Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Implementazione del modello di Biham-Middleton-Levine sulla Parallella Board

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Presentata da:
Adriano Cardace

Sessione II
Anno Accademico 2016/2017

Alla mia famiglia.

Sommario

In questa tesi si descrive l'architettura hardware e software della Parallella Board, un supercomputer delle dimensioni di una carta di credito che include una cpu ARM A9 dual core e un coprocessore Epiphany con 16 o 64 core. Grazie al suo basso costo e ad un hardware particolare, si tratta di un ottimo strumento sia per il programmatore esperto sia per lo studente che vuole avvicinarsi alla programmazione parallela. Tuttavia, a causa della limitata documentazione e dei pochi e complessi esempi disponibili, programmare la Parallella potrebbe richiedere un notevole impegno; l'obbiettivo è dunque quello di fornire una panoramica dettagliata della Parallella per aiutare chiunque sia interessato nello sviluppo di algoritmi paralleli a familiarizzare velocemente con questo dispositivo. Per comprendere nel dettaglio come programmare la Parallella Board, vengono forniti una serie di esempi introduttivi prima di arrivare ad un esempio finale più complesso che consiste in un'implementazione del modello di Biham-Middleton-Levine; nonostante sia possibile utilizzare diverse estensioni per il calcolo parallelo come OpenMP e MPI, si è scelto di scrivere i programmi utilizzando il C e le librerie di supporto fornite nativamente dall'eSDK (Epiphany Software development Kit).

Indice

Sommario **i**

1 Introduzione **1**

2 Struttura hardware della scheda Parallella **5**

2.1 Storia della Parallella 5

2.2 Descrizione dell'hardware 6

2.2.1 Processore e coprocessore 6

2.2.2 Memoria 7

2.2.3 Comunicazione 8

3 Programmare la parallella **10**

3.1 L'ambiente di sviluppo 10

3.2 "Hello world" per la Parallella 11

3.2.1 Compilazione ed esecuzione 18

3.3 Prodotto scalare tra vettori con la Parallella 20

3.3.1 Header file 20

3.3.2 Sorgente host 21

3.3.3 Sorgente Epiphany 23

3.4 Prodotto tra una matrice e uno scalare 24

3.4.1 struct.h 25

3.4.2 Sorgente host 25

3.4.3 Sorgente Epiphany 29

4 Modello di Biham-Middleton-Levine **33**
4.1 Descrizione del modello 33

5 Algoritmo sequenziale **35**
5.1 Versione sequenziale 35
5.1.1 Variabili principali 35
5.1.2 Algoritmo 36

6 Algoritmi Paralleli **41**
6.1 Distribuzione a righe 41
6.1.1 Strutture dati 42
6.1.2 Sorgente host 43
6.1.3 Sorgente Epiphany 48
6.2 Distribuzione a blocchi 55
6.2.1 Strutture dati 56
6.2.2 Sorgente Epiphany 57

7 Analisi delle prestazioni **64**
7.1 Analisi teorica 64
7.2 Tempo d'esecuzione 65

Conclusioni **67**

Bibliografia **68**

Elenco delle figure

1.1 Tassonomia di Flynn [9] 3
1.2 Strong scalability 4

2.1 Parallella Board [5] 6
2.2 Layout della memoria del coprocessore Epiphany [6] 8
2.3 Architettura del chip Epiphany [6] 9

3.1 Output hello world su Parallella 20

4.1 Esempi di griglie 256x256 dopo 1024 passi 34

6.1 Distribuzione a righe (in rosso gli elementi che vengono scambiati). . . . 41
6.2 Distribuzione a blocchi (in rosso gli elementi che vengono scambiati). . . 55

7.1 SpeedUp ottenuto con distribuzione a righe e a blocchi. 66

Elenco delle tabelle

7.1 Tempo d'esecuzione con distribuzione a righe e a blocchi. 65

Capitolo 1

Introduzione

A partire dagli anni 60 fino ai nostri giorni abbiamo assistito ad una costante crescita delle prestazioni dei calcolatori, basti pensare ad esempio alla legge di Moore [1], che afferma che circa ogni 24 mesi il numero di transistor raddoppia. Il risultato di questa crescita è che ormai siamo abituati a pensare che le prestazioni derivano principalmente dall'hardware, trascurando l'aspetto software delle applicazioni. Inoltre, un aumento continuo delle prestazioni si traduce in una maggiore frequenza dei clock delle CPU, il che significa una maggiore potenza dissipata che può portare in alcuni casi al surriscaldamento del processore stesso. La potenza P può essere espressa mediante la formula $P = C \times V^2 \times f$, e poiché ad un aumento del voltaggio V corrisponde un aumento proporzionale della frequenza f possiamo concludere che $P \in \mathcal{O}(f^3)$, ovvero che la potenza dissipata cresce come il cubo della frequenza; in termini pratici questo significa che un processore con una frequenza di clock pari a f consuma circa 4 volte quello che consumano due processori con frequenza $\frac{f}{2}$. L'osservazione precedente esprime l'importanza delle architetture parallele, perché utilizzando più core che eseguono istruzioni ad una minore frequenza, possiamo ottenere lo stesso risultato di un processore con un singolo core ma riducendo drasticamente la quantità di potenza dissipata. La tassonomia di Flynn [2] (figura 1.1) classifica i sistemi paralleli a seconda della molteplicità dei flussi di istruzioni e dei dati che possono gestire:

- SISD (single instruction-single data): si tratta del classico sistema di Von Neumann, in cui viene eseguita una singola istruzione su un singolo flusso di dati

seguendo il ciclo fetch-decode-execute.

- SIMD (single instruction-multiple data): sono sistemi paralleli in cui la stessa istruzione viene eseguita su più dati. Si pensi ad esempio alla somma di due vettori x e y: l'istruzione da eseguire è sempre la somma, ma questa può essere eseguita su più elementi contemporaneamente. I sistemi SIMD sono alla base delle GPU[1] (graphics proccesing units).

- MISD (multiple instruction-single data): questo tipo di architettura prevede un flusso multiplo di istruzioni e un flusso singolo di dati. Attualmente non ha particolari interessi pratici.

- MIMD (multiple instruction-multiple data): consiste in sistemi che supportano più flussi di istruzioni e più flussi di dati. Tipicamente si tratta di di una collezione di core indipendenti che lavorano su dati differenti. Le principali categorie di sistemi MIMD sono a memoria condivisa e a memoria distribuita. Nel primo caso si ha un insieme di processori che utilizzano uno spazio di memoria condiviso per la comunicazione, nel secondo caso ogni processore possiede la sua memoria privata e comunicano attraverso una rete di comunicazione.

Nella pratica le principali architetture HPC (high performance computing) usano varie tecnologie dando vita a sistemi ibridi: questo è proprio il caso della Parallella Board, che è particolarmente adatta alla programmazione a memoria condivisa, ma che grazie al suo particolare hardware permette anche di utilizzare il paradigma a memoria distribuita. In generale è preferibile usare architetture a memoria distribuita quando si ha a che fare con applicazioni in cui si prevede un alto rapporto computazione/comunicazione, in modo da ridurre al minimo la quantità di informazioni da scambiare, dato che i tempi di latenza della rete di comunicazioni potrebbero essere troppo elevati e degradare le prestazioni. Dalla loro parte i sistemi a memoria condivisa sono generalmente più facili da programmare, dato che lo spazio di indirizzamento è unico e globale; la comunicazione in questo caso avviene tramite metodi di sincronizzazione come mutex o barriere. La

[1] Si tratta di una particolare tipo di coprocessore particolarmente utile nel rendering di immagini grafiche

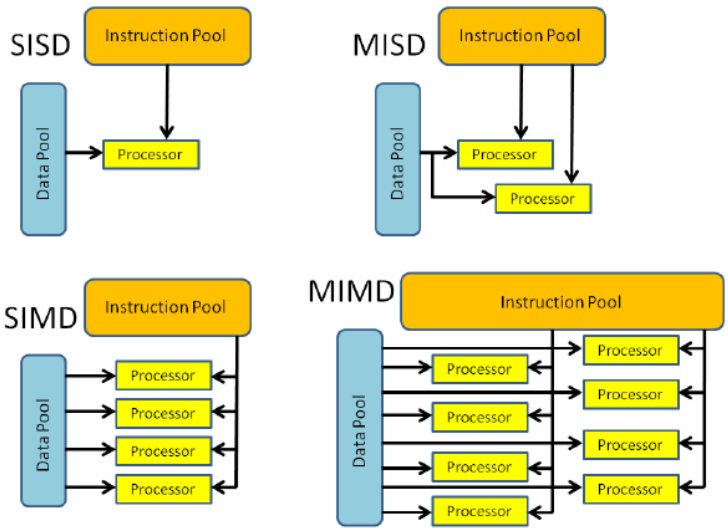


Figura 1.1: Tassonomia di Flynn [9]

valutazione di un algoritmo parallelo avviene attraverso il concetto di speeduup ($S(p)$), ovvero il rapporto tra il tempo impiegato dallo stesso algoritmo nella versione sequenziale (t_{serial}) e in quella parallela ($t_{\text{parallel}}(p)$) utilizzando p processori:

$$S(p) = \frac{t_{serial}}{t_{parallel}(p)} \tag{1.1}$$

Nel migliore dei casi ci aspetteremmo che la quantità di lavoro da svolgere sia equamente suddivisa tra i processori, ottenendo uno speedup uguale a p (in questo caso si parla di speedup lineare). In pratica questa situazione accade raramente, ad esempio a causa della presenza di sezioni critiche[2], oppure a causa dei costi di comunicazione che dominano su quelli della computazione. È naturale inoltre pensare che ad un elevato numero di processori corrisponda un miglior speedup; sfortunatamente non è sempre così. Bisogna infatti considerare l'overhead aggiuntivo dovuto all'utilizzo di più processori, che spesso porta ad un degrado inatteso delle prestazioni (la figura 1.2 esprime proprio questo concetto).

[2]Per sezioni critiche si intende porzioni di codice che non posso essere esguite contemporaneamnte da diversi thread.

Proprio per questo motivo può risultare utile il concetto di strong scalability, che permette di verificare come cambia il tempo d'esecuzione all'aumentare del numero dei core. Se mantenendo fissata la dimensione del problema e aumentando il numero di processori/threads lo speedup rimane costante, il programma si dice *strongly scalable*.

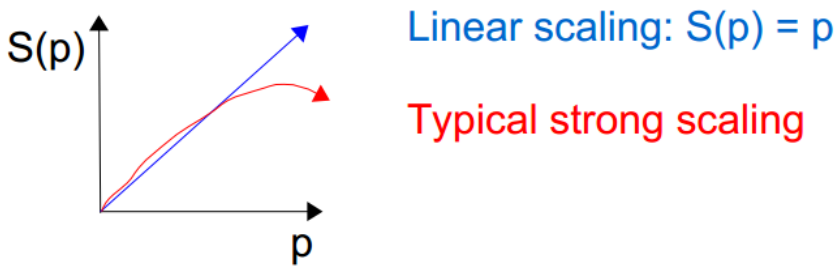


Figura 1.2: Strong scalability

Un altro parametro utile per verificare la bontà di un algoritmo parallelo è l'efficienza:

$$E(p) = \frac{S(p)}{p} \tag{1.2}$$

È facile vedere che se l'efficienza è pari a 1 stiamo usando le risorse a disposizione nel miglior modo possibile. Speedup, efficienza e strong scalability sono fattori molto importanti nella valutazione di un algoritmo parallelo, per questo motivo prima di fornire una versione parallela per la simulazione del modello di Biham-Middleton-Levine descriveremo un algoritmo sequenziale, per effettuare successivamente un confronto tra le due versioni.

Capitolo 2

Struttura hardware della scheda Parallella

In questo capitolo descriveremo l'hardware della Parallella [5]. Comprendere questi aspetti, in particolar modo l'organizzazione della memoria, sono di fondamentale importanza per capire come programmare la scheda.

2.1 Storia della Parallella

La Parallella Board, illustrata in figura 2.1, è una scheda della dimensione di una carta di credito che è stata definita come il "supercomputer per chiunque". Lanciata per la prima volta da Adaptiva[1] nel 2012 su Kickstarter[2], raggiunse un finanziamento pari a 898,921$, suscitando un grande interesse grazie un ad un prezzo di lancio competitivo (solo 100$), oltre che per le sue particolari caratteristiche. I principali campi d'utilizzo della scheda sono nella comunicazione, nel supporto alla strumentazione medica e nel riconoscimento d'immagini. Numerosi esempi di programmazione possono essere trovati nel repository ufficiale GitHub all'indirizzo https://github.com/parallella/parallella-examples. Tuttavia, a causa dell'elevata complessità di tali esempi e ad una documentazione non ottimale, comprendere nel dettaglio come programmare la Parallella potrebbe richiedere

[1] Azienda statunitense fondata nel 2008 da Andreas Olofsson.

[2] Kickstarter è un sito web di finanziamento collettivo per progetti.

molto tempo; per questo motivo si è scelto di presentare degli esempi di complessità crescente fino ad arrivare ad un programma finale che illustra le principali caratteristiche della scheda. Come primo passo però, cercheremo di analizzare l'hardware per poi addentrarci negli aspetti che riguardano l'SDK Epiphany, ovvero l'ambiente di sviluppo della Parallela.

Figura 2.1: Parallella Board [5]

2.2 Descrizione dell'hardware

2.2.1 Processore e coprocessore

La Parallella dispone di un processore dual core ARM A9, grazie al quale è possibile utilizzare diverse distribuzioni Linux come Ubuntu. La vera novità risiede nel coprocessore, un chip Epiphany con 16 o 64 core RISC (Reduced Instruction Set Computer) disposti in una griglia 4x4 oppure 8x8. Poiché per questa tesi si è utilizzato la versione a 16 core si farà sempre riferimento a questa tipologia di chip. Ciascun core ha una frequenza di clock di 1Ghz, ed è in grado di eseguire due istruzioni in virgola mobile in ogni ciclo di clock, il che significa che le prestazioni teoriche di picco per un chip con

16 core sono di 32 GFLOPS[3]. Ogni core è identificato da un ID univoco di 12 bit, in cui i 6 bit più significativi sono utilizzati per specificare la riga, mentre i restanti 6 bit per indicizzare la colonna all'interno della griglia 2D. Un aspetto importante di questo tipo di organizzazione è che il processore ARM può caricare programmi diversi su core diversi semplicemente riferendosi ad essi tramite il loro ID, oppure più semplicemente può utilizzare tutti i core per eseguire lo stesso programma. Questo secondo metodo di operare è detto SPMD (Single Program Multiple Data); noi seguiremo questo modello di programmazione in tutti i nostri esempi. Da qui in avanti ci riferiremo al chip Epiphany con il termine "device", mentre chiameremo "host" il processore ARM.

2.2.2 Memoria

La scheda dispone di 1GB di memoria RAM, per la maggior parte riservata al sistema operativo, per questo motivo viene chiama dalla comunità Epiphany "O/S DRAM"; al suo interno è presente una piccola area di 32MB, chiamata "Shared DRAM" o "memoria esterna/condivisa", che può essere utilizzata per la comunicazione bidirezionale tra host e device. Per quanto riguarda il device, ogni singolo core dispone di 32KB di memoria locale divisi in 4 banchi da 8KB; questa piccola porzione di memoria è in realà condivisa, questo significa che ciascun core può accedere alla memoria locale di un altro core, indipendentemente dalla loro posizione nella griglia. Dal punto di vista di un core, l'inizio della propria memoria locale parte dall'indirizzo 0x0 e termina all'indirizzo 0x00007FFF. Se invece si vuole fare riferimento alla memoria di un altro core gli indirizzi di inizio e fine sono 0x???00000 - 0x???07FFF, dove al posto dei 12 bit contrassegnati con la tripla "???" va inserito l'indirizzo di uno specifico core. Di default il primo core ha coordinate (32,8) e fa riferimento all'indirizzo 0x808, mentre il sedicesimo e ultimo core con coordinate (35,11) si trova all'indirizzo 0x8CB. Per visualizzare gli indirizzi degli altri core si faccia riferimento alla figura 2.2. Non è presente alcuna memoria cache.

[3]FLOPS è l'acronimo di Floating point Operations Per Second, con il quale si indica il numero di operazioni in virgola mobile eseguite in un secondo da una CPU.

Core Number	Start Address	End Address	Size
(32,8)	80800000	80807FFF	32KB
(32,9)	80900000	80907FFF	32KB
(32,10)	80A00000	80A07FFF	32KB
(32,11)	80B00000	80B07FFF	32KB
(33,8)	84800000	84807FFF	32KB
(33,9)	84900000	84907FFF	32KB
(33,10)	84A00000	84A07FFF	32KB
(33,11)	84B00000	84B07FFF	32KB
(34,8)	88800000	88807FFF	32KB
(34,9)	88900000	88907FFF	32KB
(34,10)	88A00000	88A07FFF	32KB
(34,11)	88B00000	88B07FFF	32KB
(35,8)	8C800000	8C807FFF	32KB
(35,9)	8C900000	8C907FFF	32KB
(35,10)	8CA00000	8CA07FFF	32KB
(35,11)	8CB00000	9CB07FFF	32KB

Figura 2.2: Layout della memoria del coprocessore Epiphany [6]

2.2.3 Comunicazione

La comunicazione tra i vari core è possibile grazie ad una rete 2D (figura 2.3) composta da tre canali: uno per le richieste di scrittura on-chip, un altro per le richieste di lettura on-chip, e infine un terzo canale per la lettura/scrittura off-chip, dato che è possibile creare cluster[4] di Parallella. Ciascun core può accedere sia in lettura che in scrittura alla memoria di un altro core; ovviamente accedere alla memoria dei vicini è più efficiente che accedere alla memoria di un core posizionato nella parte opposta della griglia. Un fattore molto importante da ricordare se si vuole ottenere il massimo delle prestazioni è che la scrittura avviene due volte più velocemente della lettura; in questo secondo caso infatti bisognerebbe prima inviare la richiesta per ottenere dei dati ad un particolare indirizzo, per poi riceverli, mentre nel primo caso si può scrivere direttamente ad un particolare

[4]Un claster è un insieme di computer connessi da una rete.

indirizzo. Ci sono due metodi principali per ottenere la comunicazione tra core: il primo prevede l'utilizzo diretto di una delle due DMA (Direct Memory Access) disponibili in ciascun core, grazie alla quale è possibile spostare blocchi di dati da un core all'altro; il secondo metodo di comunicazione avviene sfruttando il fatto che la memoria locale di ciascun core à mappata all'interno di uno spazio unico non protetto dove chiunque può leggere e scrivere in qualunque indirizzo. Anche l'host può accedere direttamente alla memoria locale del device grazie alle funzioni di librerie fornite nell'eSDK [3], tuttavia per semplicità di preferisce utilizzare la Shared DRAM per la comunicazione host-device.

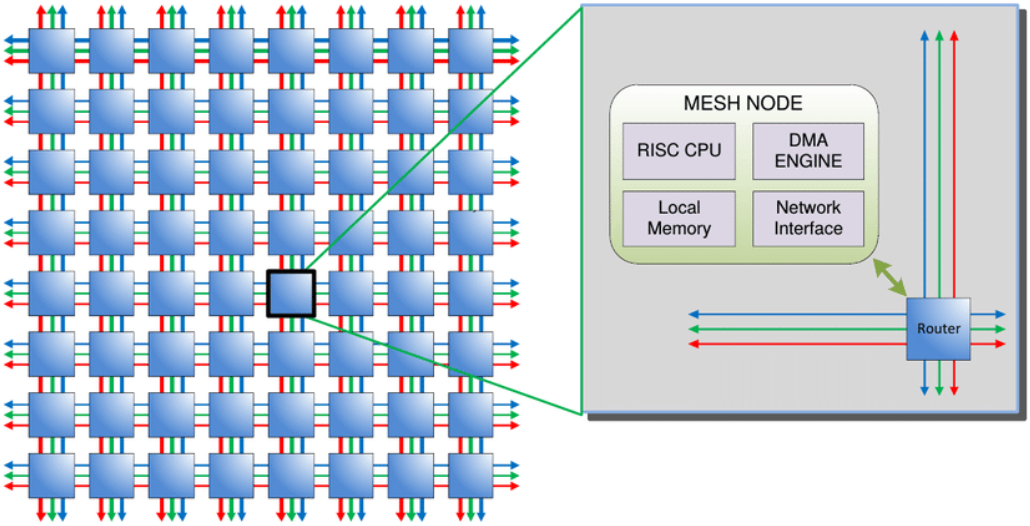


Figura 2.3: Architettura del chip Epiphany [6]

Capitolo 3

Programmare la parallella

In questo capitolo analizzeremo l'eSDK, ovvero l'ambiente di sviluppo della Parallella, che comprende tutti gli strumenti, le librerie e i linker script.

3.1 L'ambiente di sviluppo

I principali componenti dell'eSDK sono:

- Compilatore ANSI-C basato su gcc (GNU Compiler Collection).
- Editor basato su Eclipse.
- Multicore debugger basato su gdb (GNU Debugger).
- Insieme di librerie per la comunicazione e la programmazione multicore.

Notare che l'editor non è più aggiornato e presenta ancora molti problemi; per questo motivo si consiglia di utilizzare un normale editor come VI o nano, disponibili di defaulft su tutte le distribuzioni GNU/Linux. Un programma per la Parallella è composto tipicamente da due sorgenti C: il primo, eseguito dall'host, ha la responsabilità di gestire e caricare il secondo, che viene invece eseguito dal device e nel quale viene effettuata la computazione vera e propria. La Parallella fornisce un livello di astrazione chiamato eHAL (Epiphany Hardware Abstraction Layer), che facilita la programmazione lato host e contiene tutte le funzioni necessarie per gestire correttamente il device. Per usufruire

di queste funzionalità è necessario importare l'header file *e-hal.h*. Per la programmazione lato device, si utilizza invece la Epiphany Hardware Utility Library (eLib), che contiene un insieme di funzioni utili per gestire la comunicazione tra i core e per interrogare il chip Epiphany, ottenendo così informazioni su di esso. In questo caso l'header file da includere nel sorgente relativo al device è *e_lib.h*. Un elemento fondamentale nell'ambiente di lavoro della Parallella è il concetto di *Workgroup*: si tratta di un'area rettangolare che definisce i core del device da utilizzare, non è infatti obbligatorio utilizzarli tutti. Per definire un workgroup è necessario specificare le coordinate del primo nodo che corrisponde al primo core in alto a sinistra del rettangolo, e il numero di righe e di colonne da utilizzare a partire da esso. In generale i passi necessari per eseguire un programma sono i seguenti:

1. Il programma host definisce un workgroup.
2. L'host inizializza tutti i core e carica l'eseguibile per il device all'interno del workgroup.
3. L'host invia un segnale ai core del workgroup per iniziare la computazione.
4. Durante la computazione host e device possono comunicare per sincronizzarsi e salvare eventuali risultati parziali.
5. Alla fine delle computazione l'host legge il risultato finale comunicando direttamente con il chip Epiphany oppure utilizzando la memoria condivisa.

3.2 "Hello world" per la Parallella

In questo primo esempio vedremo il classico "hello world" adattato alla Parallella. Sebbene sia possibile utilizzare le più popolari librerie per il calcolo parallelo come OpenMP [7] e MPI [8], in tutti gli esempi qui riportati faremo riferimento solo alle funzioni disponibili nell'eSDK. Inizieremo analizzando il sorgente relativo al device:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include "e_lib.h"

int main(void) {
    unsigned my_row;
    unsigned my_col;
    char *msg, *done;
    e_coreid_t coreid;
    msg = (char *) 0x2000;
    done = (char *) 0x7000;
    coreid = e_get_coreid();
    e_coords_from_coreid(coreid, &my_row, &my_col);
    sprintf (msg, "Hello World from core 0x%03x, with relative coords %u,
        %u", coreid, my_row, my_col);
    *done = 1;
    return EXIT_SUCCESS;
}
```

In in questo caso il compito dei core è quello di ottenere alcune informazioni su sè stesso e scrivere tali dati in un'opportuna area di memoria (diversa per ogni core). A questo punto ciascun core segnalerà all'host il completamento delle operazioni fissando a 1 il corrispondente flag di controllo.

Le variabili *my_row* e *my_col* contengono rispettivamente il numero di riga e di colonna di un core all'interno del workgroup. Successivamente inizializziamo un puntatore a char, *msg*, che punta ad una locazione di memoria in cui ciascun core scrive un messaggio contenente le sue coordinate e il proprio ID; infine utilizziamo un secondo puntatore a char, *done*, che viene utilizzato come flag di fine computazione. Notare che gli indirizzi 0x2000 e 0x7000 sono relativi ai 32KB di memoria di ciascun core, quindi non c'è nessun rischio di contesa delle risorse. La scelta degli indirizzi da utilizzare è del tutto arbitraria, l'unica cosa da ricordare è che la memoria locale di un core è composta da quattro banchi di 8KB ciascuno e inizia all'indirizzo 0x0 e termina a 0x00007FFF, di conseguenza il messaggio verrà scritto nel secondo banco, mentre il flag è posizionato nell'ultimo, dato che 8KB corrisponde all'indirizzo 0x00002000 e gli indirizzi di inizio dei quattro banchi

sono rispettivamente 0x0, 0x00002000, 0x00004000, 0x00006000. Infine la variabile *coreid* contiene l'id di 12 bit di un core ed è di tipo *e_core_id*; si tratta di un unsigned int definito nell'header file *e_lib.h* che abbiamo incluso all'inizio.

```
    unsigned my_row;
    unsigned my_col;
    char *msg, *done;
    msg = (char *) 0x2000;
    done = (char *) 0x7000;
    e_coreid_t coreid;
```

Con la funzione *e_get_coreid()*, ciascun core ottiene il proprio ID, che verrà successivamente incluso nel messaggio da consegnare all'host. La funzione *e_coords_from_coreid()* calcola a partire dall'ID del core passato come primo parametro le corrispondenti coordinate all'interno del workgroup, che vengono quindi salvate nel secondo e nel terzo parametro.

```
coreid = e_get_coreid();
e_coords_from_coreid(coreid, &my_row, &my_col);
```

Una volta ottenute tutte le informazioni rilevanti si scrive il messaggio per l'host con una semplice *fprintf()* alla locazione di memoria puntata dalla variabile *msg* e si segnala la fine della computazione impostando al valore 1 il flag *msg*:

```
sprintf (msg, "Hello World from core 0x%03x, with relative coords %u, %u",
    coreid, my_row, my_col);
*done = 1;
return EXIT_SUCCESS;
```

Il codice relativo all'host è decisamente più lungo, ma non necessariamente più complicato dato che deve compiere alcune operazioni standard che non variano da applicazione ad applicazione:

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
#include <unistd.h>
#include <e-hal.h>

int main(int argc, char **argv){
    e_platform_t platform;
    e_epiphany_t dev;
    char emsg[128];
    int i,j;
    unsigned flag = 0;
    int done[16];

    e_init(NULL);
    e_reset_system();
    e_get_platform_info(&platform);
    e_open(&dev, 0, 0, platform.rows, platform.cols);

    for (i=0; i<platform.rows; i++)
        for(j=0; j<platform.cols; j++){
            e_write(&dev, i, j, 0x7000, &flag, sizeof(flag));
    }

    e_load_group("e_hello_world.elf", &dev, 0, 0, platform.rows,
        platform.cols, E_FALSE);

    e_start_group(&dev);

    for (i=0; i<platform.rows; i++){
        for (j=0; j<platform.cols;j++){
            while(1){
                e_read(&dev, i, j, 0x7000, &done[i*platform.cols+j],
                    sizeof(int));
                if(done[i*platform.cols+j])
```

```
                break;
            }
        }
    }

    for (i=0; i<platform.rows; i++)
        for(j=0; j<platform.cols; j++){
            e_read(&dev, i, j, 0x2000, &emsg, 128);
            printf("%s\n", emsg);
        }

    e_close(&dev);
    e_finalize();

    return 0;
}
```

Le variabili principali sono le seguenti:

- e_platform_t *platform*: il tipo e_platform_t può essere considerato come un oggetto interno al coprocessore Epiphany che può essere interrogato per estrappolare informazioni utili, ad esempio la versione del coprocessore attualmente in uso, il numero di core disponibili e il numero di righe e di colonne al massimo indicizzabili.
- e_epiphany_t *dev*: il tipo e_epiphany_t serve per fare riferimento al workgroup attualmente in uso, dunque tutte le volte che utilizzeremo la variabile dev faremo riferimento al workgroup attuale.
- char *emsg[128]*: si tratta di un buffer in cui caricare uno alla volta i messaggi ricevuti.
- unsigned *flag*: questa variabile inizializzata a 0 viene utilizzata per pulire il flag *done* di ciascun core visto nel codice relativo al device, in questo modo ad ogni avvio del programma siamo sicuri che questa variabile sia inizialmente nulla.

- *done[16]*: array locale che mantiene il valore del flag *done* di ciascun core.

Le seguenti quattro istruzioni sono presenti in tutti i programmi per la Parallella. La funzione *e_init()* serve per aprire una connessione con il chip Epiphany e per inizializzare tutte le variabili e le strutture di dati definite nell'header file *e-hal.h*. Prima di utilizzare il coprocessore è buona norma resettarlo, facendo attenzione che nessun altro programma sia in esecuzione sui core nel momento in cui questa istruzione viene eseguita. Successivamente si utilizza la funzione *e_get_platform_info()* per caricare tutte le informazioni riguardanti il device nella variabile di tipo *e_platform_t* passata come parametro. Infine prima di avviare i core, è necessario specificare il workgroup che si vuole utilizzare con la funzione *e_open()*, che prende in input come primo parametro la variabile in cui salvare il riferimento al workgroup, come secondo e terzo parametro il punto d'origine all'interno della griglia 2D e infine con gli ultimi due parametri si specifica la dimensione.

```
e_init(NULL);
e_reset_system();
e_get_platform_info(&platform);
e_open(&dev, 0, 0, platform.rows, platform.cols);
```

Per un avvio sicuro, è preferibile settare a 0 la variabile *done* di ciascun core presente all'indirizzo 0x7000. Per scrivere direttamente all'interno della memoria locale di ciascun core è possibile utilizzare la funzione definita nell' Epiphany host library *e_write()*. Il primo parametro è il workgroup all'interno del quale si vuole scrivere, dopodiché è necessario specificare il core all'interno del workgroup tramite le sue coordinate. Gli ultimi tre parametri sono: l'indirizzo di destinazione in cui si vuole scrivere, la sorgente e il numero di byte. In questo caso dunque, scriviamo il valore 0 presente nella variabile flag all'indirizzo 0x7000 del core (i,j). Dopo aver inizializzato le variabili di controllo nei vari core è possibile avviare l'esecuzione del coprocessore utilizzando la funzione *e_load_group()*, che carica l'eseguibile passato come primo parametro all'interno del workgroup passato come secondo parametro. Anche in questo caso non è obbligatorio utilizzare tutti i core del workgroup, ma è possibile utilizzarne un sottogruppo specificando le coordinate di partenza tramite la coppia (row, col) e la dimensione del rettangolo con la coppia (rows, cols). L'ultimo parametro è un booleano che permette di ritardare l'effettivo inizio

della computazione; se il suo valore è true, significa che non appena l'eseguibile è caricato i core inizieranno la loro esecuzione, se invece è uguale a false, la computazione inizierà solamente quando viene invocata funzione *e_start_group()*. Nel nostro esempio, poiché fissiamo a (0, 0) il punto di inizio e a (*platform.rows, platform.cols*) la dimensione, utilizziamo tutto il workgroup che coincide con la dimensione del chip. Un aspetto fondamentale della funzione *e_load_group()* è che l'eseguibile deve essere in formato ELF[1] (Executable and Linkable Format)

```
for (i=0; i<platform.rows; i++)
    for(j=0; j<platform.cols; j++){
        e_write(&dev, i, j, 0x7000, &flag, sizeof(flag));
}

e_load_group("e_hello_world.elf", &dev, 0, 0, platform.rows, platform.cols,
    E_FALSE);
e_start_group(&dev);
```

Ora che il device ha iniziato la sua computazione all'host non rimane altro che aspettare; esistono varie strategie per fare questo e noi adottiamo una delle più semplici: l'host legge ripetutamente a partire dal primo core se il flag *done* è stato attivato, in tal caso si esce dal ciclo più interno e si verifica se anche il core successivo ha terminato. La funzione utilizzata per leggere direttamente all'interno della memoria locale dei core è la *e_read(void *dev, unsigned row, unsigned col, off_t from_addr, void *buf, size_t size)*, che preso in input il workgroup e un core all'interno di esso, copia size byte a partire dall'indirizzo specificato da *from_addr* nella variabile locale *buf*. In questo caso vogliamo verificare se ciascun core ha settato il flag di fine computazione che avevamo fissato all'indirizzo 0x7000, salvando neil valore nella posizione corrispondente nell'array *done[]*. Questa tecnica è detta busy waiting. Non appena il coprocessore ha terminato, si procede a leggere il messaggio scritto da ciascun core nello stesso modo: si utilizzano due loop per leggere in modo ordinato in ciascun core a partire dall'indirizzo 0x2000, dopodiché si stampa il messaggio con una printf. Una volta terminate tutte le operazioni si utilizza

[1]Formato file standard per eseguibili.

la funzione *e_close()*, con la quale si chiude il workgroup aperto con la funzione *e_open()* e si chiude la connessione con il chip Epiphany invocando la funzione *e_finalize()*.

```
for (i=0; i<platform.rows; i++){
    for (j=0; j<platform.cols;j++){
        while(1){
            e_read(&dev, i, j, 0x7000, &done[i*platform.cols+j], sizeof(int));
            if(done[i*platform.cols+j])
                break;
        }
    }
}

for (i=0; i<platform.rows; i++)
    for(j=0; j<platform.cols; j++){
        e_read(&dev, i, j, 0x2000, &emsg, 128);
        printf("%s\n", emsg);
    }

e_close(&dev);
e_finalize();

return 0;
```

L'output del programma "Hello world" è mostrato in figura 3.1.

3.2.1 Compilazione ed esecuzione

Di seguito viene riportato lo script utilizzato per la compilazione dei sorgenti.

```
#!/bin/bash

ESDK=${EPIPHANY_HOME}
ELIBS="-L ${ESDK}/tools/host/lib"
EINCS="-I ${ESDK}/tools/host/include"
```

```
ELDF=${ESDK}/bsps/current/fast.ldf

# Build HOST side application
gcc hello_world.c -o hello_world.elf ${EINCS} ${ELIBS} -le-hal -le-loader
    -lpthread

# Build DEVICE side program
e-gcc -T ${ELDF} e_hello_world.c -o e_hello_world.elf -le-lib
```

Da notare è il nome del sorgente destinato al device, che deve coincidere con il nome utilizzato nella funzione *e_load_group()*. Per eseguire il programma è sufficiente eseguire il file *hello_world.elf*. Per compilare i sorgenti sono disponibili tre linker script:

- legacy.ldf: è il più lento delle tre alternative in quanto sia il codice che i dati vengono posizionati all'interno della memoria condivisa.
- fast.ldf: questo linker script è simile al precedente a parte per il fatto che il codice relativo alle librerie viene posizionato nella memoria esterna.
- internal.ldf: si tratta del linker script che garantisce le migliori prestazioni possibili, poiché tutto viene memorizzato nella memoria locale dei core, a scapito però di una memoria ridotta.

La scelta del linker script da utilizzare dipende fortemente dal tipo di applicazione che si intende realizzare: nel caso in cui l'obbiettivo sia raggiungere la massima efficienza possibile è preferibile utilizzare l'ultima configurazione, se invece si vuole sfruttare tutta la memoria presente in ciascun core è consigliabile utilizzare il primo script. In tutti i programmi qui illustrati assumeremo sempre di usare la terza configurazione; questo significa che potremo scrivere nella memoria condivisa a partire dalla primo byte, cosa non vera se si utilizza il linker script fast.ldf, in cui i primi 16MB dei 32MB totali della memoria condivisa sono riservati alle librerie.

```
cardace@parallella: ~/myExample/parallelHello
cardace@parallella:~/myExample/parallelHello$ ./run.sh
Hello World from core 0x808, with relative coords 0, 0
Hello World from core 0x809, with relative coords 0, 1
Hello World from core 0x80a, with relative coords 0, 2
Hello World from core 0x80b, with relative coords 0, 3
Hello World from core 0x848, with relative coords 1, 0
Hello World from core 0x849, with relative coords 1, 1
Hello World from core 0x84a, with relative coords 1, 2
Hello World from core 0x84b, with relative coords 1, 3
Hello World from core 0x888, with relative coords 2, 0
Hello World from core 0x889, with relative coords 2, 1
Hello World from core 0x88a, with relative coords 2, 2
Hello World from core 0x88b, with relative coords 2, 3
Hello World from core 0x8c8, with relative coords 3, 0
Hello World from core 0x8c9, with relative coords 3, 1
Hello World from core 0x8ca, with relative coords 3, 2
Hello World from core 0x8cb, with relative coords 3, 3
```

Figura 3.1: Output hello world su Parallella

3.3 Prodotto scalare tra vettori con la Parallella

In questo secondo esempio vedremo come implementare il prodotto tra due vettori sulla Parallella Board. Il programma è stato realizzato a partire da quello discusso in [10]. Per semplicità il prodotto è calcolato tra due vettori di dimensione N, entrambi inizializzati con valori crescenti da 0 a N-1. Se N=64, è molto semplice verificare la correttezza del programma, poiché il prodotto scalare nel nostro esempio consiste nell'applicare la seguente formula:

$$\sum_{i=0}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \tag{3.1}$$

3.3.1 Header file

L'header file chiamato *common.h* contiene due valori che vengono utilizzati sia nel sorgente dell'host sia in quello per il chip Epiphany:

```
#define N    64
#define CORES 16
```

N rappresenta la dimensione dei due array, mentre CORES esplicita il numero di core da utilizzare.

3.3.2 Sorgente host

```
#include <stdlib.h>
#include <stdio.h>
#include <e-hal.h>
#include "common.h"

#define RESULT 85344

int main(int argc, char *argv[]){
    e_platform_t platform;
    e_epiphany_t dev;

    int a[N], b[N], c[CORES];
    int done[CORES];
    int sum;
    int i,j;
    int sections = N/CORES;
    unsigned clr = 0;

    e_init(NULL);
    e_reset_system();
    e_get_platform_info(&platform);
    e_open(&dev, 0, 0, platform.rows, platform.cols);

    for (i=0; i<N; i++){
        a[i] = i;
        b[i] = i;
    }

    e_load_group("e_task.elf", &dev, 0, 0, platform.rows, platform.cols,
        E_FALSE);
```

```
for (i=0; i<platform.rows; i++){
    for (j=0; j<platform.cols;j++){
        e_write(&dev, i, j, 0x2000, &a[(i*platform.cols+j)*sections],
            sections*sizeof(int));
        e_write(&dev, i, j, 0x4000, &b[(i*platform.cols+j)*sections],
            sections*sizeof(int));
        e_write(&dev, i, j, 0x7000, &clr, sizeof(clr));
    }
}

e_start_group(&dev);

for (i=0; i<platform.rows; i++){
    for (j=0; j<platform.cols;j++){
        while(1){
            e_read(&dev, i, j, 0x7000, &done[i*platform.cols+j],
                sizeof(int));
            if(done[i*platform.cols+j])
            break;
        }
    }
}


for (i=0; i<platform.rows; i++){
    for (j=0; j<platform.cols;j++){
        e_read(&dev, i, j, 0x6000, &c[i*platform.cols+j], sizeof(int));
    }
}

sum=0;
for (i=0; i<CORES; i++){
    sum += c[i];
```

```
    }

    printf("Sum: %d\n",sum);
    e_close(&dev);
    e_finalize();

    if(sum==RESULT){
        return EXIT_SUCCESS;
    }
    else{
        return EXIT_FAILURE;
    }
}
```

L'idea dell'algoritmo è molto semplice: si inizializzano entrambi i vettori a e b con gli stessi valori, dopodichè l'host assegna $N/CORES$ elementi di entrambi gli array a tutti i core; in particolare gli elementi dell'array a vengono scritti a partire dalla posizione 0x2000, mentre i corrispondenti elementi di b a partire da 0x4000. La somma parziale calcolata da ciascun core viene scritta invece all'indirizzo 0x6000. Come per l'esempio illustrato nel paragrafo 3.2, l'host aspetta che ciascun core finisca la propria computazione prima di andare a sommare i risultati parziali per ottenere la somma finale.

3.3.3 Sorgente Epiphany

Il codice sorgente del device è molto semplice, in quanto ciascun core deve limitarsi ad effettuare una sommatoria con i valori caricati precedentemente dall'host e segnalare la fine della computazione.

```
#include <stdio.h>
#include <stdlib.h>
#include "e-lib.h"
#include "common.h"

int main(void)
```

```
{
    unsigned *a, *b, *c, *d;
    int i;

    a = (unsigned *) 0x2000; //pointer to first array
    b = (unsigned *) 0x4000; //pointer to second array
    c = (unsigned *) 0x6000; //pointer to final result
    d  = (unsigned *) 0x7000; //pointer to flag

    *c=0;

    for (i=0; i<N/CORES; i++){
        *c += a[i] * b[i];
    }

    *d = 1;
    return EXIT_SUCCESS;
}
```

3.4 Prodotto tra una matrice e uno scalare

In questo terzo e ultimo esempio vedremo come realizzare il prodotto tra una matrice e uno scalare, cosa che a prima vista sembrerebbe banale ma che in realtà ci permette di illustrare come utilizzare la DMA. La strategia è la seguente: l'host inizializza una matrice quadrata e lo scalare, per poi copiare tali dati nella memoria esterna. A questo punto il core 0 copia in locale gli operandi per effettuare la moltiplicazione prelevandoli dalla memoria condivisa grazie al supporto della DMA. Una volta terminata la computazione, il core 0 scrive nella memoria condivisa il risultato finale in modo da renderlo accessibile all'host. Effettuare localmente la computazione permette ai core del chip Epiphany di accedere alla memoria esterna (che è molto più lenta) solo all'inizio, per ottenere i dati, e alla fine per condividere i risultati.

3.4.1 struct.h

```
#ifndef __STRUCT__
#define __STRUCT__
#define DEFAULT 3
#define SIZE 16

typedef struct {
    int grid[SIZE * SIZE];
    int scalar;
} shared_buf_t;

typedef struct {
    void *pBase;
    int *pGrid;
    int *pScalar;
} shared_buf_ptr_t;

#endif
```

Questo header file contiene le strutture dati principali del programma. In particolare *shared_buf_t* contiene la griglia da inizializzare e lo scalare con cui effettuare la moltiplicazione. Il tipo *shared_buf_ptr_t* contiene invece tre puntatori che vengono utilizzati dal chip Epiphany: *pBase* contiene l'indirizzo di inizio della memoria condivisa, *pGrid* punta al primo elemento della griglia e infine *pScalar* serve per ottenere lo scalare.

3.4.2 Sorgente host

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <e-hal.h>
```

```
#include "struct.h"
#include <stddef.h>

shared_buf_t Mailbox;

int main(int argc, char *argv[]){
    e_platform_t epiphany;
    e_epiphany_t dev;
    e_return_stat_t result;
    e_mem_t DRAM;
    unsigned int msize;
    msize = 0x00400000;
    int i,j;
    size_t sz;
    unsigned int addr;

    e_init(NULL);
    e_reset_system();
    e_get_platform_info(&epiphany);

    if ( e_alloc(&DRAM, 0x0000000, msize) ){
        fprintf(stderr, "\nERROR: Can't allocate Epiphany DRAM!\n");
        exit(1);
    }

    if ( argc == 2 ) {
        Mailbox.scalar = atoi(argv[1]);
    }
    else
        Mailbox.scalar = DEFAULT;

    addr = offsetof(shared_buf_t, scalar);
    sz = sizeof(Mailbox.scalar);
```

```
    e_write(&DRAM, 0, 0, addr, (void *) &Mailbox.scalar, sz);

    for(i=0; i<SIZE; i++)
        for(j=0; j<SIZE; j++)
            Mailbox.grid[i*SIZE+j] = rand() % 100;

  addr = offsetof(shared_buf_t, grid);
  sz = sizeof(Mailbox.grid);
  e_write(&DRAM, 0, 0, addr, (void *) Mailbox.grid, sz);
  e_open(&dev,0,0,1,1);
  e_reset_group(&dev);
  result = e_load("e_hello_world.elf",&dev,0,0,E_FALSE);
  if (result != E_OK){
      fprintf(stderr,"Error Loading the Epiphany Application 1 %i\n",
          result);
  }

  e_start_group(&dev);
  usleep(30000);

  printf("extracting from core memory \n");

  e_read(&DRAM,0,0,0,(void *) &Mailbox, sizeof(Mailbox));
  for(i=0; i<SIZE; i++){
      for(j=0; j<SIZE; j++)
          printf("%d ",Mailbox.grid[i*SIZE+j]);
      printf("\n");
  }
  e_close(&dev);
  e_finalize();
  fprintf(stderr,"demo complete \n ");
  return 0;
 }
```

La prima parte è molto simile a quella degli esempi precedenti. Tuttavia in questo caso, poiché vogliamo utilizzare la memoria condivisa, utilizziamo la funzione *e_alloc()* per allocare dello spazio all'interno di essa. Il primo parametro di tipo *e_mem_t* è la variabile utilizzata per fare riferimento alla memoria condivisa, il secondo parametro rappresenta l'offset all'interno della quale si vuole iniziare a scrivere, e infine il terzo parametro consiste nel numero di byte da allocare. Si ricorda che possiamo utilizzare tutta la memoria esterna grazie al fatto che assumiamo di utilizzare il linker script denominato internal.ldf.

```
if ( e_alloc(&DRAM, 0x0000000, msize) ){
    fprintf(stderr, "\nERROR: Can't allocate Epiphany DRAM!\n");
    exit(1);
}
```

Successivamente, si inizializza la struttura di tipo *shared_buf_t* chiamata *Mailbox* e si provvede a copiare tale struttura nell'area condivisa ottenuta precedentemente. Per fare questo si utilizza sempre la funzione *e_write()*, in cui a differenza dell'esempio 3.3.2, il primo parametro è di tipo *e_mem_t* e non *e_epiphany_t*, proprio perché vogliamo scrivere nella memoria condivisa e non all'interno della memoria locale di un core, di conseguenza il secondo e il terzo parametro (che prima rappresentavo il numero del core), vengono ignorati. I restanti parametri rimangono invariati e sono l'offset da cui iniziare a scrivere, la sorgente e il numero di byte rispettivamente.

```
addr = offsetof(shared_buf_t, grid);
sz = sizeof(Mailbox.grid);
e_write(&DRAM, 0, 0, addr, (void *) Mailbox.grid, sz);
```

A questo punto tutta la *Mailbox* è stata caricata, quindi viene lanciato il programma nel core 0 che si occupa dell'effettiva computazione mentre l'host attende qualche secondo, prima di andare ad utilizzare la funzione *e_read()*, in cui si utilizza nuovamente come primo parametro la variabile di tipo *e_mem_t*, dato che vogliamo prelevare il risultato dalla memoria condivisa.

```
e_read(&DRAM,0,0,0,(void *) &Mailbox, sizeof(Mailbox));
```

3.4.3 Sorgente Epiphany

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "e_lib.h"
#include "struct.h"
#include <stddef.h>

void data_copy(e_dma_desc_t *dma_desc, void *dst, void *src){

    e_dma_wait(E_DMA_0);
    dma_desc->src_addr = src;
    dma_desc->dst_addr = dst;
    e_dma_start(dma_desc, E_DMA_0);
    e_dma_wait(E_DMA_0);

    return;
}

int main(void){
    int i,j;
    int *p;
    volatile shared_buf_ptr_t Mailbox;
    int localMatrix[SIZE*SIZE];
    e_dma_desc_t shared2local;
    e_dma_desc_t local2shared;
    int scalar;

    Mailbox.pBase = (void *) e_emem_config.base;
    Mailbox.pGrid = Mailbox.pBase + offsetof(shared_buf_t, grid);
    Mailbox.pScalar = Mailbox.pBase + offsetof(shared_buf_t, scalar);
```

```
    scalar = *Mailbox.pScalar;

    e_dma_wait(E_DMA_0);
    shared2local.config = E_DMA_WORD | E_DMA_MASTER | E_DMA_ENABLE;
    shared2local.inner_stride = (0x04 << 16) | 0x04;
    shared2local.count = (SIZE << 16) | SIZE;
    shared2local.outer_stride = (0x04 << 16) | 0x04;

    e_dma_wait(E_DMA_0);
    local2shared.config = E_DMA_WORD | E_DMA_MASTER | E_DMA_ENABLE;
    local2shared.inner_stride = (0x04 << 16) | 0x04;
    local2shared.count = (SIZE << 16) | SIZE;
    local2shared.outer_stride = (0x04 << 16) | 0x04;

    data_copy(&shared2local, localMatrix, Mailbox.pGrid);

    for(i=0; i<SIZE; i++)
        for(j=0; j<SIZE; j++)
            localMatrix[i*SIZE+j] *= scalar;

    data_copy(&local2shared, Mailbox.pGrid, localMatrix);

    return EXIT_SUCCESS;
}
```

La funzione *data_copy()* trasferisce i dati dalla sorgente alla destinazione in accordo al descrittore della DMA passato come primo parametro. Prima di invocare la *e_dma_start()* per avviare il trasferimento, si utilizza la funzione *e_dma_wait()*, che blocca l'esecuzione del programma fintanto che una delle due DMA non è disponibile (come spiegato nel paragrafo 2.2.3 ogni core possiede due DMA) . Inizialmente il core 0 ottiene l'indirizzo di inizio della memoria condivisa sfruttando un oggetto globale chiamato *e_emem_config*. Questo puntatore vene salvato in *Mailbox.pBase*, che viene successivamente utilizzato come base a cui aggiungere un offset per ottenere i riferimenti agli oggetti inizializzati

dall'host.

```
Mailbox.pBase = (void *) e_emem_config.base;
Mailbox.pGrid = Mailbox.pBase + offsetof(shared_buf_t, grid);
Mailbox.pScalar = Mailbox.pBase + offsetof(shared_buf_t, scalar);
```

Il vantaggio di questa tecnica è quello di mantenere in un'unica struttura di dati tutti i puntatori necessari per accedere in lettura e scrittura agli oggetti condivisi con il processore ARM. Prima di invocare la funzione *data_copy()* spiegata precedentemente è necessario inizializzare i descrittori nel modo opportuno. La variabile *shared2local* è il descrittore da utilizzare per trasferire la matrice dalla memoria condivisa a quella locale, in particolare all'interno di *localMatrix*. Il flag *E_DMA_WORD* serve per indicare alla DMA di trasferire una word (4 bytes) alla volta, *E_DMA_ENABLE* attiva il canale scelto, e infine con *E_DMA_MASTER* richiediamo che la DMA lavori in modalità master, ovvero che completi autonomamente il trasferimento dalla sorgente alla destinazione. Non specificare questo flag significa assumere che la DMA lavori in modalità slave, in cui il trasferimento dei dati viene governato da un componente esterno (per maggiori dettagli fare riferimento a [6]). L'hardware della Parallella propone una DMA in grado di trasferire dati seguendo un layout 2D, questo significa che dobbiamo gestire due loop: quello più interno può essere considerato come le colonne della matrice da copiare, mentre quello più esterno può essere utilizzato per indicizzare le righe della matrice. Il campo *inner_stride* serve a specificare il salto da eseguire nella sorgente (i 16 bit meno significativi) e nella destinazione (i 16 bit più significativi) dopo ogni iterazione del ciclo interno; nel nostro esempio avanziamo di 4 byte in entrambi dato che abbiamo utilizzato il flag E_DMA_WORD. L'idea intuitiva è quella di avere due puntatori, uno per la sorgente e uno per la destinazione, che ad ogni iterazione vengono incrementati.

```
shared2local.config = E_DMA_WORD | E_DMA_MASTER | E_DMA_ENABLE;
shared2local.inner_stride = (0x04 << 16) | 0x04;
shared2local.count = (SIZE << 16) | SIZE;
shared2local.outer_stride = (0x04 << 16) | 0x04;
```

Il campo *count* contiene invece il numero di cicli da effettuare nel loop interno nei pri-

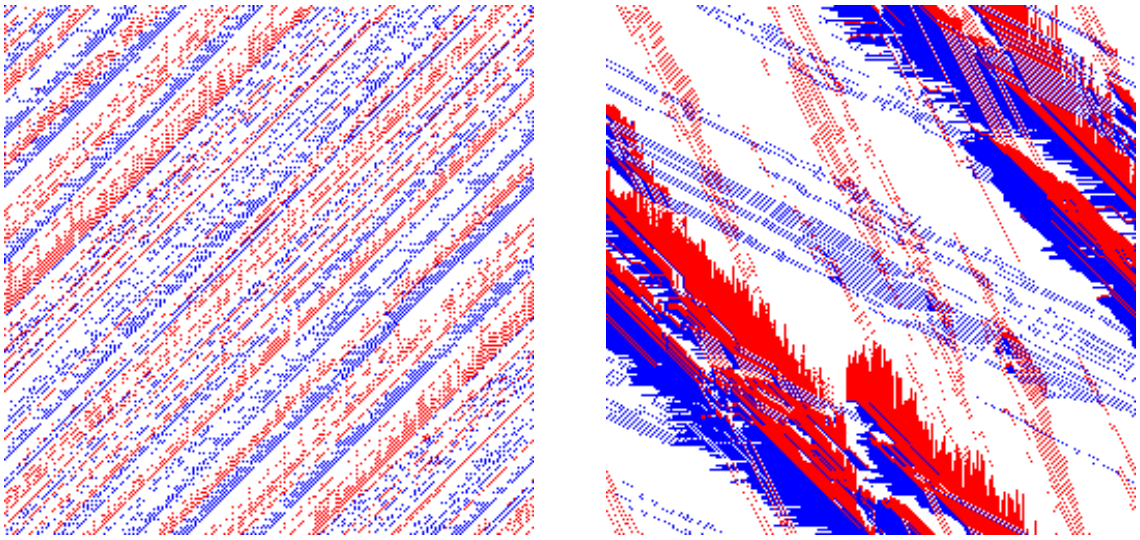
mi 16 bit e il numero di cicli nel loop esterno nei restanti bit; al termine di tutte le iterazioni previste nel ciclo interno il contatore del loop esterno viene decrementato di uno mentre quello interno riparte dal valore iniziale. Nell'esempio viene trasferita una matrice quadrata $SIZE \times SIZE$, dove ogni elemento è un intero di 4 byte, per questo motivo entrambi i loop sono settati a SIZE iterazioni. Infine come per il loop interno, è necessario specificare anche il salto da effettuare alla fine del ciclo esterno. Anche in questo caso il puntatore della sorgente e della destinazione vengono spostati in avanti di 4 byte, poiché la matrice bidimensionale è memorizzata in memoria come se fosse un array unidimensionale, e dunque per accedere alla riga successiva basta spostarsi avanti di un intero. Un aspetto importante da riportare è che nell'ultima iterazione prevista dal ciclo più interno, l'incremento ai due puntatori non viene applicato, questo è il motivo per cui è necessario avanzare anche quando il contatore del ciclo esterno viene decrementato. Il descrittore per trasferire la matrice aggiornata dalla memoria locale a quella condivisa è del tutto identico, poiché la matrice locale del core 0 ha la stessa dimensione di quella condivisa dal processore ARM. Tuttavia si tratta solo di un caso particolare, di solito infatti, tutti i sedici core sono coinvolti nella computazione e ciascuno di essi lavora solo su una porzione dei dati presenti nella memoria condivisa. Una volta inizializzati correttamente i due descrittori si può procedere ad avviare il trasferimento in locale della matrice, effettuare la moltiplicazione con lo scalare e infine scrivere il risultato nella memoria condivisa.

Capitolo 4

Modello di Biham-Middleton-Levine

4.1 Descrizione del modello

Il modello di Biham-Middleton-Levine consiste in un automa cellulare auto organizzante per la simulazione di un flusso di traffico formalizzato da Ofer Biham, A. Alan Middleton e Dov Levine nel 1992 [11]. Consiste in una griglia 2D in cui ogni cella può essere vuota oppure contenere una macchina di colore rosso o blu. Il colore della macchina determina il movimento che essa può eseguire: le macchine di colore blu possono spostarsi di una sola posizione verso il basso, mentre quella di colore rosso solo verso destra. Il comportamento dell'automa dipende sia dalla dimensione della griglia che dalla densità iniziale di macchine presenti, e dopo una fase iniziale, può trovarsi nello stato finale di blocco, in cui nessuna macchina sulla griglia ha la possibilità di muoversi, oppure in uno stato finale di traffico libero, in cui vi è un flusso costante di veicoli. In particolare sembrerebbe che la percentuale di soglia di passaggio da uno stato all'altro sia intorno al 33%, e per densità vicine alla soglia si possono osservare delle fasi intermedie in cui l'automa alterna situazioni di blocco a fasi di traffico scorrevole, sia in modo periodico che in modo disordinato. Nella figura 4.1 sono presenti due esempi che mostrano lo stato dell'automa dopo 1024 step partendo da densità diverse. Nonostante il modello sia facile da comprendere, ben poche proprietà sono state dimostrate rigorosamente: nel 2005 è stato provato in [12] che con una densità di macchine iniziale vicino ad uno il sistema raggiunge sempre lo stato di blocco, mentre nel 2006 è stato dimostrato che il modello



(a) Densità iniziale 0.2

(b) Densità iniziale 0.7

Figura 4.1: Esempi di griglie 256x256 dopo 1024 passi

raggiunge sempre uno stato di flusso libero se sono presenti meno di N/2 macchine, dove N è la dimensione del lato del quadrato della griglia 2D [13].

Capitolo 5

Algoritmo sequenziale

In questo capitolo illustreremo un possibile algoritmo sequenziale per il simulazione del modello di Biham-Middleton-Levine. Ad ogni passo della simulazione lo stato dell'automa viene salvato su un file esterno nel formato PPM (Portable PixMap); questa operazione non è tuttavia necessaria ai fini della correttezza del programma, ma ci permette di vedere l'intera simulazione tramite un'animazione. I passi dell'algoritmo sono i seguenti:

1. Inizializzare la griglia con una densita P di veicoli.
2. Eseguire uno step orizzontale.
3. Eseguire uno step verticale.
4. Copiare lo stato del modello su un file esterno (opzionale).

5.1 Versione sequenziale

5.1.1 Variabili principali

- *SIZE*: costante che rappresenta la dimensione della griglia.
- *cur*: rappresenta la griglia corrente e può assumere il valore 0 o 1.

- *grid[2][SIZE][SIZE]*: vettore di due griglie in cui la prima corrisponde allo stato corrente, mentre la seconda viene utilizzata per ottenere lo stato successivo.
- *nsteps*: numero di step da simulare.
- *prob*: probabilità iniziale che una cella sia occupata da un veicolo.
- *TB*: char che rappresenta un veicolo che può effettuare uno spostamento dall'alto verso il basso (colore rosso).
- *LR*: char che rappresenta un veicolo che può effettuare uno spostamento da sinistra a destra (colore blu).
- *EMPTY*: char che rappresenta una cella vuota.

5.1.2 Algoritmo

Il main è molto semplice: dopo l'inizializzazione della griglia tramite la funzione *setup()*, si eseguono *nsteps* passi della simulazione, in cui si effettua uno step orizzontale nel quale si muovono solo le macchine di colore blu (se possibile), seguito da uno step verticale in cui spostano solo le macchine di colore rosso. Prima di iniziare un nuovo ciclo è possibile eventualmente effettuare il dump della matrice con la funzione *dump_cur()*.

```
int main( int argc, char* argv[] ) {
    const int buflen = 255;
    int s;
    char buf[buflen];
    float prob = 0.2;
    int nsteps = 256;
    if ( argc > 1 )
        nsteps = atoi(argv[1]);
    if ( argc > 2 )
        prob = atof(argv[2]);

    setup(prob);
```

```
    for (s=0; s<nsteps; s++) {
        horizontal_step();
        cur = 1 - cur;
        vertical_step();
        snprintf(buf, buflen, "out%05d.ppm", s);
        dump_cur(buf);
        cur = 1 - cur;
    }
    return 0;
}
```

La funzione *setup()* sfrutta la funzione *rand01()* che genera valori compresi tra zero e uno e verifica se questi sono minori della probabilità p, di conseguenza l'intera griglia avrà una densità iniziale di veicoli pari a p.

```
int rand01( float p ){
    return ((rand() / (float)RAND_MAX) < p);
}

void setup( float p ){
    int i, j;
    cur = 0;
    for ( i=0; i<SIZE; i++ ) {
        for ( j=0; j<SIZE; j++) {
            if ( rand01( p ) ) {
                grid[cur][i][j] = ( rand01(0.5) ? LR : TB );
        } else
            grid[cur][i][j] = EMPTY;
        }
    }
}
```

Nella funzione *horizontal_step()* tutti i veicoli di tipo *LR* effettuano uno spostamento da sinistra verso se la cella alla loro destra è vuota. In particolare, se la cella corrente

è vuota e la cella alla sinistra è occupata da una macchina di tipo LR, nella iterazione successiva la casella libera deve essere occupata da questa macchina. Altrimenti, se la cella corrente è occupata da un veicolo LR e la cella adiacente a destra risulta vuota, la casella corrente al passo successivo deve essere libera. Se nessuno di questi casi viene verificato, significa che lo stato della cella deve rimanere invariato. Notare come questo modello esprime l'andamento naturale di una coda di traffico: le macchine non si muovono tutte contemporaneamente sulla destra, ma al contrario si sposta inizialmente la prima macchina della coda (quella più a sinistra per ogni riga) e successivamente tutte le altre.

```
void horizontal_step( ) {
    int i, j;
    const int next = 1 - cur;
    for (i=0; i<SIZE; i++) {
        for (j=0; j<SIZE; j++) {
            const int left = (j > 0 ? j-1 : SIZE-1);
            const int right = (j < SIZE-1 ? j+1 : 0);
            if ( grid[cur][i][j] == EMPTY && grid[cur][i][left] == LR ) {
                grid[next][i][j] = LR;
            } else {
                if ( grid[cur][i][j] == LR && grid[cur][i][right] == EMPTY ) {
                    grid[next][i][j] = EMPTY;
                } else {
                    grid[next][i][j] = grid[cur][i][j];
                }
            }
        }
    }
}
```

La funzione *vertical_step()* è simmetrica.

```
void vertical_step( void ){
    int i, j;
```

```
    const int next = 1 - cur;
    for (i=0; i<SIZE; i++) {
        const int top = (i > 0 ? i-1 : SIZE-1);
        const int bottom = (i < SIZE-1 ? i+1 : 0);
        for (j=0; j<SIZE; j++) {
            if ( grid[cur][i][j] == EMPTY && grid[cur][top][j] == TB ) {
                grid[next][i][j] = TB;
            } else {
                if ( grid[cur][i][j] == TB && grid[cur][bottom][j] == EMPTY) {
                    grid[next][i][j] = EMPTY;
                } else {
                    grid[next][i][j] = grid[cur][i][j];
                }
            }
        }
    }
}
```

Infine, la *dump_cur()* invocata opzionalmente alla fine di ogni passo della simulazione, scrive su un file esterno lo stato della griglia corrente secondo il formato PPM.

```
void dump_cur( const char* filename ){
    int i, j;
    FILE *out = fopen( filename, "w" );
    if ( NULL == out ) {
        printf("Cannot create file %s\n", filename );
        abort();
    }
    fprintf(out, "P6\n");
    fprintf(out, "%d %d\n", SIZE, SIZE);
    fprintf(out, "255\n");

    for (i=0; i<SIZE; i++) {
        for (j=0; j<SIZE; j++) {
```

```
            switch( grid[cur][i][j] ) {
                case 0:
                    fprintf(out, "%c%c%c", 255, 255, 255);
                    break;
                case 1:
                    fprintf(out, "%c%c%c", 0, 0, 255);
                    break;
                case 2:
                    fprintf(out, "%c%c%c", 255, 0, 0);
                    break;
                default:
                    printf("Error: unknown cell state %u\n", grid[cur][i][j]);
                    abort();
            }
        }
    }
    fclose(out);
}
```

Capitolo 6

Algoritmi Paralleli

In questo capitolo vedremo due possibili algoritmi paralleli che implementano il modello di Biham-Middleton-Levine. Nella prima versione viene utilizzata una distribuzione a righe, mentre nella seconda una distribuzione a blocchi.

6.1 Distribuzione a righe

In questo algoritmo ogni core possiede un blocco di *_Nelem/_Ncores* righe di dimensione *_Nelem*; questo significa che ciascun core deve comunicare con i suoi vicini poiché all'iterazione $i + 1$ lo stato della prima riga e dell'ultima dipendono da quello dei vicini all'iterazione i-esima. La figura 6.1 riassume questa idea.

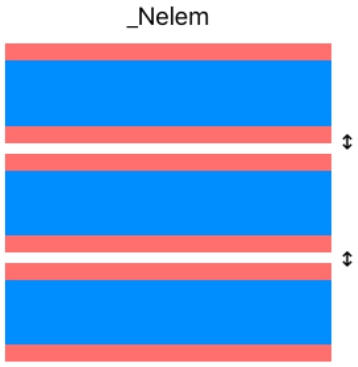


Figura 6.1: Distribuzione a righe (in rosso gli elementi che vengono scambiati).

6.1.1 Strutture dati

Prima di illustrare l'algoritmo parallelo, è opportuno analizzare le strutture dati definite nell'header file *struct.h* per facilitare la comunicazione host-device. L'idea ricalca lo schema utilizzato nell'esempio 3.4: l'host si occupa dell'inizializzazione della matrice iniziale per poi copiarla nella memoria condivisa per renderla accessibile a tutti i core. Ciascun core ha la responsibilità di prelevare una porzione della matrice e lavorare localmente con essa comunicando con i core adiacenti per effettuare la simulazione. Al termine di ogni passo intermedio, i core copiano il frame prodotto nella memoria esterna e aspettano che l'host dia il via libera per iniziare l'iterazione successiva.

```
#define _Ncores 16
#define _Nelem 256

typedef struct {
    unsigned coreID;
    unsigned corenum;     //number of the core in a workgroup
    unsigned row;         //core row coordinates
    unsigned col;         //core column coordinates
    unsigned row_next;    //row coordinate of the (corenum+1) % 16 core
    unsigned col_next;    //col coordinate of the (corenum+1) % 16 core
    unsigned row_previous;  //row coordinate of the (corenum-1) % 16 core
    unsigned col_previous; //core coordinate of the (corenum-1) % 16 core
    char _grid[2][_Nelem* (_Nelem/_Ncores)]; //Local portion of the whole
        grid to update
} core_t;

typedef struct {
    char grid[_Nelem*_Nelem];
    int nsteps;
} shared_buf_t;

typedef struct {
```

```
    void *pBase;
    char *pGrid;
    int *pNsteps;
} shared_buf_ptr_t;
```

_Ncores e *_Nelem* rappresentano rispettivamente il numero di core da utilizzare per la simulazione e la dimensione della griglia dei veicoli. Ciascun core possiede una variabile chiamata *me* di tipo *core_t*. Questa struttura ha la scopo di contenere tutte le informazioni utili relative al core, come il suo ID o le coordinate dei suoi vicini, in modo da semplificare la comunicazione con essi. Come per l'esempio 3.4, utilizziamo una *Mailbox* di tipo *shared_buf_t* che contiene tutti i dati che devono essere condivisi tra il processore ARM e il chip Epiphany, il quale utilizzerà una struttura di tipo *shared_buf_ptr_t* che contiene dei puntatori agli elementi della *Mailbox*.

6.1.2 Sorgente host

```
#include <stdlib.h>
#include <stdio.h>
#include <e-hal.h>
#include "struct.h"
#include <stddef.h>
#include <sys/time.h>

e_platform_t platform;
shared_buf_t Mailbox;
e_epiphany_t dev;

char EMPTY = 0;       //empty cell
char LR = 1;          //left-to-right moving vehicle
char TB = 2;          //top-to-bottom moving vehicle

unsigned clr = (unsigned)0x00000000;
unsigned flag = (unsigned)0x00000001;
```

```
int main(int argc, char *argv[]){
    int i,j;
    char buf[255];
    e_mem_t DRAM;
    unsigned int msize;
    unsigned int addr;
    int result;
    size_t sz;
    float prob = 0.3;
    int nsteps = 1024;
    msize = 0x00400000;

    if ( argc > 1 ) {
        nsteps = atoi(argv[1]);
    }
    if ( argc > 2 ) {
        prob = atof(argv[2]);
    }

    //Initalize Epiphany device
    e_init(NULL);
    e_reset_system();                   //reset Epiphany
    e_get_platform_info(&platform);  //get platform information
```

L'inizio del programma è molto simile a tutti quelli visti fino ad ora: si inizializza il device, si effettua un reset e si ottengono le informazioni sul chip Epiphany. Le variabili *prob* e *nsteps* possono essere acquisite dall'utente e rappresentano rispettivamente la densità iniziale della griglia e il numero di passi da simulare. *clr* e *flag* servono invece per modificare il valore delle variabili utili a garantire la sincronizzazione tra host e device.

```
if ( e_alloc(&DRAM, 0x00000000, msize) ){
    fprintf(stderr, "\nERROR: Can't allocate Epiphany DRAM!\n");
```

```
    exit(1);
}

if ( e_open(&dev, 0, 0, platform.rows, platform.cols) ){
    fprintf(stderr, "\nERROR: Can't establish connection to Epiphany
        device!\n");
    exit(1);
}.
```

Prima di scrivere inizializzare la matrice, è necessario invocare la *e_alloc()* per richiedere l'utilizzo della memoria condivisa (in questo caso tutti i 32MB a partire dall'indirizzo 0x00). Tutti i successivi accessi a questa memoria verranno effettuati utilizzando la variabile *DRAM*. Dopodiché si definisce il workgroup nel solito modo.

```
    e_write(&dev, 0, 0, 0x7000, &clr, sizeof(clr));
    e_write(&dev, 0, 0, 0x7010, &clr, sizeof(clr));

    Mailbox.nsteps = nsteps;
    addr = offsetof(shared_buf_t, nsteps);
    sz = sizeof(Mailbox.nsteps);
    e_write(&DRAM, 0, 0, addr, (void *) &Mailbox.nsteps, sz);

    setup(prob);

    addr = offsetof(shared_buf_t, grid);
    sz = sizeof(Mailbox.grid);
    e_write(&DRAM, 0, 0, addr, (void *) Mailbox.grid, sz);

    snprintf(buf, 255, "out00000.ppm");
    dump_cur(buf, Mailbox.grid);

    fprintf(stderr, "Loading program on Epiphany chip...\n");
    result = e_load_group("e_traffic.elf", &dev, 0, 0, platform.rows,
        platform.cols, E_FALSE);
```

```
    if (result == E_ERR) {
        fprintf(stderr, "Error loading Epiphany program.\n");
        exit(1);
    }
    e_start_group(&dev);
```

A questo punto si può procedere con l'inizializzazione dei dati. Come prima cosa si azzerano le due variabili all'interno del core 0 posizionate all'indirizzo 0x7000 e 0x7010. Apparentemente la sincronizzazione avviene solamente tra il processore ARM e il core 0 del chip Epiphany, ma in realtà grazie all'utilizzo di barriere tutti i core procederanno alla stessa velocità, dunque nessun core può iniziare l'iterazione successiva prima che tutti gli altri abbiano finito quella corrente. Successivamente si inizializzano i dati da passare al coprocessore prima di scriverli nella memoria condivisa. Notare anche in questo caso l'uso differente della stessa funzione *e_write()*: per scrivere direttamente all'interno di un core il primo parametro è il riferimento al device, mentre per effettuare una scrittura all'interno dell'area in comune si usa la variabile *DRAM* ignorando le coordinate passate in input. Infine si effettua il dump dello stato interno dell'automa prima di iniziare la simulazione vera e propria caricando l'eseguibile nei core del workgroup.

```
    bml_go(&DRAM, nsteps);
```

```
void bml_go(e_mem_t *pDRAM, int nsteps) {
    unsigned int addr;
    size_t sz;
    int i,j,k;
    char buf[255];
    unsigned done = 0;

    for(k=1; k<nsteps; k++){
        while(1){
            e_read(&dev, 0, 0, 0x7010, &done, sizeof(unsigned));
            if (done == 1){
                done =0;
```

```
                e_write(&dev, 0, 0, 0x7010, &clr, sizeof(clr));
                break;
            }
        }
        addr = 0;
        sz = sizeof(Mailbox);
        e_read(pDRAM, 0, 0, addr, &Mailbox, sz);

        snprintf(buf, 255, "out%05d.ppm",k);
        dump_cur(buf, Mailbox.grid);
        e_write(&dev, 0, 0, 0x7000, &flag, sizeof(flag));
    }
 }
```

Una volta avviato il coprocessore viene invocata la funzione più importante: la *bml_go()*. Al suo interno, per *nsteps* volte, il processore ARM legge dall'indirizzo 0x7010 del primo core; solo quando il valore di questa variabile viene impostata ad uno il processore è sicuro che tutti i core abbiano terminato la computazione e può procedere a leggere l'intera griglia dalla memoria condivisa per copiarla in un file esterno. Dopodiché l'host segnala al device di procedere con la prossima iterazione cambiando il valore della variabile posizionata all'indirizzo 0x7000 del primo core, il quale sbloccherà tutti gli altri core bloccati. La funzione *dump_cur()* è uguale a quella dell'algoritmo sequenziale.

```
if (e_close(&dev)) {
    fprintf(stderr, "\nERROR: Can't close connection to Epiphany
        device!\n\n");
    exit(1);
}
if (e_free(&DRAM)){
    fprintf(stderr, "\nERROR: Can't release Epiphany DRAM!\n\n");
    exit(1);
}

e_finalize();
```

```
return EXIT_SUCCESS;
}
```

Dopo *nsteps* iterazioni la computazione è terminata, ed è necessario chiudere il workgroup e invocare le funzioni *e_free()* e *e_finalize()*, rispettivamente per liberare la memoria allocata precedentemente e per chiudere la connessione con il chip Epiphany.

6.1.3 Sorgente Epiphany

Possiamo riassumere i passi principali che ogni core deve eseguire nel seguente modo:

1. Eseguire uno step orizzontale.
2. Prelevare l'ultima riga del core precedente.
3. Prelevare la prima riga del core successivo.
4. Eseguire lo step verticale.
5. Segnalare all'host che un passo della simulazione è stato completato (solo il primo core).

```
#include <stdio.h>
#include <stdlib.h>
#include "e-lib.h"
#include "struct.h"
#include <string.h>
#include <stddef.h>

char EMPTY = 0;
char LR = 1;
char TB = 2 ;

core_t me;
int cur = 0;
int nrows= _Nelem / _Ncores;
```

```
int main(void){
    volatile shared_buf_ptr_t Mailbox;
    int nsteps;
    float elapsed;
    e_dma_desc_t smem2local;
    e_dma_desc_t vneigh2local;
    e_dma_desc_t local2smem;
    volatile e_barrier_t barriers[_Ncores];
    volatile e_barrier_t *tgt_bars[_Ncores];
    int i,j,k;
    volatile unsigned *go, *ready;
    char upper_row[_Nelem];
    char lower_row[_Nelem];
    char *upper_core_grid;
    char *lower_core_grid;

    go = (unsigned *) 0x7000;    //done flag
    ready = (unsigned *) 0x7010; //ready flag to signal end of computation

    me.coreID = e_get_coreid();
    me.row = e_group_config.core_row;
    me.col = e_group_config.core_col;
    me.corenum = me.row * e_group_config.group_cols + me.col;
```

Nel main, ciascun core inizializza i campi della struttura *Me* interrogando un oggetto globale chiamato *e_group_config*, dal quale è possibile ricavare informazioni utili come le coordinate del core all'interno del workgroup.

```
    e_neighbor_id(E_PREV_CORE, E_GROUP_WRAP, &me.row_previous,
        &me.col_previous);
    e_neighbor_id(E_NEXT_CORE, E_GROUP_WRAP, &me.row_next, &me.col_next);

    upper_core_grid =e_get_global_address(me.row_previous, me.col_previous,
```

```
    &me._grid[1][0]);
lower_core_grid =e_get_global_address(me.row_next, me.col_next,
    &me._grid[1][0]);
```

La funzione *e_neighbor_id()* consente di ottenere le coordinate dei core adiacenti. In particolare si specifica nel primo parametro quale vicino si vuole ottenere secondo la topologia passata nel secondo parametro; in questo caso consideriamo i core come se fossero consecutivi, dunque utilizziamo il flag *E_GROUP_WRAP*. Per ottenere le coordinate del core soprastante o sottostante è necessario utilizzare il flag *E_COL_WRAP*, mentre se si vuole uno dei core adiacenti nella stessa riga si utilizza *E_ROW_WRAP*. Le coordinate del core richiesto vengono salvate nelle ultime due variabili passate alla funzione *e_neighbor_id()*. La funzione *e_get_global_address()*, restituisce l'indirizzo globale della variabile passata come ultimo parametro del core le cui coordinate sono specificate nei primi due argomenti. L'idea è dunque quella di ottenere un riferimento alla griglia dei core adiacenti partendo dalle loro coordinate, per poi utilizzare la DMA per prelevare direttamente le righe necessarie dai vicini. Un particolare fondamentale da osservare è che viene richiesto l'indirizzo di *me._grid[1][0]* e non me._grid[0][0] , perché ciascun core deve richiedere le righe dai vicini dopo che essi hanno eseguito lo step orizzontale. Successivamente si inizializzano i puntatori della *Mailbox* ai dati da prelevare presenti nella memoria condivisa come fatto nell'esempio 3.4.3:

```
Mailbox.pBase = (void *) e_emem_config.base;
Mailbox.pGrid = Mailbox.pBase + offsetof(shared_buf_t, grid);
Mailbox.pNsteps = Mailbox.pBase + offsetof(shared_buf_t, nsteps);

nsteps = *Mailbox.pNsteps;

e_barrier_init(barriers, tgt_bars);
```

Per il trasferimento dei dati si utilizzano tre descrittori per la DMA:

- *smem2local* per copiare i dati dalla memoria condivisa a quella locale di ciascun core.

- *vneigh2local* per prelevare l'ultima riga del core precedente e la prima del core successivo.
- *local2smem* per copiare i dati dalla memoria locale a quella condivisa.

```
e_dma_wait(E_DMA_0);

smem2local.config = E_DMA_WORD | E_DMA_MASTER | E_DMA_ENABLE;
smem2local.inner_stride = (0x04 << 16) | 0x04;
smem2local.count = (0x01 << 16) | _Nelem*nrows >> 2 ;
smem2local.outer_stride = (0x04 << 16) | 0x04;

vneigh2local.config = E_DMA_WORD | E_DMA_MASTER | E_DMA_ENABLE;
vneigh2local.inner_stride = (0x04 << 16) | 0x04;
vneigh2local.count = (0x01 << 16) | _Nelem >> 2;
vneigh2local.outer_stride = (0x04 << 16) | 0x04;

local2smem.config = E_DMA_WORD | E_DMA_MASTER | E_DMA_ENABLE;
local2smem.inner_stride = (0x04 << 16) | 0x04;
local2smem.count = (0x01 << 16) | _Nelem*nrows >> 2;
local2smem.outer_stride = 0x04 | 0x04;

data_copy(&smem2local, &me._grid, (Mailbox.pGrid +
    _Nelem*nrows*me.corenum ));
```

In tutti questi descrittori richiediamo di trasferire 4 byte alla volta *(E_DMA_WORD)* utilizzando la dma numero 0. Il primo descrittore è utilizzato per prelevare i dati iniziali dalla memoria condivisa. Poiché ciascun core detiene *nsteps* righe allocate in modo contiguo, il numero di iterazioni da eseguire nel loop più esterno è pari a uno, mentre il numero di iterazioni del ciclo interno è $(\_Nelem \times nrows)/4$ dato che la DMA sposta 4 byte per volta. Il campo *inner_stride* è settato sia alla sorgente che alla destinazione a 4 byte. Analogamente il campo *outer_stride* è impostato a 4 byte, anche se in questo caso non è rilevante dato che il ciclo esterno viene eseguito una sola volta. Nel secondo descrittore vogliamo trasferire una sola riga dal core adiacente ad un vettore locale

dunque è sufficiente un solo ciclo esterno e *_Nelem /4* interni. L'ultimo descrittore è del tutto analogo al primo dato che si tratta dell'operazione inversa. Una volta definiti i tre descrittori, e prima di iniziare la simulazione vera e propria, si invoca la *data_copy()* utilizzata nell'esempio 3.4.3 per far sì che ciascun core copi la corrispondente porzione della griglia totale in *me._grid[0][0]*.

```
    for(k=1; k<nsteps; k++){
        e_barrier(barriers, tgt_bars);

        horizontal_step(&me);
        cur = 1 - cur;

        e_barrier(barriers, tgt_bars);

        data_copy(&vneigh2local, upper_row, upper_core_grid +(nrows-1)*_Nelem);
        data_copy(&vneigh2local, lower_row, lower_core_grid);

        vertical_step(upper_row, lower_row, &me);
        cur = 1 - cur;

        data_copy(&local2smem, (Mailbox.pGrid + _Nelem*nrows*me.corenum),
            me._grid);

        if (me.corenum == 0){
            *ready = 0x00000001;
            while( *go ==0) {};
            *go = 0;
        }
return 0;
}
```

Una volta ottenuti tutti i dati ciascun core è pronto per eseguire la simulazione. La *horizontal_step()* è del tutto simile a quella vista nell'algoritmo sequenziale poiché questa non coinvolge i core adiacenti. Successivamente, si aggiorna la variabile *cur* per

non sovrascrivere lo stato corrente. Notare inoltre l'uso delle barriere, che garantiscono che nessun core prelevi le righe dai suoi vicini prima che essi abbiamo completato lo step intermedio. A questo punto ciascun core preleva autonomamente le righe necessarie alla computazione dai core adiacenti: la prima invocazione della *data_copy()* permette di copiare l'ultima riga del core precedente in un array locale denominato *upper_row()* (*upper_core_grid* e *lower_core_grid* sono stati ottenuti grazie alla *e_get_global_address()*), mentre la seconda chiamata preleva la prima riga dal successore. Terminato lo step verticale ciascun core scrive in memoria condivisa il frame ottenuto. Il core numero 0 è responsabile della sincronizzazione con l'host, di conseguenza segnala all'indirizzo 0x7010 che la computazione è terminata, e aspetta il segnale per procedere alla prossima iterazione leggendo ciclicamente la variabile *go* che punta all'indirizzo 0x7000. La funzione *vertical_step()* è molto simile a quella della versione sequenziale, con la sola differenza che la prima e l'ultima riga devono essere modificate in accordo allo stato dele righe di confine dei core adiacenti.

```
void vertical_step( char *firstRow, char *lastRow, core_t *me){
    int i, j;
    const int next = 1 ? cur;

    //update first row
    for(j=0; j<_Nelem; j++) {
        if (me->_grid[cur][j] == EMPTY && firstRow[j] == TB ){
            me->_grid[next][j] = TB;
        }
        else {
            if ( me->_grid[cur][j] == TB && me->_grid[cur][1*_Nelem+j] ==
                EMPTY) {
                me->_grid[next][j] = EMPTY;
            } else
                me->_grid[next][j] = me->_grid[cur][j];
        }
    }
```

```
for (i=1; i<nrows-1; i++) {
    const int top = i-1;
    const int bottom = i+1;
    for (j=0; j<_Nelem; j++) {
        if (me->_grid[cur][i*_Nelem+j] == EMPTY &&
            me->_grid[cur][top*_Nelem+j] == TB ) {
            me->_grid[next][i*_Nelem+j] = TB;
        }
        else {
            if (me->_grid[cur][i*_Nelem+j] == TB &&
                me->_grid[cur][bottom*_Nelem+j] == EMPTY) {
                me->_grid[next][i*_Nelem+j] = EMPTY;
            } else {
                me->_grid[next][i*_Nelem+j] = me->_grid[cur][i*_Nelem+j];
                }
        }
    }
}

//update last row
i=nrows-1;
for(j=0; j<_Nelem; j++) {
    if (me->_grid[cur][i*_Nelem+j] == EMPTY &&
        me->_grid[cur][(i-1)*_Nelem+j] == TB ) {
        me->_grid[next][i*_Nelem+j] = TB;
    }
    else {
        if ( me->_grid[cur][i*_Nelem+j] == TB && lastRow[j] == EMPTY) {
            me->_grid[next][i*_Nelem+j] = EMPTY;
        } else
            me->_grid[next][i*_Nelem+j] = me->_grid[cur][i*_Nelem+j];
    }
}
```

```
}
```

Per una visione completa del codice sorgente fare riferimento a [14].

6.2 Distribuzione a blocchi

Uno dei grandi vantaggi dell'hardware della Parallella è che facendo un'accurata progettazione delle strutture dati è possibile rendere il programma host indipendente da quello del device. Nel nostro caso accade proprio questo: le operazioni del processore ARM non cambiano in quanto deve solamente limitarsi ad inizializzare la griglia e a sincronizzarsi con il chip Epiphany, mentre quest ultimo è libero di effettuare la computazione nel modo più opportuno. Di conseguenza, non presenteremo nuovamente il codice sorgente per l'host che rimane invariato, ma analizzeremo soltanto il codice per il device in cui cambia il pattern di distribuzione dei dati e della comunicazione tra i core. In questo algoritmo si cerca di sfruttare la topologia del chip Epiphany, in cui i core formano una griglia 4×4; la matrice iniziale viene infatti suddivisa in sedici blocchi, uno per ciascun core (figura 6.2).

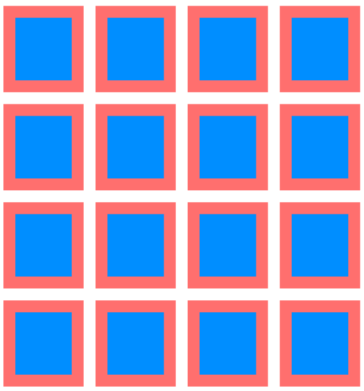


Figura 6.2: Distribuzione a blocchi (in rosso gli elementi che vengono scambiati).

6.2.1 Strutture dati

Le strutture dati utilizzate nell'algoritmo che fa uso di una distribuzione a blocchi sono del tutto analoghe a quelle della strategia precedente, con la differenza che in questo caso ciascun core deve conoscere le coordinate di tutti e quattro i suoi vicini e la dimensione del lato di ciascun blocco ($\_Nside$) deve essere pari a $\_Nelem/\sqrt{\_Ncores}$.

```
#define _Ncores 16
#define _Nelem 256
#define _Nside 64

typedef struct {
    unsigned coreID;
    unsigned corenum;
    unsigned row;
    unsigned col;
    unsigned left_core_row;
    unsigned left_core_column;
    unsigned right_core_row;
    unsigned right_core_column;
    unsigned upper_core_row;
    unsigned upper_core_column;
    unsigned lower_core_row;
    unsigned lower_core_column;
    char _grid[2][_Nside*_Nside];
} core_t;

typedef struct {
    char grid[_Nelem*_Nelem];
    int nsteps;
} shared_buf_t;

typedef struct {
    void *pBase;
```

```
    char *pGrid;
    int *pNsteps;
} shared_buf_ptr_t;

#endif
```

6.2.2 Sorgente Epiphany

```
int main(void){
    volatile shared_buf_ptr_t Mailbox;
    int nsteps;

    e_dma_desc_t smem2local ;
    e_dma_desc_t hneigh2local ;
    e_dma_desc_t vneigh2local ;
    e_dma_desc_t local2smem ;

    volatile e_barrier_t barriers[_Ncores];
    volatile e_barrier_t *tgt_bars[_Ncores];
    int i,j,k;
    volatile unsigned *go, *ready;
    unsigned *elapsed;

    char upper_row[_Nside];
    char lower_row[_Nside];
    char left_column[_Nside];
    char right_column[_Nside];

    char *left_core_grid;           //pointer to the grid of the previous core
    char *right_core_grid;          //pointer to the grid of the following
       core
    char *upper_core_grid;          //pointer to the grid of the core above
```

```
char *lower_core_grid;          //pointer to the grid of the core below

//initialization
go = (unsigned *) 0x7000;       //done flag
ready = (unsigned *) 0x7010;    //flag to signal end of computation

me.coreID = e_get_coreid();
me.row = e_group_config.core_row;
me.col = e_group_config.core_col;
me.corenum = me.row * e_group_config.group_cols + me.col;

e_neighbor_id(E_PREV_CORE, E_ROW_WRAP, &me.left_core_row,
    &me.left_core_column);
e_neighbor_id(E_NEXT_CORE, E_ROW_WRAP, &me.right_core_row,
    &me.right_core_column);
e_neighbor_id(E_PREV_CORE, E_COL_WRAP, &me.upper_core_row,
    &me.upper_core_column);
e_neighbor_id(E_NEXT_CORE, E_COL_WRAP, &me.lower_core_row,
    &me.lower_core_column);

//set pointers to the shared memory
Mailbox.pBase = (void *) e_emem_config.base;
Mailbox.pGrid = Mailbox.pBase + offsetof(shared_buf_t, grid);
Mailbox.pNsteps = Mailbox.pBase + offsetof(shared_buf_t, nsteps);

//get number of steps
nsteps = *Mailbox.pNsteps;
left_core_grid = e_get_global_address(me.left_core_row,
    me.left_core_column, &me._grid[0][0]);
right_core_grid = e_get_global_address(me.right_core_row,
    me.right_core_column, &me._grid[0][0]);
upper_core_grid =e_get_global_address(me.upper_core_row,
    me.upper_core_column, &me._grid[1][0]);
```

```
    lower_core_grid =e_get_global_address(me.lower_core_row,
        me.lower_core_column, &me._grid[1][0]);

    e_barrier_init(barriers, tgt_bars);
```

Nella fase di inizializzazione si definiscono tutte le variabili necessarie come i descrittori per la DMA e gli array locali in cui copiare le righe e le colonne di confine dei core adiacenti. Successivamente si utilizza la *e_neighbor_id()* per ottenere le coordinate di tutti i vicini e vengono inizializzati tutti i puntatori della *Mailbox*. Infine si ottengono i riferimenti alle aree di memoria in cui i vicini mantengono la loro griglia con la *e_get_global_address()*. Notare anche in questo caso che per i core che risiedono nella stessa colonna si ottiene il puntatore a *me._grid[1][0]*, questo perché lo step verticale viene eseguito dopo quello orizzontale.

```
e_dma_wait(E_DMA_0);

smem2local.config = E_DMA_WORD | E_DMA_MASTER | E_DMA_ENABLE;
smem2local.inner_stride = (0x04 << 16) | 0x04;
smem2local.count = (_Nside << 16) | _Nside >> 2 ;
smem2local.outer_stride = (0x04 << 16) | (((_Nelem - _Nside) * sizeof(char))
    + 0x04);

hneigh2local.config = E_DMA_BYTE | E_DMA_MASTER | E_DMA_ENABLE;
hneigh2local.inner_stride = (0x01 << 16) | 0x01;
hneigh2local.count = (_Nside << 16) | 0x01;
hneigh2local.outer_stride = (0x01 << 16) | _Nside;

vneigh2local.config = E_DMA_WORD | E_DMA_MASTER | E_DMA_ENABLE;
vneigh2local.inner_stride = (0x04 << 16) | 0x04;
vneigh2local.count = (0x01 << 16) | _Nside >> 2;
vneigh2local.outer_stride = (0x04 << 16) | 0x04;

local2smem.config = E_DMA_WORD | E_DMA_MASTER | E_DMA_ENABLE;
local2smem.inner_stride = (0x04 << 16) | 0x04;
```

```
local2smem.count = (_Nside << 16) | _Nside >> 2;
local2smem.outer_stride = ( (((_Nelem - _Nside) * sizeof(char)) + 0x04) <<
    16) | 0x04

data_copy(&smem2local, &me._grid, (Mailbox.pGrid + _Nelem*_Nside*me.row +
    _Nside*me.col));
```

Settare i descrittori con questo tipo di distribuzione è sicuramente più complesso dato che i dati da trasferire non sono contigui. Il primo descrittore è utilizzato da un core per copiare in locale la porzione della griglia di cui è responsabile; ricordiamo che la griglia ha dimensione $\_Nelem \times \_Nelem$ mentre i blocchi $\_Side \times \_Side$. Fissiamo quindi il numero di cicli esterni a $\_Nside$ e quelli interni a $\_Nside/4$ a causa del flag *E_DMA_WORD*. Il salto da effettuare al termine di un'iterazione interna è di 4 byte sia alla sorgente che alla destinazione. Nel momento in cui il ciclo interno raggiunge il valore zero, il puntatore alla destinazione deve essere incrementato solo di 4 byte, mentre quello alla sorgente (la griglia iniziale) deve spostarsi alla riga successiva; questo significa che se una riga della griglia ha dimensione $\_Nelem$ per saltare alla riga successiva dobbiamo avanzare di $\_Nelem - \_Nside$ byte. I 4 bytes aggiuntivi sono necessari perchè come spiegato nell'esempio 3.4.3, ogniqualvolta il contatore del ciclo interno aggiunge lo zero l'*inner_stride* non viene applicato.

Il secondo descrittore è utilizzato da un core per ottenere una colonna dai core adiacenti nella stessa riga. Dato che i blocchi sono disposti in memoria per righe e ciascun elemento della colonna ha la dimensione di un CHAR, in questo caso il flag corretto da specificare è *E_DMA_BYTE*. Una colonna ha dimensione $\_Side$, dunque sono sufficienti $\_Side$ iterazioni nel ciclo esterno e una sola iterazione nel ciclo interno. Sempre a causa del fatto che le colonne non sono memorizzate in modo contiguo, dopo ogni CHAR copiato è necessario saltare di $\_Side$ byte alla sorgente.

Il terzo descrittore serve per ottenere la prima o l'ultima riga dei blocchi mantenuti dai core adiacenti nella stessa colonna. In questo caso si sfrutta l'allocazione contigua, di conseguenza si trasferiscono 4 byte alla volta. L'ultimo descrittore è analogo al primo poiché si vuole copiare il blocco locale nella memoria condivisa. Infine, prima di iniziare la simulazione si invoca la *data_copy()* per far sì che ciascun core copi in locale il cor-

rispondente blocco della griglia iniziale; la destinazione è *me._grid*, mentre per ottenere l'indirizzo sorgente si utilizza un offset calcolato a seconda delle coordinate del core da aggiungere all'indirizzo base. La griglia è memorizzata nella memoria condivisa per righe (di dimensione $\_Nelem$), dunque l'offset è $\_Nelem \times \_Nside \times me.row + \_Nside \times me.col$.

```
    for(k=1; k<nsteps; k++){
        e_barrier(barriers, tgt_bars);
        data_copy(&hneigh2local, left_column, left_core_grid+_Nside-1);
        data_copy(&hneigh2local, right_column, right_core_grid);

        horizontal_step(left_column, right_column, &me);
        cur = 1 - cur;
        e_barrier(barriers, tgt_bars);
        data_copy(&vneigh2local, upper_row, upper_core_grid
            +(_Nside-1)*_Nside);
        data_copy(&vneigh2local, lower_row, lower_core_grid);
        vertical_step(upper_row, lower_row, &me);
        cur = 1 - cur;

        data_copy(&local2smem, (Mailbox.pGrid + _Nelem*_Nside*me.row +
            _Nside*me.col), me._grid);
        if (me.corenum == 0){
            *ready = 0x00000001;
            while( *go ==0) {};
            *go = 0;
        }
    }
return 0;
}
```

A questo punto la simulazione può iniziare. Prima di eseguire uno step orizzontale ciascun core preleva l'ultima riga dal blocco del core precedente e la prima di quello successivo. Allo stesso modo prima di eseguire uno step verticale si prelevano le colonne

dai vicini e le si salvano in due array locali per effettuare la computazione. Infine, dopo che un passo è stato completato, il core numero 0 si sincronizza con l'host.

```
void horizontal_step(char *firstColumn,char *lastColumn, core_t *me)
{
    int i, j;
    const int next = 1 - cur;

    //update first column
    for (i=0; i<_Nside; i++)
    {
        if (me->_grid[cur][i*_Nside] == EMPTY && firstColumn[i] == LR ) {
            me->_grid[next][i*_Nside] = LR;
        }
        else {
            if ( me->_grid[cur][i*_Nside] == LR && me->_grid[cur][i*_Nside+1]
                == EMPTY ) {
                me->_grid[next][i*_Nside] = EMPTY;
            } else
            me->_grid[next][i*_Nside] = me->_grid[cur][i*_Nside];
        }
    }
    for (i=0; i<_Nside; i++) {
        for (j=1; j<_Nside-1; j++) {
            const int left = j-1;
            const int right = j+1;
            if ( me->_grid[cur][i*_Nside+j] == EMPTY &&
                me->_grid[cur][i*_Nside+left] == LR ) {
                me->_grid[next][i*_Nside+j] = LR;
            }
            else
            {
                if (me->_grid[cur][i*_Nside+j] == LR &&
                    me->_grid[cur][i*_Nside+right] == EMPTY) {
```

```
                me->_grid[next][i*_Nside+j] = EMPTY;
            } else
            me->_grid[next][i*_Nside+j] = me->_grid[cur][i*_Nside+j];
        }
    }
  }
  //update last column
  j=_Nside-1;
  for (i=0; i<_Nside; i++) {
    if (me->_grid[cur][i*_Nside+j ] == EMPTY &&
        me->_grid[cur][i*_Nside+j-1] == LR ) {
        me->_grid[next][i*_Nside+j] = LR;
    }
    else {
        if ( me->_grid[cur][i*_Nside+j] == LR && lastColumn[i] == EMPTY ) {
            me->_grid[next][i*_Nside+j] = EMPTY;
        } else
            me->_grid[next][i*_Nside+j] = me->_grid[cur][i*_Nside+j];
    }
  }
}
```

Per una visione completa del codice sorgente fare riferimento a [14].

Capitolo 7

Analisi delle prestazioni

In questo capitolo affronteremo un'analisi dei due algoritmi paralleli per effettuarne un confronto. Come vedremo una sola analisi teorica non è sufficiente per stabilire quale dei due sia in grado di garantire le prestazioni migliori; come spesso accade nel calcolo parallelo infatti, le costanti sono di fondamentale importanza.

7.1 Analisi teorica

Per effettuare l'analisi utilizzeremo la seguente notazione:

- n: dimensione del lato della griglia su cui si effettua la simulazione.
- p: numero di core utilizzati per la computazione.
- c_1: tempo necessario per l'avvio della comunicazione.
- c_2: tempo necessario per trasferire un byte da un core all'altro utilizzando la DMA.
- t_{comp}: tempo richiesto per la computazione di un passo della simulazione.
- t_{comm}: tempo richiesto per la comunicazione durante un passo della simulazione.

Nell'algoritmo in cui si utilizza una distribuzione a righe ciascun core detiene $\frac{n^2}{p}$ elementi, e ciascuno di essi viene aggiornato in tempo costante. Di conseguenza si ha $t_{comp} \sim (\frac{n^2}{p})$ In ciascuna iterazione della simulazione, ogni core deve prelevare una intera riga da

entrambi i vicini che si trovano nella stessa colonna, per cui $t_{comm} \sim 2c_1 + 2nc_2$. Per quanto riguarda l'algoritmo che utilizza una strategia a blocchi invece, ciascun core lavora in locale con $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ elementi, quindi si ottiene ancora una volta $t_{comp} \sim (\frac{n^2}{p})$. Le cose cambiano nella comunicazione, in quanto ciascun core deve prelevare per quattro volte un array di $\frac{n}{\sqrt{p}}$, per cui vale $t_{com} \sim 4c_1 + 4c_2\frac{n}{\sqrt{p}}$. Ricordando che l'obbiettivo è massimizzare il rapporto $\frac{t_{comp}}{t_{comm}}$, non siamo in grado di stabilire quale dei due programmi sia il migliore, poiché il costo relativo alla computazione è lo stesso per entrambi gli algoritmi, mentre per conoscere i costi di comunicazioni è necessario esplicitare il valore di c_1 e c_2; dobbiamo quindi affidarci ad un'analisi sperimentale.

7.2 Tempo d'esecuzione

Per misurare il tempo d'esecuzione si è escluso il tempo necessario al processore ARM per scrivere su un file esterno lo stato della griglia, in quanto siamo interessati a confrontare due metodi differenti per la stessa simulazione. Per garantirne la correttezza tuttavia, il primo e l'ultimo frame vengono comunque stampati, in modo da verificare se partendo dallo stesso input i due programmi producono lo stesso output. A causa dei limiti in termini di memoria e del numero di core, la strong scaling analysis è stata effettuata fissando la dimensione della griglia a 128×128 e variando il numero di core da quattro a sedici; mentre il numero di passi della simulazione è stato fissato a 1024. La tabella 7.1 riassume i tempi ottenuti a seconda delle configurazioni utilizzate:

Core	Row distribution time	Block distribution time
4	0,437 s	0.337 s
16	0.112 s	0.088 s

Tabella 7.1: Tempo d'esecuzione con distribuzione a righe e a blocchi.

Il tempo d'esecuzione risulta molto simile con un leggero vantaggio a favore della strategia che utilizza una distribuzione a blocchi, ma in entrambi in casi si ha un consistete guadagno dato che l'esecuzione dell'algoritmo sequenziale sulla Parallella Board richiede 0,935 secondi. Dividendo il tempo sequenziale con i tempi mostrati nella tabella 7.1 si ottiene lo Speedup per ciascuna configurazione (figura 7.1) . Probabilmente i

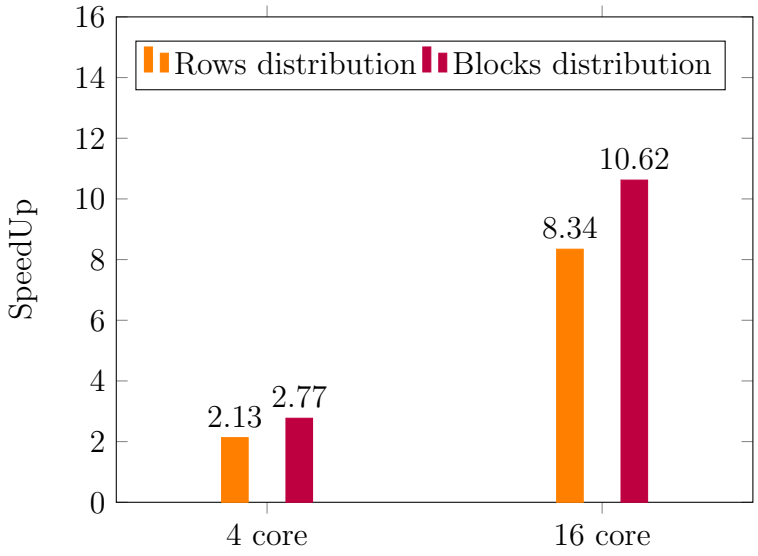


Figura 7.1: SpeedUp ottenuto con distribuzione a righe e a blocchi.

risultati migliori si ottengono con la distribuzione a blocchi perché la costante relativa al tempo d'avvio della comunicazione (e quindi della DMA) ha un valore molto piccolo, di conseguenza i termini dominanti per quanto riguarda la comunicazione per il primo e il secondo algoritmo sono rispettivamente $2c_2n$ e $4c_2\frac{n}{\sqrt{p}}$. Per $n = 128$ si ottiene $256c_2$ e $128c_2$ rispettivamente, dunque il secondo algoritmo presenta un miglior rapporto $\frac{t_{comp}}{t_{comm}}$ e quindi migliori prestazioni.

Conclusioni e sviluppi futuri

In questa tesi si è analizzata l'architettura hardware e software della Parallella Board, un supercomputer delle dimensioni di una carta di credito. Sono stati introdotti una serie di esempi di difficoltà crescente per familiarizzare con la scheda, per finire con due diverse implementazioni del modello di Biham-Middleton-Levine che utilizzano le principali funzioni fornite dall'eSDK Epiphany. Uno dei grandi vantaggi della Parallella Board è il basso costo; per questo motivo potrebbe essere interessante creare un cluster di Parallella ed estendere i programmi per analizzarne le prestazioni con input di dimensioni maggiori. Questo lavoro potrà anche essere un aiuto per tutti gli studenti o chiunque fosse interessato al calcolo parallelo per imparare a programmare questa scheda, svincolando i programmatori dal dovere leggere una documentazione che presenta purtroppo alcune lacune.

Bibliografia

[1] G. E. Moore, "Cramming more components onto integrated circuits", Electronics Magazine, 1965.

[2] M. Flynn, in "Very high-speed computing systems", IEEE, 1966, 1901-1909

[3] A. Varghese, B. Edwards, G. Mitra, A. P. Rendell, "Programming the Adapteva Epiphany 64-core Network-on-chip Coprocessor", in Proceedings of the 28th International Parallel & Distributed Processing Symposium Workshops, pp.984-992, Phoenix, USA, May 19-23, 2014.

[4] Epiphany SDK reference Manual: http://adapteva.com/docs/epiphany_sdk_ref.pdf.

[5] Adapteva: Parallella Reference Manual: http://www.parallella.org/docs/parallella_manual.pdf.

[6] Adapteva: Epiphany Architecture Reference: http://adapteva.com/docs/epiphany_arch_ref.pdf.

[7] A. Papadogiannakis, S. N. Agathos, and V. V. Dimakopoulos, "OpenMP 4.0 Device Support in the OMPi Compiler", in OpenMP: Heterogenous Execution and Data Movements. Springer, 2015, pp. 202-216.

[8] J. A. Ross, D. A. Richie, S. J. Park, and D. R. Shires, "Parallel programming model for the Epiphany many-core coprocessor using threaded MPI", in Proceedings of the 3rd International Workshop on Many-core Embedded Systems, 2015, pp. 41-47.

[9] M. Ilg, J. Rogers, M. Costello, "Projectile Monte Carlo Trajectory Analysis Using a Graphics Processing Unit", 2011 AIAA Atmospheric Flight Mechanics Conference.

[10] http://suzannejmatthews.github.io/2015/06/10/dot-product-revisited.

[11] O. Biham, A. A. Middleton, D. Levine, "Self organization and a dynamical transition in traffic flow models", Phys. Rev. A 46, R6124, 1992.

[12] O. Angel, A. E. Holroyd, and J. B. Martin, "The Jammed Phase of the Biham-Middleton-Levine Traffic Model", Electronic Communications in Probability, 10:167-178, 2005.

[13] T. D. Austin, I. Benjamini, "For what number of cars must self organization occur in the Biham?Middleton?Levine traffic model from any possible starting configuration?", 2006, https://arxiv.org/abs/math/0607759.

[14] https://github.com/adricarda/Traffic-model.