuo figlio da creare per crossover a partire dai primi due.

4. Un valore di un tipo enumerazione che specifica l'operatore di selezione. Attualmente è implementato solo l'algoritmo *Stochastic Universal Sampling*, dovuto a [9], che si specifica con il tipo enum `GAUtils::SUS`.

Il framework fornisce degli operatori generici già disponibili che possono essere utilizzati con i tipi di ingresso `float`, `double`, `FloatVector` e `DoubleVector`[2]. Gli operatori di mutazione già disponibili sono

- `GAUtils::mutationGaussian` - Crea l'individuo mutante sommando all'individuo da mutare un rumore Gaussiano a media nulla ed a varianza che decresce con l'aumentare del numero di iterazioni.

[2] I tipi `FloatVector` e `DoubleVector` sono rispettivamente di vettori di `float` e vettori di `double`, ed hanno la stessa interfaccia della classe `vector` di STL

Gli operatori di crossover già disponibili sono

- `GAUtils::crossoverConvex` - Crea l'individuo figlio come combinazione lineare convessa degli individui genitori. In formula, seleziona un numero casuale β compreso tra 0 e 1 e pone $children = \beta \cdot parent_1 + (1-\beta) \cdot parent_2$.
- `GAUtils::crossoverScattered` - L'i-esima componente dell'individuo figlio viene scelta causalmente tra la i-esima componente del primo genitore e la i-esima componente del secondo genitore.

Per utilizzare gli operatori già disponibili è sufficiente specificare il valore di tipo enum riportato nei precedenti due elenchi (i.e. `GAUtils::crossoverConvex`) al posto del corrispondente puntatore a funzione.
Sempre per quanto riguarda i tipi suindicati, il framework fornisce infine la possibilità di generare popolazioni iniziali in modo casuale. Per i tipi `float` e `double` è possibile invocare, sull'oggetto creato precedentemente, la funzione `G.gaUtils.generate( population_size, initial_population, lower_bound, upper_bound )`, che accetta in ingresso un intero che indica la dimensione della popolazione da creare, un `vector<IT>` in cui verrà costruita la popolazione iniziale ed infine due float che specificano rispettivamente il limite inferiore ed il limite superiore dell'intervallo monodimensionale a cui gli individui generati devono appartenere.
In modo analogo per i tipi `FloatVector` e `DoubleVector` è possibile invocare la funzione `G.gaUtils.generate( population_size, initial_population, lower_bounds, upper_bounds )`, che accetta in ingresso un intero che indica la dimensione della popolazione da creare, un `vector<IT>` in cui verrà costruita la popolazione iniziale ed infine due `vector<float>` che specificano rispettivamente i limiti inferiori ei superiori dell'intervallo multidimensionale a cui gli individui generati devono appartenere.

A questo punto è possibile invocare sull'oggetto algoritmo genetico il metodo `run` che lancia l'algoritmo genetico stesso. Questa funzione ritorna un valore di tipo `IT`, che è la soluzione trovata dall'algoritmo. Poichè il framework va utilizzato secondo il paradigma SPMD, in realtà ogni core esegue una sua chiamata alla funzione `run`, e ritorna un proprio valore. Come specificato nella sezione 4.5, il nodo con identificatore 0 raccoglie i risultati e dunque e proprio quest'ultimo che ritorna la soluzione globale. Gli altri nodi ritornano la loro soluzione locale (che in genere

non interessa). Per questo motivo la classe `GeneticAlgorithm` dispone di una funzione `IamMaster()` che ritorna il valore booleano `true` quando chiamata sul core avente identificatore 0. In questo modo l'utente può correttamente selezionare la soluzione globale.
La funzione `run` accetta in ingresso i seguenti parametri

1. Un riferimento ad una costante di tipo vector<IT>, che specifica la popolazione iniziale da cui l'algoritmo genetico parte.

2. Un intero che indica il massimo numero di iterazioni che possono essere eseguite prima di terminare. Valore di default = 100.

3. Un float compreso tra 0.0 ed 1.0 che specifica la frazione di crossover (*crossover fraction*). Valore di default = 0.8.

4. Un intero positivo che specifica il numero di elite children. Valore di default = 3.

5. Un float compreso tra 0.0 ed 1.0 che specifica la frazione di individui della popolazione corrente che vengono propagati da ogni nodo verso il suo successivo durante una fase di migrazione. Valore di default = 0.1.

6. Un intero positivo che specifica il periodo di migrazione, ossia il numero di iterazioni che intercorrono tra due fasi di migrazione successive.

Poichè i metodi della classe `GeneticAlgorithm` possono sollevare eccezioni di tipo `GAError`, è necessario utilizzare il costrutto `try-catch`.

5.2 Implementazione

Come già accennato in precedenza, il framework è stato implementato facendo uso dei template C++, per permettere all'utente di effettuare minimizzazioni di funzioni obiettivo il cui ingresso è qualsiasi.
Lavorando con i template, non potendo contare sulla keyword C++ `export`, il framework è quasi completamente contenuto in due header file, il file 'ga.h' (incluso dall'utente) ed il file 'gaUtils.h'. Il primo include il secondo, e contiene la definizione della classe `GeneticAlgorithm` (la classe principale) e quindi implementa il cuore dell'algoritmo. Nell'header 'ga.h' sono implementate anche le

macchine a stati relative all'algoritmo distribuito di terminazione. Per motivi legati all'implementazione è stato necessario creare due versioni di macchine a stati differenti per nodi di colore diverso, oltre alla differenza che esiste tra il nodo coordinatore e gli altri nodi. Il secondo header contiene le definizioni degli operatori genetici già disponibili e le funzioni per generare le popolazioni iniziali, per i quattro tipi indicati nella sezione precedente.

Per generare numeri random con con distribuzione uniforme e gaussiana sono state usate delle librerie liberamente scaricabili sul sito http://www.agner.org/. Il modulo per la generazione di numeri random con distribuzione uniforme è disponibile come libreria statica (‘randomaelf32.a‘), mentre il modulo che genera numeri con distribuzioni non uniforme (utilizzando la libreria statica precedente) è disponbile come sorgente C++ (‘stoc1.cpp‘).

Un aspetto importante del progetto è l'indipendenza del framework dalla libreria usata per la comunicazione (in questo caso RCCE), e quindi dall'intera architettura (in questo caso SCC). La libreria per la comunicazione è infatti racchiusa nel modulo ‘mesh2D‘, separato dal resto del framework. Questo modulo può avere bisogno di essere inizializzato (è il caso di RCCE), ma non lascia trasparire all'esterno il modo in cui questo avviene. Il suo scopo è sostanzialmente quello di fornire alla classe principale un'interfaccia che permette di trasmettere/ricevere ad/da un altro nodo, specificandone l'identificatore, un certo insieme di byte (senza interpretarli). Inoltre deve effettuare il calcolo del ciclo hamiltoniano[3] e dunque restituire al nodo chiamante i suoi parametri di configurazione nella mesh: il suo identificatore, il suo numero d'ordine, il suo colore, l'identificatore del nodo che lo precede nel ciclo, l'identificatore del nodo che ad esso segue. Note queste informazioni, l'algoritmo genetico parallelo descritto nel capitolo precedente ha tutto ciò che occorre per il funzionamento.

Come conseguenza diretta dell'indipendenza da architettura e libreria, è possibile, cambiando l'implementazione del modulo ‘mesh2D‘, far girare l'algoritmo su una tradizionale rete di calcolatori, ad esempio attraverso l'uso dei socket. Ovviamente in quest'ultimo caso può diventare problematica la ricerca di un ciclo hamiltoniano sufficienetemente bilanciato, a meno che esso non venga impostato staticamente, cosa che può avvenire ad esempio in un cluster di calcolatori connessi via LAN ad

[3] Il calcolo dipende in generale dall'architettura

alta velocità.

Il core dell'algoritmo utilizza due array di oggetti di tipo `IT`. Il primo array contiene la popolazione corrente[4], mentre il secondo viene gradualmente costruito durante l'iterazione attraverso l'applicazione degli operatori genetici agli individui contenuti nel primo array. Alla fine dell'iterazione i due array vengono scambiati con un semplice scambio di puntatori.
Per diversi motivi è necessario ordinare gli individui per fitness ad ogni iterazione. Ad esempio durante la selezione bisogna associare agli individui che hanno una migliore fitness una più alta probabilità di riprodursi (per mutazione o ricombinazione). Nelle fasi di migrazioni, inoltre, bisogna trasmettere al nodo successivo gli individui con la fitness più altra.
A questo scopo è stato implementato un heap binario di massimo[5], che in ogni iterazione viene riempito con un inserimento alla volta dagli individui della popolazione corrente (man mano che viene calcolata la loro fitness), e che viene svuotato in modo brusco alla fine dell'iterazione, in modo da poter essere riempito nuovamente alla iterazione successiva.

[4] All'inizio dell'algoritmo viene inizializzato con la popolazione iniziale passata dall'utente

[5] Anche se la funzione viene minimizzata, si utilizza un heap di massimo a causa del fitness scaling, che mappa gli individui con fitness più bassa in valori scalati maggiori, in modo che l'individuo migliore si trovi ad avere il valore di fitness scalato più grande di tutti e così via

Capitolo 6

Risultati sperimentali

Allo scopo di testare il funzionamento dell'algoritmo e verificare i vantaggi dell'elaborazione parallela, sono stati effettuati alcuni test su un problema di ottimizzazione che richiede una notevole quantità di calcoli.
Il problema scelto consiste nell'addestramento una rete neurale feed-forward a due strati con un ingresso, quattro neuroni nascosti ed una uscita. La funzione di attivazione dei neuroni nello strato nascosto è di tipo sigmoidale, mentre quella dell'unico neurone nello strato di uscita è lineare.
I dataset scelti sono costituiti da 50 elementi. Ai fini dell'ottimizzazione, i dataset sono stati mappati linearmente sull'intervallo [-1,1]. La funzione obiettivo da minimizzare è la deviazione standard dell'errore di approssimazione sugli elementi del dataset.

Nei test effettuati sono stati utilizzate popolazioni di dimensione variabile tra 15 e 200, utilizzando un numero di core progressivamente crescente. La popolazioni iniziali sono state inizializzate in modo casuale scegliendo i valori reali in un intervallo simmetrico centrato nell'origine.
Per quanto riguarda gli altri parametri dell'algoritmo genetico, i valori scelti sono riportati nella tabella di figura 6.1.

6.1 Dataset 1

Il primo dataset su cui è stata effettuata l'ottimizzazione è riportato in figura 6.2 (in cui i punti del dataset sono stati congiunti con delle spezzate).

Parametro	Valore
Massimo numero di generazioni	5000
Crossover fraction	0,8
Numero di elite children	3
Migration fraction	0,1
Periodo di migrazione	20 generazioni
Shrink factor (gaussian mutation)	1,0

Figura 6.1: Parametri dell'algoritmo genetico utilizzati.

I risultati ottenuti sono mostrati nella tabella di figura 6.4.
Gli individui delle popolazioni iniziali sono stati inizializzati casualmente con valori appartenenti all'intervallo [-10,10].
Il grafico in figura 6.3 mostra l'andamento dell'errore al crescere del numero di core nel caso in cui la popolazione iniziale su ciascuno core sia costituita da 20 elementi. Come è possibile notare, all'aumentare del numero di core impiegati il risultato ottenuto migliora.

6.2 Dataset 2

Il secondo dataset su cui è stata effettuata l'ottimizzazione è riportato in figura 6.5 (in cui i punti del dataset sono stati congiunti con delle spezzate).
I risultati ottenuti sono mostrati nella tabella di figura 6.7. In questo caso l'intervallo su cui sono stati generati casualmente gli individui delle popolazioni iniziali è stato fatto variare in funzione della dimensione della popolazione.
Il grafico in figura 6.6 mostra l'andamento dell'errore al crescere del numero di core nel caso in cui la popolazione iniziale su ciascun core sia costituita da 80 elementi. Come è possibile notare, all'aumentare del numero di core impiegati il risultato ottenuto migliora.

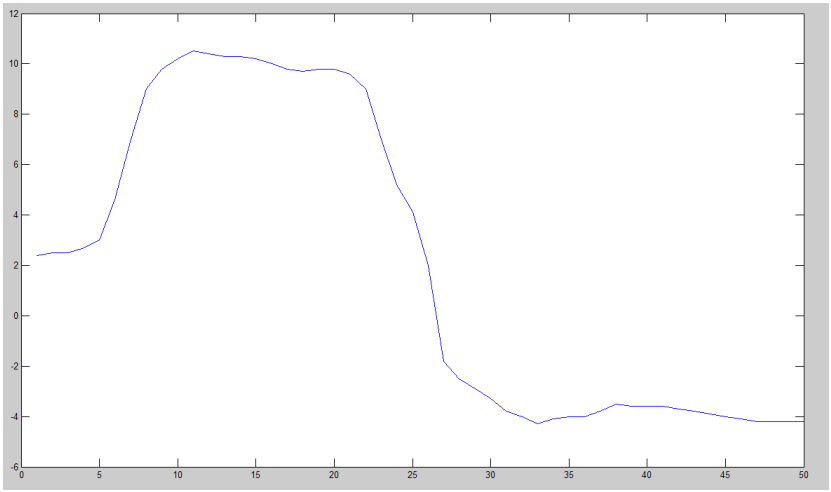

Figura 6.2: Dataset 1

6.3 Considerazioni sui risultati ottenuti

Prima di commentare i risultati, è necessario fare una osservazione. Il numero di iterazioni necessarie all'algoritmo genetico per convergere (su ciascun core) dipende molto poco dal numero di core impiegati. Questo perchè le perturbazioni casuali tramite le quali l'ottimizzazione stocastica ha luogo sono dovute principalmente all'operatore di mutazione gaussiano. Questo operatore, il cui funzionamento è descritto nella sezione 4.1.4, è stato definito in modo da far convergere l'algoritmo in un numero di iterazioni il cui ordine di grandezza è tendenzialmente lo stesso di quello del numero massimo di iterazioni specificato.
Solitamente il numero effettivo di iterazioni necessarie per arrivare a convergenza è inferiore al numero massimo di generazioni, perchè l'algoritmo genetico tende ad attestarsi su uno o più minimi locali, nel momento in cui riesce a trovarne.
Il tempo di esecuzione è dunque praticamente indipendente dal numero di core impiegati, e dipende sostanzialmente dalla dimensione della popolazione iniziale, come è possibile evincere dalle tabelle.

Il vantaggio di avere più unità di calcolo sta nella possibilità di esplorare un dominio di ricerca più ampio, e dunque nella possibilità di trovare un numero di minimi locali maggiore. Perchè questo sia possibile, bisogna inizializzare le popolazioni iniziali scegliendo valori in intervalli di inizializzazione adaguatamente

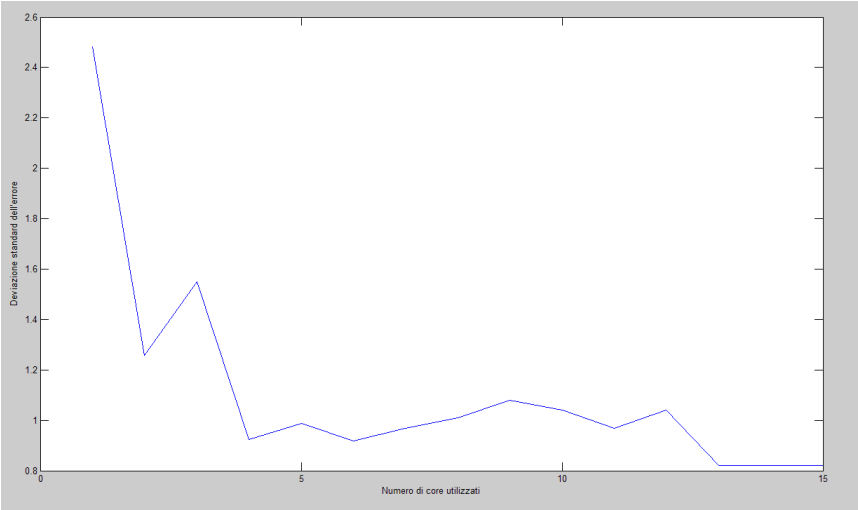

Figura 6.3: Andamento del minimo ottenuto in funzione del numero di core impiegati per il dataset 1

dimensionati.
Questi intervalli non devono esere troppo piccoli, altrimenti si rischia che tutti i core tendano a trovare gli stessi minimi locali, rendendo di fatto completamente inutile l'impiego di tanti core[1].
Allo stesso modo gli intervalli non possono essere troppo grandi, perchè altrimenti la probabilità di trovare buone soluzioni (in relazione al minimo globale, che nel caso in esame esiste) diminuisce notevolmente. Ciò accade perchè se tutti gli individui dell'algoritmo genetico si trovano in una zona di $\mathbb{R}^n$ in cui la funzione obiettivo è sostanzialmente piatta, è molto difficile l'algoritmo riesca a trovare un minimo (si procederebbe sostanzialmente per ricerca casuale). Naturalmente bisogna tenere in considerazione che scegliendo intervalli più grandi è possibile trovare soluzioni migliori poichè si allarga il dominio di ricerca.

Avendo più core a disposizione, in ogni caso, aumenta la probabilità di riuscire a trovare dei minimi locali.
Di conseguenza, è possibile compensare la diminuzione della probabilità di trovare buone soluzioni dovuta all'allargamento degli intervalli di inizializzazione con l'aumento del numero di core impegati.

[1] In pratica core diversi tenderebbero a fare lo stesso lavoro.

Gli esperimenti condotti sul secondo dataset mostrano come un buon compromesso tra numero di core e dimensione degli intervalli di inizializzazione porti ad un buon risultato (si veda la figura 6.6).

Come ultima considerazione, si può notare dalle tabelle come l'impiego di popolazioni più numerose porti in media a risultati migliori.

Dimensione popolazione	Intervallo di inizializzazione	Core utilizzati	Tempo di esecuzione	Minimo trovato
20	[-10,10]	1	11,6304	2,48052
20	[-10,10]	2	11,7304	1,25779
20	[-10,10]	3	11,7343	1,55092
20	[-10,10]	4	11,7518	0,923783
20	[-10,10]	5	11,7103	0,9867
20	[-10,10]	6	11,7472	0,9182963
20	[-10,10]	7	11,7691	0,968374
20	[-10,10]	8	11,7652	1,01134
20	[-10,10]	9	11,7592	1,07959
20	[-10,10]	10	11,7605	1,04003
20	[-10,10]	11	11,7772	0,967369
20	[-10,10]	12	11,7615	1,04077
20	[-10,10]	13	11,7709	0,820182
20	[-10,10]	14	11,7886	0,819999
20	[-10,10]	15	11,781	0,820448
40	[-10,10]	1	23,094	0,999334
40	[-10,10]	2	23,3749	0,928093
40	[-10,10]	3	23,3928	0,897
40	[-10,10]	4	23,4147	0,963512
40	[-10,10]	5	23,273	0,920977
40	[-10,10]	6	23,3863	0,877322
40	[-10,10]	7	23,3899	0,885797
40	[-10,10]	8	23,5634	0,793111
40	[-10,10]	9	23,4383	0,886601
40	[-10,10]	10	23,4872	0,88508
40	[-10,10]	11	23,4834	0,949068
40	[-10,10]	12	23,482	0,913928
40	[-10,10]	13	23,5061	0,895675
40	[-10,10]	14	23,4285	0,928242
40	[-10,10]	15	23,5007	0,925256
100	[-10,10]	1	58,0948	0,915558
100	[-10,10]	2	58,6782	0,90368
100	[-10,10]	3	58,6731	0,912437
100	[-10,10]	4	58,6988	0,877113
100	[-10,10]	5	58,7149	0,831404
100	[-10,10]	6	58,7444	0,881967
100	[-10,10]	7	58,7055	0,890126
100	[-10,10]	8	58,7666	0,888534
100	[-10,10]	9	58,7314	0,861331
100	[-10,10]	10	58,7337	0,780734
100	[-10,10]	11	58,7262	0,872388
100	[-10,10]	12	58,6923	0,854948
100	[-10,10]	13	58,779	0,879639
100	[-10,10]	14	58,7528	0,881154
100	[-10,10]	15	58,7711	0,852893
200	[-10,10]	1	116,399	0,806941
200	[-10,10]	2	117,683	0,773813
200	[-10,10]	3	117,558	0,762187
200	[-10,10]	4	117,593	0,75609
200	[-10,10]	5	117,64	0,755486
200	[-10,10]	6	117,598	0,760713
200	[-10,10]	7	117,693	0,719338
200	[-10,10]	8	117,75	0,745024
200	[-10,10]	9	117,653	0,740501
200	[-10,10]	10	117,606	0,749774
200	[-10,10]	11	117,701	0,751782
200	[-10,10]	12	117,52	0,763753
200	[-10,10]	13	117,571	0,77848
200	[-10,10]	14	117,668	0,77835
200	[-10,10]	15	117,67	0,756412

Figura 6.4: Tabella dei risultati per il dataset 1

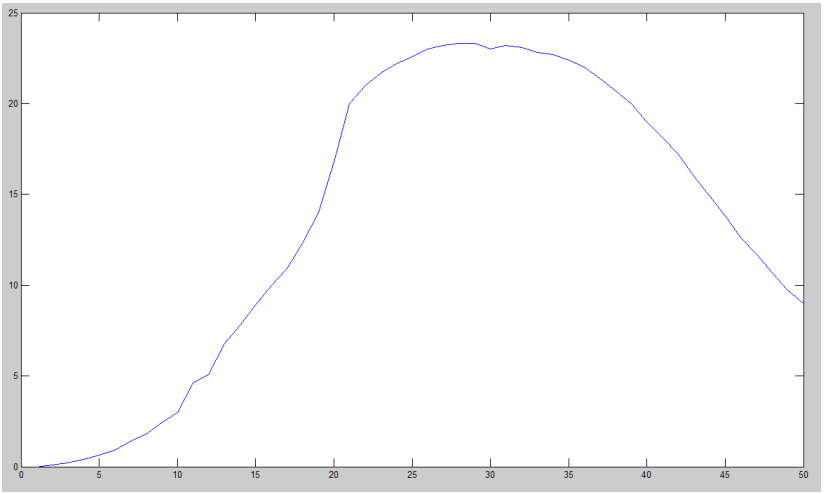

Figura 6.5: Dataset 2

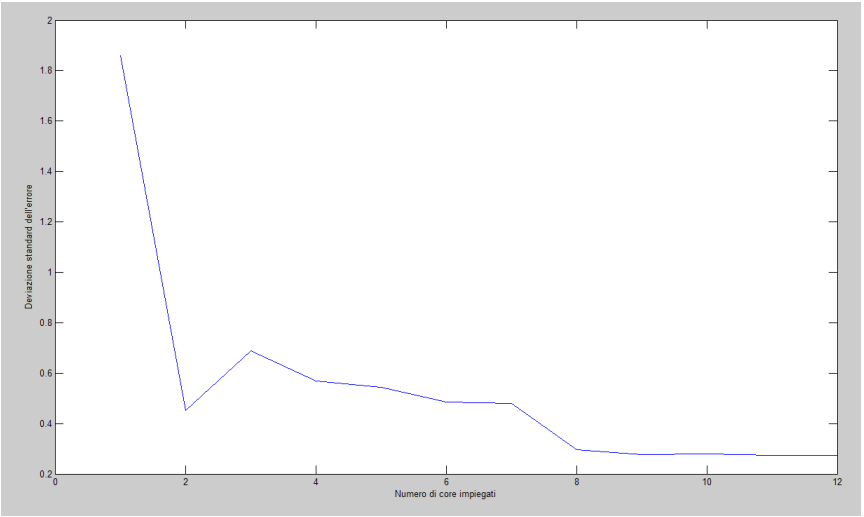

Figura 6.6: Andamento del minimo ottenuto in funzione del numero di core impiegati per il dataset 2

Dimensione popolazione	Intervallo di inizializzazione	Core utilizzati	Tempo di esecuzione	Minimo trovato
20	[-10,10]	1	11,6324	2,15854
20	[-10,10]	2	11,7395	0,41716
20	[-10,10]	3	11,7201	0,547362
20	[-10,10]	4	11,7432	0,360455
20	[-10,10]	5	11,734	0,435141
20	[-10,10]	6	11,7441	0,326799
20	[-10,10]	7	11,7573	0,562178
20	[-10,10]	8	11,7506	0,358399
20	[-10,10]	9	11,7481	0,486682
20	[-10,10]	10	11,7126	0,35107
20	[-10,10]	11	11,7675	0,326036
20	[-10,10]	12	11,7639	0,328033
40	[-15,15]	1	23,2204	2,30857
40	[-15,15]	2	23,4465	0,488601
40	[-15,15]	3	23,4724	0,416216
40	[-15,15]	4	23,4401	0,371194
40	[-15,15]	5	23,4637	0,477581
40	[-15,15]	6	23,4131	0,372113
40	[-15,15]	7	23,4931	0,349183
40	[-15,15]	8	23,5052	0,344357
40	[-15,15]	9	23,4966	0,319741
40	[-15,15]	10	23,5244	0,360389
40	[-15,15]	11	23,5092	0,38529
40	[-15,15]	12	23,5367	0,382599
80	[-20,20]	1	46,5392	1,86003
80	[-20,20]	2	47,0165	0,45209
80	[-20,20]	3	46,8581	0,686955
80	[-20,20]	4	46,9152	0,568158
80	[-20,20]	5	46,9321	0,542676
80	[-20,20]	6	47,0732	0,484291
80	[-20,20]	7	46,9365	0,479617
80	[-20,20]	8	46,964	0,29701
80	[-20,20]	9	46,8979	0,277331
80	[-20,20]	10	46,9471	0,278206
80	[-20,20]	11	46,9518	0,274325
80	[-20,20]	12	46,9533	0,275084
160	[-30,30]	1	93,0736	0,49708
160	[-30,30]	2	93,817	0,353486
160	[-30,30]	3	94,1815	0,33497
160	[-30,30]	4	94,1201	0,324885
160	[-30,30]	5	94,301	0,325818
160	[-30,30]	6	94,2542	0,323735
160	[-30,30]	7	94,2526	0,321943
160	[-30,30]	8	94,262	0,322298
160	[-30,30]	9	94,3054	0,3236
160	[-30,30]	10	94,332	0,321689
160	[-30,30]	11	94,3177	0,322855
160	[-30,30]	12	94,2944	0,32193

Figura 6.7: Tabella dei risultati per il dataset 2

Bibliografia

[1] Riccardo Poli, Mariusz Nowostawski (1999)
Parallel Genetic Algorithm Taxonomy

[2] Max Baron (2010)
The Single-chip Cloud Computer
Microprocessor report, www.MPRonline.com

[3] Intel (1995) Pentium® Processor Family Developer's Manual, Architecture and Programming Manual

[4] Howard, Dighe et al. (2010)
A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS

[5] Tim Mattson, Rob van der Wijngaart (2010)
RCCE: a Small Library for Many-Core Communication - Specification document

[6] Rob van der Wijngaart, Timothy Mattson, Werner Haas (2010)
Light-weight Communications on Intel's Single-Chip Cloud Computer Processor

[7] Intel labs (2010)
SCC External Architecture Specification (EAS), Revision 1.1

[8] Intel labs (2010)
The SCC Programmer's Guide, Revision 0.75

[9] Baker, James E. (1987) Reducing Bias and Inefficiency in the Selection Algorithm Proceedings of the Second International Conference on Gene-

tic Algorithms and their Application (Hillsdale, New Jersey: L. Erlbaum Associates)