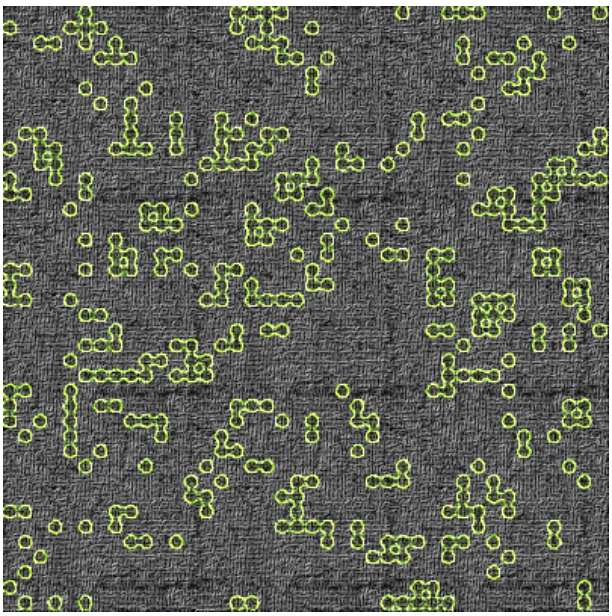
Studio e implementazione parallela del Gioco della Vita

Zanelli Andrea



9 marzo 2010

Indice

1 Introduzione

Lo scopo di questo documento è studiare una possibile implementazione parallela del **gioco della vita**, realizzarla con il supporto della libreria di skeleton Muesli [3] e verificarne l'effettivo funzionamento con alcuni test pratici.

Si inizia quindi con una breve introduzione del gioco della vita, come questo può essere rappresentato e realizzato a livello di programmazione e quale algoritmo e strutture dati si possono utilizzare. Dopodiché si passa allo studio di come il gioco della vita può essere implementato in modo parallelo, quindi come possono essere adattati l'algoritmo e le strutture dati a questo scopo. Finita la parte di studio del problema si passa alla descrizione dell'implementazione del programma, fatta con linguaggio `C++`, utilizzando la libreria Muesli per realizzarne la struttura parallela. Infine verranno mostrati e discussi i risultati dell'esecuzione del programma al variare di alcuni parametri in input.

2 Il gioco della vita

Il gioco della vita è un automa cellulare[1] ideato dal matematico inglese John Horton Conway nel 1970. Il "gioco" non ha giocatori, nel senso che la sua evoluzione dipende da uno stato iniziale e non ha bisogno di interazioni umane, lo stato successivo è interamente calcolato a partire da quello precedente.

2.1 Regole

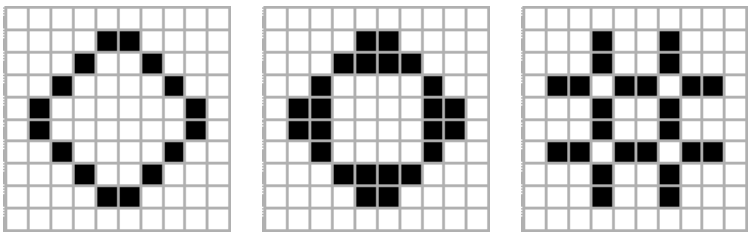
Il gioco della vita può essere ridotto ad alcune semplici regole:

- Si svolge su una griglia di caselle quadrate (**celle**) questa griglia è detta **mondo**.
- Il mondo gira attorno: la colonna sul lato sinistro è considerata successiva a quella destra (e viceversa), la riga superiore è considerata successiva a quella inferiore (e viceversa).
- Ogni cella ha 8 **vicini**, che sono le celle ad essa adiacenti in verticale, orizzontale e diagonale.
- Ogni cella può trovarsi in uno tra due stati: **viva** o **morta**.

[1] Un automa cellulare non è altro che una griglia regolare di celle in cui ogni cella può avere un numero finito di stati (per esempio "acceso" e "spento"). Uno stato iniziale (t=0) è stabilito assegnando uno stato ad ogni cella. Una **nuova generazione** è creata (avanzando t di 1) in accordo ad alcune regole che determinano un nuovo stato per ogni cella a partire dallo stato precedente.

- Gli stati di tutte le celle ad un dato istante sono utilizzati per calcolare lo stato successivo in accordo con le seguenti regole:
  - Ogni cella viva con meno di due vicini vivi muore, per isolamento.
  - Ogni cella viva con più di tre vicini vivi muore, per sovraffollamento.
  - Ogni cella viva con due o tre vicini vivi sopravvive (resta viva).
  - Ogni cella morta con esattamente tre vicini vivi nasce (diventa una cella viva).
- Tutte le celle del mondo vengono così aggiornate simultaneamente nel passaggio da un istante a quello successivo: passa così una **generazione**.

Di seguito è riportato un esempio del gioco della vita in una griglia 10x10 (il mondo) dove si può vedere come evolgono gli stati delle celle in tre generazioni successive. Le celle vive sono rappresentate con quadrati neri, quelle morte con quadrati bianchi.



t = 0 *t = 1* *t = 2*

2.2 Algoritmi

Per rappresentare e simulare il gioco della vita sono stati studiati diversi algoritmi e strutture dati, soprattutto con lo scopo di ottimizzare i calcoli delle generazioni successive (quindi per velocizzare computazioni di molte generazioni) o per rappresentarlo nel modo più completo possibile, ad esempio permettendo un mondo "infinitamente" grande.

La prima e più semplice rappresentazione è attraverso una matrice (o un array bidimensionale) in cui ogni oggetto della matrice rappresenta una cella del gioco della vita, e può prendere valore 1 o 0 per rappresentare le celle rispettivamente vive o morte. In questo caso l'algoritmo per il calcolo della generazione successiva costruisce una nuova matrice di dimensioni identiche alla prima, per ogni cella conta il numero di vicini vivi e in base a tale numero decide il nuovo stato della cella (viva o morta, 1 o 0) nella nuova matrice.

Un'ottimizzazione a questo algoritmo potrebbe tenere conto che se una cella non ha cambiato stato nell'ultima generazione e nessuno dei suoi vicini ha cambiato stato, allora tale cella sicuramente non cambierà stato nella

nuova generazione; un algoritmo più furbo può quindi mantenere traccia delle zone che non hanno subito cambiamenti nell'ultima generazione ed evitare molti conti inutili.

Per il calcolo di grandi quantità di generazioni è stato invece studiato un sofisticato algoritmo, l'HashLife, che sfrutta il comportamento ripetitivo di molte configurazioni (patterns) del gioco della vita per evitare di dover eseguire sempre gli stessi calcoli, in questo modo si riescono a calcolare miliardi di miliardi di generazioni in pochi secondi ("saltando" però dei passi intermedi).

Se si vuole invece rappresentare un mondo infinito, senza confini, allora si deve cambiare struttura dati, non più una matrice ma un array di coordinate delle celle vive, in questo modo si riuscirebbe a rappresentare un mondo praticamente infinito (limitato solamente dal massimo numero rappresentabile sul calcolatore per le coordinate estreme), ma chiaramente il conto dei vicini vivi si trasformerebbe in una ricerca sul vettore, aumentando la complessità del calcolo di nuove generazioni.

In questo documento si ci vuole però concentrare sul calcolo parallelo e distribuito, percui per rappresentare il gioco della vita viene preso il primo modello, quello più semplice ed intuituvo, una matrice di 1 e 0. Con questa struttura l'algoritmo per il calcolo della generazione successiva è molto semplice e riportato di seguito in pseudo-codice.

Sia `M1` una matrice di `R` righe e `C` colonne che rappresenta uno stato del gioco della vita. Sia `M2` una matrice delle dimensioni di `M1`. Allora l'algoritmo `generazione_successiva` calcola la generazione successiva rispetto alla matrice `M1` e mette il risultato nella matrice `M2`:

```
generazione_successiva(M1, R, C, M2) {
  for I from 0 to R do
    for J from 0 to C do
      V := numero_vicini_vivi(M1,I,J);
      if M1[I][J] == 1 then
        if V == 2 or V == 3 then
          M2[I][J] := 1;           // la cella sopravvive
        else
          M2[I][J] := 0;           // la cella muore
        endif
      else
        if V == 3  then
          M2[I][J] := 1;           // la cella "nasce"
        else
          M2[I][J] := 0;           // la cella rimane morta
        endif
      endif
    endfor
```

```
  endfor
}
```

Diremo allora che un'**iterazione** sulla matrice che rappresenta il gioco della vita è l'applicazione dell'algoritmo precedente su di essa, sostituendola infine con la nuova matrice (`M2`). Compiere n iterazioni sulla matrice vorrà quindi dire calcolare n generazioni.

Su una matrice di r righe e c colonne l'algoritmo ha costo nell'ordine di $8 * r * c$, infatti è previsto che siano fatti otto controlli (per il calcolo del numero di vicini vivi) per ogni elemento della matrice; nel caso di matrici quadrate il costo dell'algoritmo è quindi esponenziale. Una simulazione del gioco della vita consisterà nel compiere n iterazioni dell'algoritmo, il che è ovviamente molto costoso per matrici di grandi dimensioni. Questi tempi possono essere ammortizzati distribuendo il calcolo in parallelo.

3 Parallelizzazione

Come già detto rappresentiamo il gioco della vita con una matrice. Siccome l'algoritmo prevede di computare ogni cella della matrice, il modo più opportuno per parallelizzare e ditribuire il calcolo è suddividere la matrice in **fette** ed assegnare una fetta ad ogni **PE** (Process Element)[2] che gli potrà applicare l'algoritmo in modo parallelo. Nella figura 1 sono rappresentate

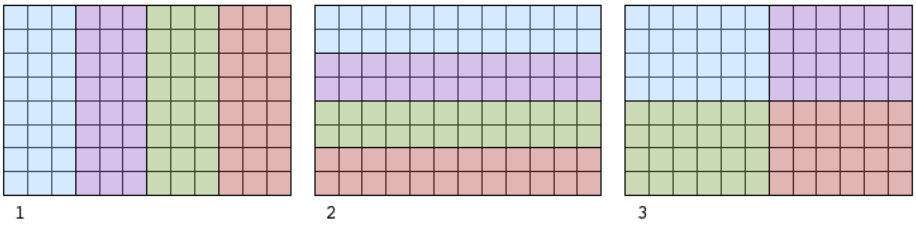


Figura 1: *Suddivisione della matrice in fette: per colonne (1), per righe (2) o per "pezzi" (3).*

tre tipi di suddivisioni: per colonne, per righe e per "pezzi". La suddivisione influisce sulle prestazioni poiché, come si vedrà meglio più avanti, ogni PE che ha una fetta dovrà comunicare (in una fase di sincronizzazione) con i PE che hanno le fette di matrice confinanti alla sua. Allora le prime due suddivisioni praticamente si equivalgono (nel caso di matrici quadrate sarebbero identiche, altrimenti dipende se ci sono più righe o colonne), mentre la terza, per "pezzi", non porta alcun vantaggio rispetto alle prime due, ma

[2] Process Element: può essere un processo concorrente su una singola macchina oppure una macchina dedicata al calcolo.

anzi può portare alcuni problemi dato che ogni fetta confina con altre otto fette (tenendo sempre conto della regola che il mondo gira attorno), mentre con le prime due si ha che ogni fetta confina solamente con altre due fette. Perciò la matrice verrà suddivisa per colonne, assegnando quindi ad ogni PE un certo numero di colonne della matrice, su cui poi eseguirà l'algoritmo del gioco della vita. Siccome suddividiamo la matrice per colonne, chiameremo **bordo sinistro** della fetta la prima colonna e **bordo destro** l'ultima colonna della fetta, ovvero l'insieme delle celle che hanno come vicini celle in altre fette. Chiameremo inoltre **vicino destro** di un PE il PE che ha la fetta precedente alla sua, mentre il **vicino sinistro** sarà il PE che ha la fetta successiva. Da notare che non necessariamente due PE con indice consecutivo, ad esempio il PE 0 e il PE 1, saranno vicini, ma dipenderà da quale fetta gli verrà assegnata.

Ogni PE, per applicare l'algoritmo del gioco della vita sulla propria fetta, calcola il valore di una cella sulla base delle otto celle vicine; se ogni PE vede però solamente la sua fetta non riuscirà mai a calcolare neanche una singola iterazione siccome non conosce i valori delle celle vicine alle celle sui bordi, che fanno parte di una fetta assegnata ad un'altro PE. Si deve quindi dare in qualche modo, ad ogni PE, una visione dei bordi dei PE vicini.

Una prima soluzione può essere dare una copia dell'intera matrice ad ogni PE, con un approccio fortemente centralizzato, in cui ogni PE può comunicare solamente con un PE radice; il processo potrebbe seguire quindi i seguenti passi:

1. Il PE radice comunica a tutti i PE la matrice completa e la fetta di cui ognuno si deve occupare.

2. Ogni PE compie un'iterazione sulle celle della fetta che gli è stata assegnata, guardando i vicini delle celle nella propria copia locale della matrice.

3. Terminata un'iterazione, ogni PE rimanda indietro alla radice solamente la fetta di matrice che ha modificato.

4. La radice riceve tutte le fette e ricostruisce la matrice, che a questo punto rappresenta la generazione successiva, e ricomincia il processo dall'inizio fino a compiere n iterazoni.

Seppur corretto e funzionante, questo approccio ha l'evidente problema che viene perso molto tempo per trasferire dati inutili. Infatti ogni PE riceve l'intera matrice all'inizio di ogni iterazione, quando gli basterebbe ricevere solamente la sua fetta più le due colonne dei bordi dei PE vicini, mentre il PE radice riceve l'intera matrice alla fine di ogni iterazione. Se indichiamo con con p il numero di PE coinvolti, in totale vengono trasferite p matrici ad ogni iterazione; considerando che la matirce sarà di notevoli dimensioni

(migliaia di righe per migliaia di colonne) con questo apporccio si rischia di perdere più tempo a trasferire dati che ad eseguire l'algoritmo.

Come detto, ad ogni PE che deve eseguire l'algoritmo gli basta ricevere la fetta su cui deve lavorare più le due colonne dei PE vicini. Ma ovviamente ad ogni iterazione le colonne dei PE vicini devono essere aggiornate perché anche su quelle colonne è passata una generazione. Rimanendo nell'approccio precedente ogni PE potrebbe ricevere la fetta più le due colonne, eseguire un'iterazione sulla sua fetta, rimandarla al PE radice il quale ricostruisce la matrice e rinvia le fette e le colonne aggiornate ad ogni PE, ricominciando l'intero processo. È però subito evidente che il processo può essere ulteriormente migliorato facendo in modo che ogni PE scambi direttamente i suoi bordi con i PE vicini, senza passare da un PE centralizzato, che porterebbe ad un rallentamento del calcolo. Allora si può migliorare l'approccio precedente nel modo seguente:

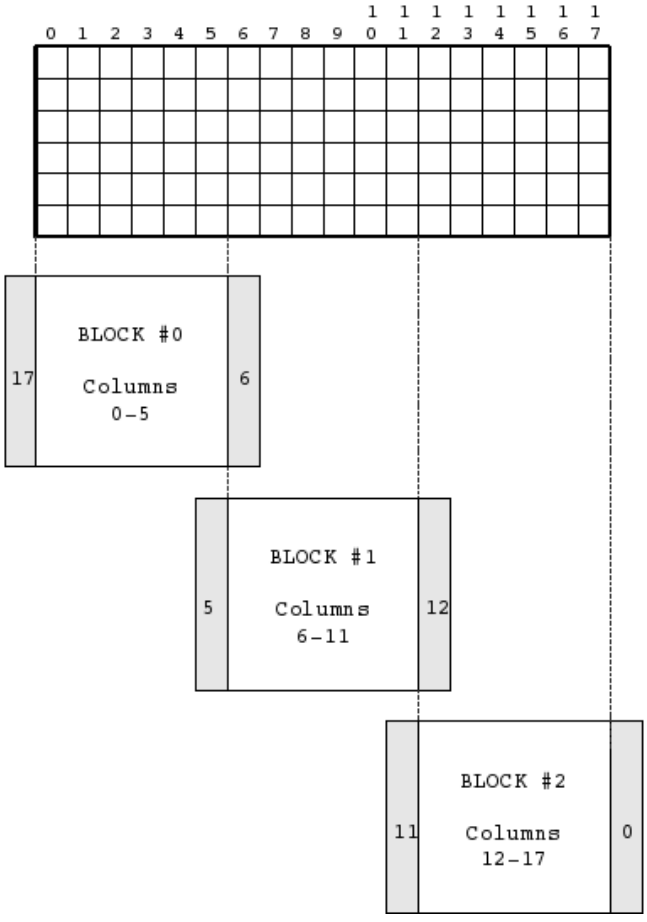


Figura 2: *Suddivisione di una matrice 6x18 in 3 blocchi.*

1. Il PE radice costruisce la matrice e e la suddivide in **blocchi**, composti dalla fetta di matrice su cui un PE lavorerà, più la colonna immediatamente a sinistra prima della fetta ed immediatamente a destra dopo la fetta, che serviranno al PE per calcolare i valori delle celle nel bordo della sua fetta; dopodiché invia un blocco ad ogni PE. Nella figura 2 si può vedere un esempio di come la matrice viene separata in blocchi. Ogni blocco è composto da una fetta di matrice più due vettori, che chiameremo **vettore sinistro** e **vettore destro**, che rappresentano rispettivamente il bordo destro del blocco a sinistra e il bordo sinistro del blocco a destra.

2. I PE eseguono un'iterazione sulla loro fetta di matrice, utilizzando i vettori del blocco per calcolare i nuovi valori delle celle sul bordo.

3. Ogni PE si sincronizza con gli altri aspettando che tutti abbiano concluso l'iterazione.

4. Ogni PE invia il suo bordo destro al vicino destro ed il suo bordo sinistro al suo vicino sinistro, aspettandosi di ricevere i rispettivi bordi dai rispettivi vicini che andranno ad aggiornare i vettori del blocco. A questo punto ogni PE ha i vettori aggiornati e può procedere ad un'altra iterazione.

5. Quando un PE ha eseguito n iterazioni sul suo blocco, lo rinvia al PE radice che ricostruisce la matrice che sarà il risultato del gioco della vita.

Con questo approccio si riesce a ridurre al minimo la quantità di dati scambiati tra i processi ed in questo modo si potranno ottenere prestazioni ottimali dalla parallelizzazione. Questo metodo è quello che verrà utilizzato nell'implementazione parallela del gioco della vita, e più avanti vengono descritti i dettagli su come dividere esattamente la matrice e come si possono sincronizzare i PE alla fine di ogni iterazione.

Un altro modo possibile per risolvere il problema delle celle sui bordi è un approccio "su richiesta". Invece che scambiarsi ad ogni iterazione le intere colonne dei bordi, ogni PE potrebbe fare la richiesta del valore della cella che non fa parte della sua fetta direttamente al PE vicino nel momento in cui gli serve (quindi durante il calcolo dell'iterazione). In questo modo si riuscirebbero ad ottenere benefici se solo alcune delle celle nei bordi fossero elaborate, poiché in tal caso non sarebbe necessario scambiarsi l'intera colonna ma sarebbero sufficienti solo pochi valori relativi ad alcune celle. Siccome però l'alogritmo prevede di elaborare tutte le celle, due processi si dovrebbero comunque comunicare l'intera colonna, in questo caso non tutta in una volta ma un valore alla volta, quando richiesto, portando però a rallentamenti dovuti alle numerose operazioni di sincronizzazione che sarebbero necessarie per scambiarsi i valori in questo modo. Per questo si preferisce

utilizzare l'approccio precedente, in cui è necessaria una sola operazione di sincronizzazione alla fine di ogni iterazione.

3.1 Bilanciamento del carico di lavoro

Stabilito che per il calcolo parallelo la matrice viene suddivisa per colonne, e ad ogni PE verranno quindi assegnate un certo numero di colonne su cui dovrà lavorare, resta da specificare come esattamente dividere la matrice dato il numero di PE, ovvero quante colonne mettere in ogni blocco.

Diciamo allora che abbiamo una matrice di c colonne (il numero di righe è ininfluente), ed n PE a cui poter assegnare dei blocchi di matrice su cui lavorare. Assumiamo che n sia minore-uguale a c (il caso in cui n è maggiore non ha senso dal lato pratico, percui lo escludiamo). Naturalmente se n divide esattamente c (senza resto), allora daremo c/n colonne ad ogni PE, in questo modo ogni PE avrà lo stesso carico di lavoro. Se invece n non divide c, si potrebbe pensare di assegnare il resto della divisione all'ultimo PE (in aggiunta alle altre colonne già assegnate); quindi, per esempio, se abbiamo una matrice 32x32 e 6 PE, assegnamo 5 colonne ai PE da 0 a 4, e 7 colonne (5 + il resto 2) al PE numero 5. In questo modo un PE si troverebbe con più lavoro da eseguire rispetto agli altri, e questa non è solo un'ingiustizia, ma porterebbe anche ad un rallentamento degli altri PE. Infatti alla fine di ogni iterazione è previsto che ogni PE si sincronizzi con gli altri per scambiarsi il proprio bordo, e se un solo PE tarda rallenta anche l'esecuzione degli altri. Percui, in generale, un buon assegnamento delle colonne ai PE influisce sulle prestazioni dell'intero processo, come rappresentato in figura 3.

Allora l'ideale è avere una situazione in cui ogni PE ha "praticamente" lo stesso numero di colonne degli altri. Il problema si pone quando vi è del resto nella divisione tra il numero di colonne c e il numero di PE n. Allora, intuitivamente, si può assegnare uno stesso numero di colonne ad ogni PE ($\lfloor c/n \rfloor$) e distribuire un eventuale resto r assegnando una sola colonna in più ai primi r PE, poichè sicuramente si ha $r < n$. In altre parole l'idea è di assegnare ciclicamente una colonna per volta ad ogni PE, se in questo modo l'ultima colonna viene assegnata all'ultimo PE significa che non c'era resto, altrimenti i primi r PE avranno una colonna in più rispetto agli altri. Con questo metodo si ottiene una situazione in cui, per qualunque numero di colonne, ogni PE avrà al più una colonna in più rispetto agli altri PE; considerando che, in casi reali, ogni PE avrà centinaia di colonne da processare, una in più non appesantisce in modo rilevante il carico di lavoro.

Ovviamente si deve assegnare ad ogni PE un blocco della matrice con un certo numero di colonne consecutive, percui si vuol sapere, al momento della suddivisione in blocchi, il numero di colonne che ogni blocco deve avere. Dato `C` il numero di colonne della matrice, `N` il numero di blocchi in cui si vuole suddividere la matrice (che sarà il numero di PE che devono ricevere un blocco) ed `I` il numero del blocco che si vuole costruire, il seguente algoritmo

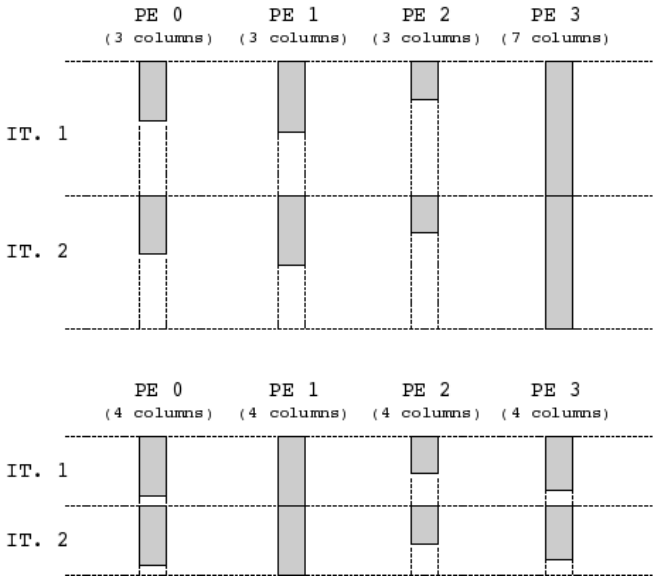


Figura 3: *Un bilanciamento sbagliato delle colonne può avere forti ripercussioni sul tempo totale di esecuzione.*

restituisce il numero di colonne da assegnare al blocco in accordo con il metodo appena descritto:

```
number_of_columns(C, N, I) {
  D := parte_intera_inferiore(C/N);
  R := C mod N;
  if I < R then
    return D + 1;
  else
    return D;
}
```

Come già detto, utilizzando questo algoritmo, si può separare la matrice in blocchi che differiscono al più di una colonna, ottenendo così un ottimo bilanciamento del carico di lavoro dei PE che dovranno applicare il gioco della vita sui blocchi.

3.2 Sincronizzazione

Alla fine di ogni iterazione ogni PE deve scambiare i propri bordi con i PE vicini ed aggiornare i vettori del suo blocco ricevendo i bordi dei vicini. Un esempio degli scambi che devono avvenire si può vedere in figura 4.

Guardando la figura si può capire quanto sia delicato questo passaggio e che può essere molto facile arrivare ad un deadlock in questa fase di sincronizzazione. Assumiamo allora di avere una funzione *send* non bloccante per

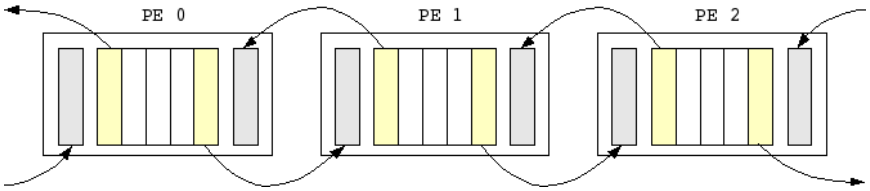


Figura 4: *Lo scambio dei bordi tra PE alla fine di ogni iterazione. Ogni PE ha il suo blocco su cui ha eseguito un'iterazione, composto dalla fetta di matrice (con evidenziate le colonne di bordo) e dai due vettori sinistro e destro che devono essere aggiornati.*

spedire una colonna del blocco e una funzione *receive* bloccante per ricevere una colonna. Un primo modo per sincronizzare i dati tra i PE può essere un modello a catena come quello raffigurato in figura 5.

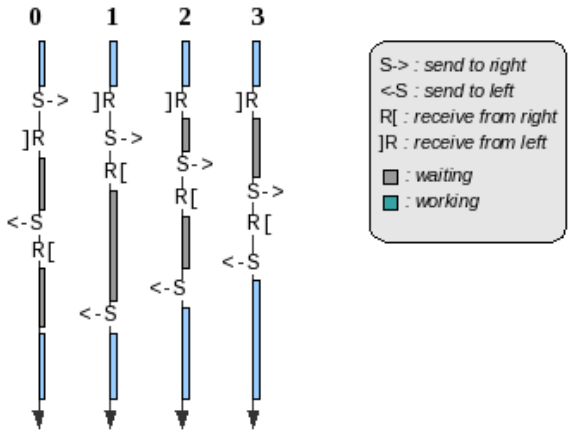


Figura 5: *Un esempio del primo approccio alla sincronizzazione, con 4 PE.*

Tutti i PE tranne quello con il primo blocco si mettono in attesa di ricevere il vettore sinistro, mentre il PE con il primo blocco è il primo ad inviare il suo bordo destro al vicino destro, dopodiché si mette in attesa di ricevere dal vicino sinistro. Mano a mano che un PE finisce di ricevere il suo vettore sinistro, spedisce il bordo destro al vicino a destra, e si rimette in attesa per ricevere il vettore destro dal vicino destro. Il primo ciclo si chiude quando il PE che ha l'ultimo blocco spedisce il suo bordo destro al PE iniziale, a questo punto la catena si inverte e il primo PE spedisce all'ultimo PE il suo bordo sinistro e si mette in attesa di ricevere dal vicino destro. Ricomincia quindi il ciclo inverso, che si conclude con il primo processo che riceve il vettore destro dal vicino destro. È facile vedere che con questo semplice approccio non si potranno mai creare deadlock, inoltre può andare bene per un qualunque numero di PE (maggiore-uguale a 2). È però altrettanto evidente che viene sprecato molto tempo per scambiarsi i dati in questa catena in

cui nessun PE (a parte il primo) spedisce prima di aver ricevuto, che porta ad uno scambio di dati che si potrebbe definire sequenziale, mentre sarebbe intuitivamente possibilie scambiarsi i dati in modo parallelo, cioè mentre due PE stanno trasmettendo tra loro, altri due potrebbero contemporaneamente fare lo stesso, e risparmiare quindi nel tempo totale della sincronizzazione.

Per effettuare uno scambio di dati parallelo adottiamo allora uno schema differenta, che chiameremo pari-dispari, in cui i PE eseguiranno passi diversi a seconda se hanno un blocco pari (blocco numero 0, 2, 4, ecc.) o dispari (blocco numero 1, 3, 5, ecc.):

- PE con blocco pari.
    1. Spedisce il bordo destro al vicino destro.
    2. Riceve il vettore sinistro dal vicino sinistro.
    3. Spedisce il bordo sinistro al vicino sinistro.
    4. Riceve il vettore destro dal vicino destro.
- PE con blocco dispari.
    1. Riceve il vettore sinistro dal vicino sinistro.
    2. Spedisce il bordo destro al vicino destro.
    3. Riceve il vettore destro dal vicino destro.
    4. Spedisce il bordo sinistro al vicino sinistro.

Dopodiché ovviamente ogni PE può eseguire l'iterazione successiva sul suo blocco. In figura 6 si può vedere un esempio con un numero pari e dispari di blocchi. Per leggere a modo la figura bisogna sempre tenere conto che "il mondo gira attorno", percui l'ultimo blocco comunica con il primo blocco, e viceversa, e che le send sono operazioni non bloccanti mentre le receive sono operazioni bloccanti e possono leggere una send anche dopo che è stata inviata (si può pensare che il messaggio venga memorizzato in un buffer locale al PE che lo ha ricevuto, il quale può anche andare a prenderlo in un secondo momento). Ovviamente le cose possono non andare perfettamente come rappresentate in figura, un PE può impiegare più tempo degli altri ad arrivare alla fase di sincronizzazione (ed in pratica si avrà sempre una situazione in cui ogni PE arriverà in tempi diversi), ma questo non comporta problemi, assumendo che non possano esserci errori ed ogni *send* e *receive* vada sempre a buon fine, infatti si ha solamente un rallentamento dei PE vicini, che si mettono in attesa di ricevere i messaggi del PE in ritardo, situazione comunque inevitabile siccome ad ogni iterazione è necessaria una sincronizzazione che dipende dal calcolo di altri PE.

Quindi, con questo tipo di sincronizzazione, si ha la possibilità di scambiare i dati in parallelo, senza deadlock, e sia per un numero pari di processi

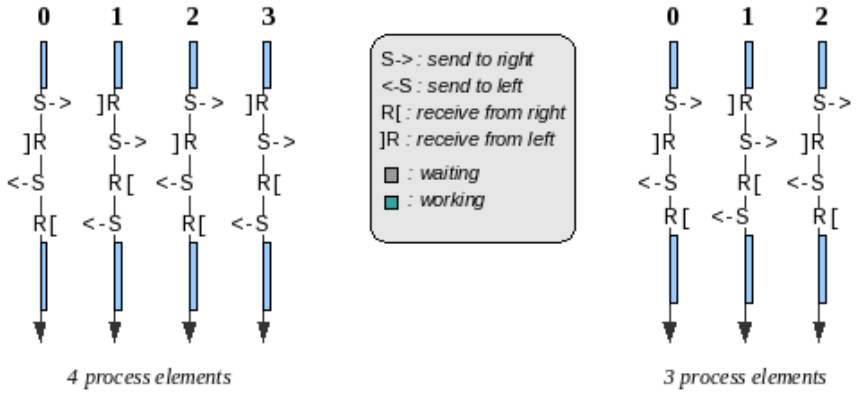


Figura 6: *Esempio di sincronizzazione pari-dispari con 4 e 3 PE.*

che per un numero dispari. Inoltre non sono necessarie barriere alla fine della sincronizzazione, infatti un PE prosegue con l'iterazione *k+1* solamente se ha inviato i suoi bordi e ricevuto i vettori dell'iterazione k, percui se un PE è riuscito ad aggiornare i suoi due vettori può proseguire con l'iterazione successiva senza attendere oltre.

Per i vantaggi portati e per la semplicità della procedura verrà utilizzato questo approccio (pari-dispari) per la sincronizzazione dei bordi fra i PE.

3.3 Lo skeleton: una farm

Lo schema descritto fin'ora consiste in un PE iniziale che suddivide la matrice in blocchi in base a quanti altri PE sono disponibili ad elaborarli, dopodiché invia ad ogni PE un blocco su cui verranno eseguite le n iterazioni dell'algoritmo del gioco della vita; i PE che elaborano i blocchi si sincronizzano ad ogni iterazione, per poi spedire, compute le n iterazioni, i blocchi elaborati ad un PE finale che riassembla la matrice e restituisce il risultato finale dell'intero processo.

Questo schema può essere implementato con una **Farm**, in figura 7, composta da un PE iniziale (**initial stage**), un insieme di PE che chiameremo **workers** che lavorano in parallelo, ed un PE finale (**final stage**). Lo stage iniziale costruisce una serie di blocchi a partire dalla matrice iniziale (funzione `init()`) e li spedisce ai workers che li elaborano applicandogli n iterazioni dell'algoritmo del gioco della vita (`compute()`), in modo parallelo l'uno con l'altro, e sincronizzandosi ad ogni iterazione; infine i workers inviano i blocchi elaborati allo stage finale che li raccoglie e ricostruisce la matrice (`fin()`) che sarà il risultato dell'intero processo.

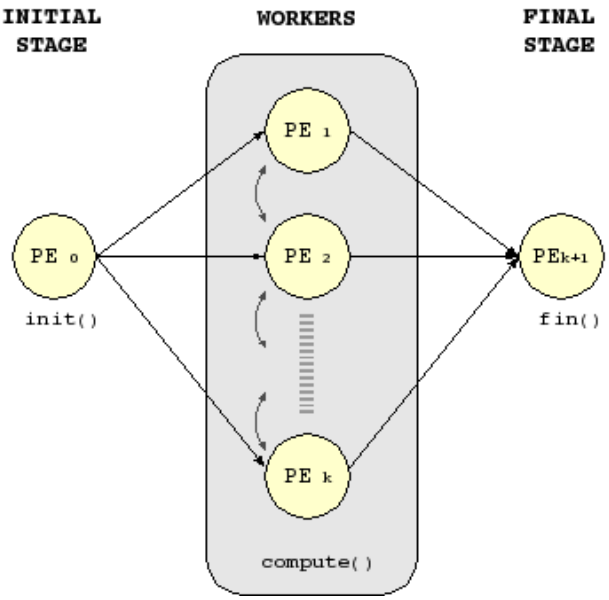


Figura 7: *Una Farm con k workers.*

4 Implementazione

L'applicazione è stata implementata in linguaggio `C++` con il supporto della libreria di skeleton Muesli [3] per la parte di parallelizzazione. Grazie a questa libreria è possibile scrivere un'applicazione parallela come se si stesse scrivendo una normale applicazione sequenziale, in particolare è stata sfruttata la parte di libreria sugli skeleton per implementare la farm. La struttura del programma di per se è molto semplice: la funzione `main` inizializza la matrice iniziale e costruisce una farm utilizzando la libreria Muesli, passandogli una funzione `init()` per lo stage iniziale, una funzione `compute()` per i workers ed una funzione `fin()` per lo stage finale, dopodichè "avvia" l'esecuzione della farm. La funzione `init` divide la matrice iniziale in blocchi ed invia un blocco ad ogni worker. La funzione `compute` (che viene eseguita da ogni worker) esegue n iterazioni sul blocco (con n parametro del programma) sincronizzandosi con gli altri workers alla fine di ogni iterazione. Infine la funzione `fin` riceve i blocchi dai workers e li riassembla costruendo la matrice finale. A supporto di tutto sono state implementate tre strutture dati: `Matrix`, `Block` e `Vector`, per rappresentare rispettivamente la matrice del gioco della vita, i blocchi in cui dividere la matrice e i vettori all'interno dei blocchi che dovranno essere sincronizzati.

4.1 Principali funzioni e strutture dati

Siano n il numero di iterazioni che l'applicazione deve eseguire e w il numero di workers della farm. Di seguito sono riportati i passi in dettaglio di ogni funzione e una descrizione delle tre strutture dati.

int main(int argc, char* argv[])

La funzione `main` compie i seguenti passi:

1. Inizializza gli skeleton della libreria Muesli.
2. Legge ed inizializza i parametri dell'applicazione.
3. Cotruisce la farm utilizzando la libreria Muesli.
4. Avvia l'esecuzione della farm.
5. Se non si sono verificati errori "chiude" la libreria Muesli e termina l'esecuzione.

Block* init(Empty)

La funzione `init` viene eseguita dallo stage iniziale della farm, non prende argomenti in input, il suo output è l'input di un worker (ad ogni invocazione crea un input per un worker differente). La funzione divide la matrice in w blocchi e restituisce un blocco alla volta ogni volta che è invocata, fino a quando non li ha restituiti tutti, a quel punto torna `NULL`.

Block* compute(Block* input)

La funzione `compute` è la funzione eseguita dai workers, prende l'input dallo stage iniziale e restituisce l'output per lo stage finale (funzione `fin`):

1. Comunica con gli altri workers per scoprire chi ha il blocco precedente al suo (il vicino sinistro) e chi ha il blocco successivo (il vicino destro) attraverso la funzione `discoverNeighbors`. Queste informazioni serviranno al momento della sincronizzazione.
2. Per n volte esegue una computazione sul blocco e sincronizza i vettori del blocco comunicando con i workers "vicini".
3. Restituisce il blocco su cui sono state eseguite n iterazioni.

void fin(Block* input)

La funzione `fin` riceve in input i blocchi restituiti dai workers. Questa funzione non fa altro che costruire la matrice finale componendo nell'ordine esatto i blocchi che gli arrivano in input.

void discoverNeighbors()

Questa funzione viene utilizzata da tutti i workers prima di iniziare la computazione per scoprire i PE vicini (ovvero quelli a cui sono stati assegnati i blocchi precedente e successivo al suo). Viene utilizzata la funzione `MPI_Allgather` di MPI per eseguire un broadcast di un vettore di coordinate: ogni PE invia attraverso tale funzione una coppia *<ID del blocco,ID del PE>* e riceverà, sempre dalla stessa funzione, un vettore con le coppie di tutti i workers; sapendo che il vettore contiene una serie di coppie e conoscendo l'ID del proprio blocco, ogni PE effettua una scansione di tale vettore per ricavare l'ID dei processi che hanno il blocco precedente e il blocco successivo.

Matrix

La classe `Matrix` rappresenta la matrice del gioco della vita attraverso un array bidimensionale di `bool`. La classe viene cotruita passandogli le dimensioni della matrice (numero di righe e di colonne) e la densità, un valore tra 0 e 1 che rappresenta la proporzione tra celle vive e celle morte all'interno della matrice; le celle vengono quindi inizializzate in modo casuale tenendo conto della densità, assegnando il valore `true` alle celle vive e `false` alle celle morte.

Attraverso il metodo `getBlock`, specificando il numero di blocchi in cui dividere la matrice e l'indice del blocco che si vuole estrarre dalla matrice, costruisce e restituisce il blocco richiesto in un oggetto di tipo `Block`. Con il metodo `setBlock` prende invece in input un oggetto di tipo `Block` e sostuituisce parte dei suoi elementi con quelli interni al blocco.

Block

La classe `Block` rappresenta un blocco che viene ricavato dalla matrice iniziale ed inviato ai workers per la computazione. Implementa l'interfaccia `MSL_Serializable` per poter essere utilizzato come input e output negli skeleton della libreria Muesli, tale iterfaccia prevede infatti che vengano implementati alcuni metodi che permettono di trasformare l'oggetto in una sequenza di byte tale che possa essere trasmessa tra un PE e l'altro.

Al suo interno è composta da un array bidimensionale di `bool` per mantenere la fetta di matrice interna al blocco, e da due oggetti di tipo `Vector` che rappresentano i vettori sinistro e destro del blocco (che devono essere aggiornati ad ogni iterazione).

Attraverso il metodo `compute` viene eseguita una iterazione dell'algoritmo del gioco della vita sul blocco. Con i metodi `getLeftBoundary` e `getRight Boundary` restituisce rispettivamente il bordo sinistro e il bordo destro del blocco in oggetti di tipo `Vector`; con i metodi `setLeftVector` e `setRight Vector` permette di impostare rispettivamente il vettore sinistro e il vettore

destro interni al blocco. Questi ultimi quattro metodi sono utilizzati per la fase di sincronizzazione.

Vector

Rappresenta un vettore interno ad un blocco, percui non è altro che un array di `bool`. Anche questa classe implementa l'interfaccia `MSL_Serializable` poiché oggetti di questo tipo devono essere scambiati tra i workers durante la fase di sincronizzazione.

4.2 Compilazione ed esecuzione

Essendo l'applicazione basata su Muesli, quindi su MPI, per compilarla ed eseguirla l'applicazione è necessario utilizzare l'ambiente MPICH2 [5] (utilizzato come implementazione di MPI).

Compilazione

Per compilare il programma e creare l'eseguibile `game-of-life` si deve dare il seguente comando all'interno della cartella con i files sorgenti:

```
$ mpicxx -fpermissive -I muesli/ gameoflife.cpp \
      -o ./bin/game-of-life
```

e l'eseguibile viene creato all'interno della cartella `bin`.

Parametri

Per l'esecuzione è necessario passare alcuni parametri obbligatori in input all'applicazione, altri invece sono facoltativi:

-h stampa su standard output l'elenco dei parametri dell'applicazione.

-r numero di righe della matrice (obbligatorio).

-c numero di colonne della matrice (obbligatorio).

-d desità delle celle vive nella matrice iniziale (numero tra 0 e 1).

-i numero di iterazioni da eseguire (default 1).

-p stampa su standard output la matrice iniziale e la matrice finale.

-t stampa su standard output i tempi di esecuzione di ogni PE.

Esecuzione su una macchina

Per eseguire l'applicazione su una singola macchina (con più processi paralleli) si deve innanzitutto avviare il demone di mpich2.

```
$ mpd &
$ mpdtrace
```

Con il primo comando si avvia il demone, con il secondo si controlla che sia effettivamente in esecuzione, e dovrebbe restituire il nome della macchina su cui è stato lanciato. Dopodichè si può eseguire il programma con

```
$ mpiexec -n <numero processi> ./game-of-life <parametri>
```

specificando il numero di processi che si vogliono coinvolgere nell'esecuzione e i parametri dell'applicazione. Il numero di processi deve essere almeno 3: uno per lo stage iniziale, uno per lo stage finale, ed almeno un worker. Con il comando

```
$ mpdallexit
```

si termina il demone `mpd`.

Esecuzione su più macchine

Per eseguire l'applicazione su più macchine bisogna semplicemente configurare mpich2 per farlo, di seguito sono descritti brevemenți i passi necessari, ma maggiori dettagli (soprattutto sull'esatta configurazione delle macchine coinvolte) si possono trovare in modo dettagliato nella documentazione di mpich2 [6].

Innanzitutto si deve creare il file `mpd.hosts` nel quale vanno messi i nomi delle macchine sulle quali si vuole eseguire l'applicazione[3], con un nome per riga. Dopodiché si può lanciare il demone `mpd` su tali macchine con il comando

```
$ mpdboot -n <numero di macchine> -f mpd.hosts
```

dove il numero di macchine dev'essere minore o uguale a 1 più il numero di macchine nel file mpd.hosts. Con il comando `mpdtrace` si può verificare che il demone sia stato lanciato correttamente su tutte le macchine, a tale comando dovrebbe corrispondere la lista degli hostname delle macchine coinvolte.

Se il demone è in esecuzione correttamente su tutte le macchine, l'applicazione si può eseguire allo stesso modo che in una singola macchina, con il comando

```
$ mpiexec -n <numero processi> ./game-of-life <parametri>
```

[3] Ovviamente tali macchine devono essere opportunamente configurate [6]

Infine, con il comando

```
$ mpdallexit
```

si terminano i demoni `mpd` su tutte le macchine.

5 Risultati

Possiamo concludere con alcune prove dell'applicazione e la discussione dei risultati ottenuti.

Innanzitutto si può provare l'applicazione senza parallelismo, eseguendola su una sola macchina, con un solo workers (quindi con tre processi, che è il minimo richiesto dall'applicazione) e con una matrice di piccole dimensioni che verrà stampata in output.

La matrice iniziale viene generata in modo casuale dal programma, non è quindi possibile fornire in input lo stato iniziale della matrice, l'unico parametro che si può stabilire è la densità di celle vive nella matrice iniziale (opzione `-d`), un valore tra 0 e 1 che stabilisce la percentuale di celle vive (con 1 la matrice avrà solo celle vive, con 0 solo celle morte). Per verificare che l'applicazione calcoli effettivamente il gioco della vita si possono specificare poche iterazioni da compiere, dopodiché si può confrontare la matrice finale stampata in output con una calcolata manualmente o con l'uso di altre applicazioni[4] a partire dalla stessa configurazione iniziale.

Un'altro modo per verificarne il corretto funzionamento è basarsi sul fatto che, dopo un numero sufficente di iterazioni, il gioco della vita tende a stabilizzarsi su configurazioni già note, chiamate patterns, come ad esempio i *blocchi* (un quadrato di due celle di lato) oppure i *lampeggiatori* (tre celle consecutive su una stessa riga o colonna), anche se in generale ne esistono svariate centinaia[5]; si possono quindi specificare molte iterazioni (ad esempio 1000) e vedere che, se non muoiono tutte le celle, la matrice finale sarà sempre composta da un insieme di questi patterns.

Dopo aver avviato il demone `mpd`, si può quindi provare l'applicazione con il comando

```
$ mpiexec -n 3 ./game-of-life -r 12 -c 18 -d 0.4 -i 1000 -p
```

che calcola mille generazioni a partire da una matrice iniziale 12x18 con densità 0.4, stampando in output la matrice iniziale e la matrice finale (opzione `-p`):

```
3 process elements, 1 workers.
12x18 matrix, with density 0.4.
```

[4] Esistono molte applicazioni web, implementate in javascript, che calcolano il gioco della vita. Una di queste si può trovare in [2].

[5] Un elenco esaustivo ed illustrato di tutti i patterns conosciuti si può trovare in [1].

```
1000 iterations to compute.

[*][ ][ ][ ][ ][ ][ ][ ][*][ ][*][*][ ][ ][ ][ ][ ]
[ ][ ][ ][*][*][*][ ][*][*][ ][*][ ][ ][ ][*][*][*][*]
[ ][ ][ ][ ][ ][ ][*][ ][ ][*][ ][ ][ ][ ][*][*][*][*][ ][*]
[*][*][ ][ ][ ][ ][ ][*][ ][*][ ][ ][ ][ ][ ][*][*][*][ ]
[*][*][ ][ ][ ][ ][*][*][*][ ][*][*][*][ ][*][ ][*][ ][*]
[ ][*][ ][*][ ][ ][ ][ ][*][ ][ ][ ][ ][ ][ ][ ][*][*][ ]
[ ][ ][ ][*][ ][ ][ ][ ][*][ ][ ][*][ ][*][ ][ ][*][*][*]
[ ][ ][ ][*][*][*][ ][ ][*][ ][*][ ][ ][*][*][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][*][*][ ][ ][ ][*][ ][*][ ][ ][ ][ ][ ]
[*][ ][ ][*][ ][*][ ][ ][*][ ][ ][ ][ ][*][*][ ][ ][*][ ]
[ ][*][ ][ ][*][*][*][ ][*][ ][ ][*][ ][ ][ ][ ][*][*]
[ ][ ][*][*][ ][ ][ ][ ][*][ ][ ][*][*][ ][ ][ ][ ][ ]

[ ][ ][ ][ ][ ][ ][*][*][*][ ][ ][ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
[*][*][ ][ ][ ][ ][ ][ ][ ][*][*][ ][ ][ ][ ][ ][ ][ ]
[ ][ ][*][ ][ ][ ][ ][ ][ ][*][ ][ ][*][ ][ ][ ][ ][*]
[*][*][ ][ ][ ][ ][ ][ ][ ][*][*][ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][*][*][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][*][*][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
```

Si possono notare nella matrice finale (separata da uno spazio da quella iniziale) diversi patterns del gioco della vita: un *blocco* (in basso sulla sinistra), un *lampeggiatore* (in alto al centro) e due *alveari* (nel mezzo della matrice).

Nelle prove successive, come nel comando precedente, verrà sempre usata una desità di 0.4, questo non influisce sui calcoli da compiere poiché l'algoritmo prevede di guardare sempre tutte le celle della matrice indipendentemente che siano vive o morte.

5.1 Misurazione dei tempi

Attraverso l'opzione `-t` si può specificare di stampare su standard output i tempi di esecuzione dell'applicazione. Ogni PE cronometra il suo tempo effettivo di esecuzione e il tempo di utilizzo della cpu. Questi due tempi non sempre coincidono, infatti il tempo effettivo di esecuzione dipende da quanto altro lavoro ha da eseguire la macchina su cui viene eseguita l'applicazione, che nel caso sia impegnata ad eseguire anche altri processi potrà dedicare

meno tempo all'esecuzione della nostra applicazione, e si avrà un tempo di esecuzione relativamente alto, mentre l'utilizzo di cpu sarà molto minore. Come si vedrà meglio in seguito questo caso si verifica in particolare quando si esegue l'applicazione su una singola macchina con più processi paralleli.

Il PE dello stage iniziale, i PE dei workers e il PE dello stage finale misurano tempi di esecuzione differenti:

- Lo stage iniziale, che sarà il PE 0, misura il tempo dalla creazione della matrice iniziale fino a che non ha inviato tutti i blocchi ai workers.
- Lo stage finale, che sarà l'ultimo PE, misura il tempo dal momento in cui viene lanciato fino a quando ha ricevuto tutti i blocchi elaborati dai workers e costruito la matrice finale.
- I workers misurano il tempo impiegato a compiere le n iterazioni sul loro blocco.

Come indicazione del tempo totale di esecuzione dell'applicazione viene preso il tempo restituito dallo stage finale, questo tempo infatti parte dall'inizio dell'esecuzione fino alla fine dell'intero processo, ed utilizza la cpu per tutto l'arco del tempo.

5.2 Esecuzione distribuita su più macchine

In questa prova si passa all'esecuzione dell'applicazione su più macchine, all'interno di una rete locale. Al momento delle prove il traffico nella rete era minimo e le macchine utilizzate hanno le stesse caratteristiche:

- CPU: Intel Pentium Dual CPU 1.80GHz
- Memoria: 2 GB

I test effettuati sono due: con una matrice di grandi dimensioni (10.000x 10.000) e poche iterazioni (100), di cui l'andamento all'aumentare delle macchine coinvolte è rappresentato in figura 8, e con una matrice di piccole dimensioni (1.000x1.000) ma con tante iterazioni da eseguire (10.000), con l'andamento raffigurato nel grafico in figura 9. Lo scopo della prova chiaramente è verificare un miglioramento dei tempi di esecuzione aumentando il numero di workers coinvolti. Dal grafico del primo test si vede come inizialmente, passando da 1 worker a 2 workers, si hanno netti miglioramenti nel tempo di esecuzione totale, ma questi miglioramenti sono sempre minori aumentando il numero di workers. Questo andamento è del tutto normale, infatti passare da un solo worker che deve lavorare sull'intera matrice, a due workers che invece devono eseguire la metà dei conti, comporta ovviamente quasi un dimezzamento dei tempi di calcolo; se invece si passa da nove workers a dieci, ogni workers avrà comunque meno lavoro da fare, ma con un calo meno significativo rispetto al dimezzamento del passaggio da uno a due. Se

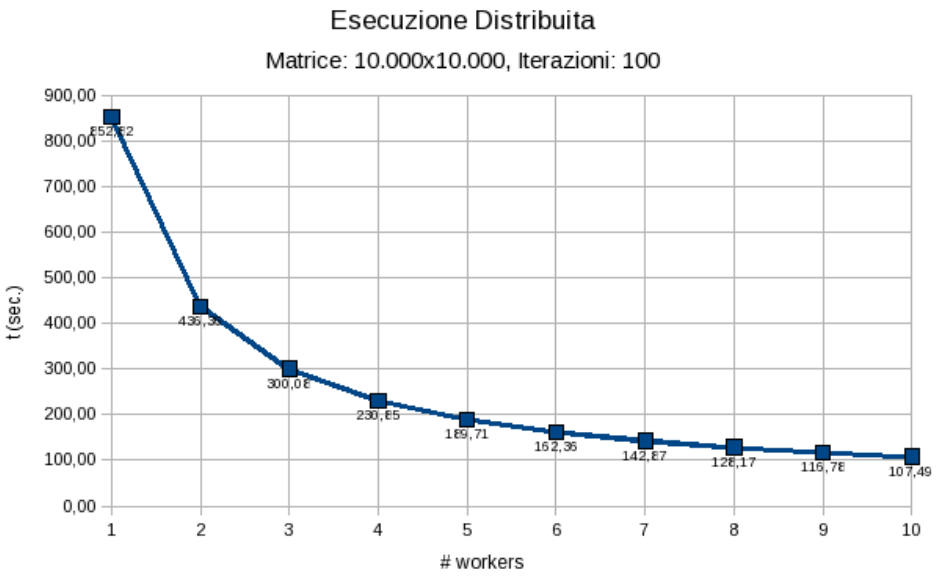


Figura 8: *Esecuzione distribuita: primo test (matrice grande, poche iterazioni).*

ad esempio prendiamo una matrice da 100 colonne, passando da un worker a due, si passa da un PE che lavorava su 100 colonne a due che lavorano su 50 (il carico dimezza), ma se passiamo da nove workers a dieci, si passa da nove PE che avevano circa 11 colonne a dieci che ne hanno 10, percui il carico di lavoro ha un calo minimo. Inoltre si deve considerare che più workers significa anche più operazioni di sincronizzazione ad ogni iterazione, percui aumentando i workers si ha un piccolo rallentamento che si deve sottrarre all'aumento di prestazioni dovuto alla distribuzione del carico di lavoro.

Il secondo grafico segue un andamento del tutto analogo al primo, questo mostra il fatto che le operazioni di sincronizzazione, che in questa prova sono 10.000 per ogni coppia di workers, sono veloci e non portano ad un visibile rallentamento del calcolo parallelo.

Da notare che in entrambi i grafici sono riportati solamente i tempi effettivi di esecuzione e non i tempi di utilizzo della cpu, questo perché i due tempi, in entrambe le prove, sono stati sempre praticamente uguali. Bisogna infatti considerare che in questi casi si ha una macchina dedicata per ogni workers, che dedica il suo processore solamente al calcolo della nostra applicazione. Percui, nel caso di sistemi distribuiti su più macchine, si riescono ad ottenere miglioramenti reali ed effettivi aumentando il numero di workers impiegati. Da fare però particolare attenzione al fatto che se una macchina è più lenta rispetto alle altre, oppure ha altro lavoro da eseguire e non riesce a dedicare a pieno il tempo di utilizzo della cpu all'applicazione, tale macchina porterebbe ad un rallentamento dell'intero processo, dovuto al fatto che è

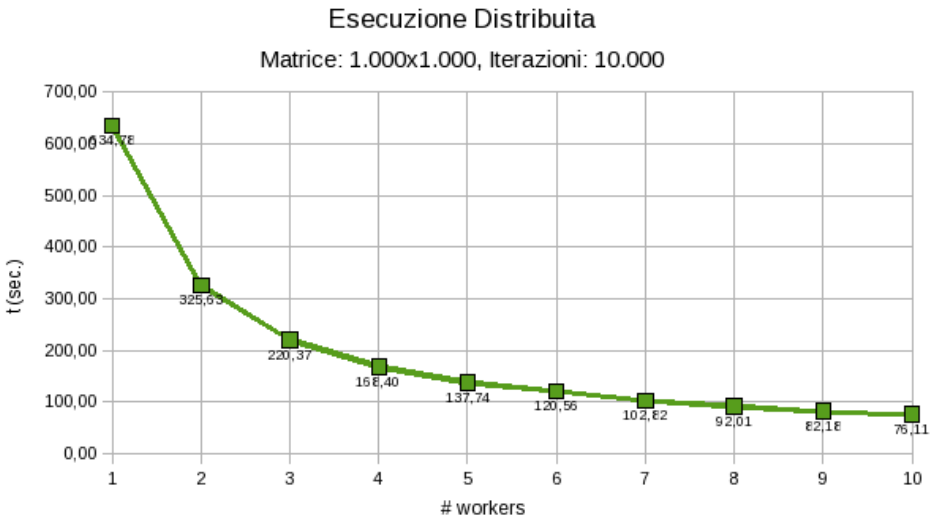


Figura 9: *Esecuzione distribuita: secondo test (matrice piccola, molte iterazioni).*

necessaria la fase di sincronizzazione al termine di ogni iterazione.

5.3 Esecuzione a più processi su una macchina

In questa prova viene eseguita l'applicazione in una singola macchina, mantenendo invariate le dimensioni della matrice e le iterazioni da eseguire, ed aumentando il numero di workers coinvolti nella computazione con l'obiettivo di diminuire il tempo totale di esecuzione. Ogni workers (oltre allo stage iniziale e finale) sarà quindi un processo parallelo e concorrente rispetto agli altri, eseguito su un solo processore della macchina. Il computer utilizzato ha le seguenti caratteristiche:

- CPU: Intel Core 2 Duo CPU 2.80GHz
- Memoria: 2 GB

Nel grafico in figura 10 sono riportati i tempi effettivi e i tempi di utilizzo della cpu all'aumentare del numero di workers impiegati. Si nota subito che i tempi di utilizzo della cpu sono più bassi e seguono un andamento più lineare, mentre il tempo effettivo di esecuzione ha un brusco calo passando da uno a due workers, ma poi diminusce in modo minimo all'aumentare del numero di workers. Questo andamento è dovuto al fatto che i processi vengono eseguiti su un singolo processore dual core, percui due processi possono essere effettivamente eseguiti in modo parallelo su i due core del processore, e da qui il miglioramento passando da 1 a 2 workers, ma il parallelismo diventa "virtuale" per più di 2 processi, che verrano fatti eseguire a turno

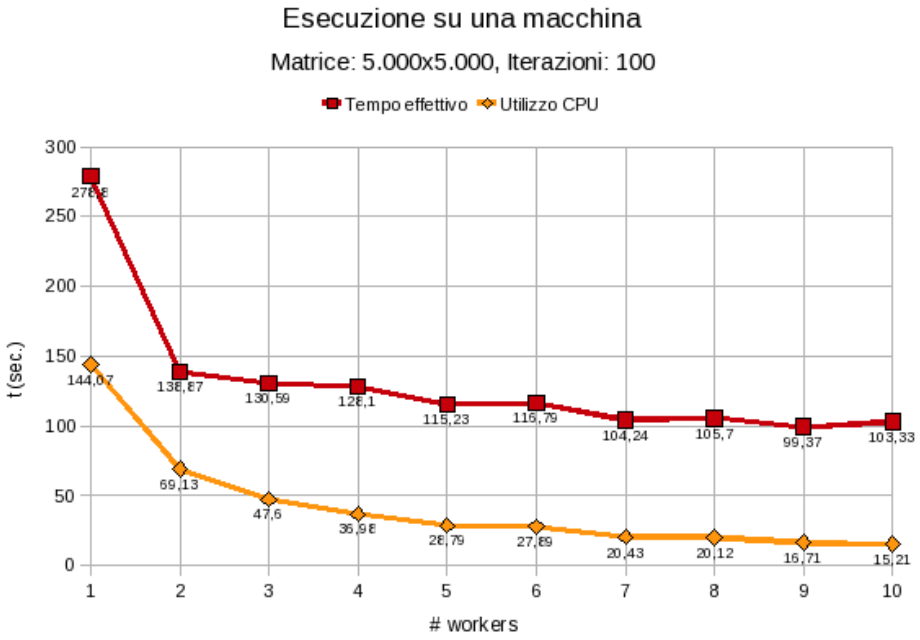


Figura 10: *Esecuzione dell'applicazione su una singola macchina.*

sul processore con la politica di scheduling del sistema operativo. Mentre invece il tempo di utilizzo della cpu segue un andamento del tutto analogo a quello su più macchine distribuite, poiché in tal caso viene misurato il lavoro compiuto da ogni processo e non il tempo reale impiegato. Percui, seppur l'impiego di cpu diminuisce aumentando il numero di processi, su una singola macchina si possono ottenere miglioramenti effettivi dalla parallelizzazione solamente utilizzando tanti workers quanti sono i core (o i processori) della macchina stessa, altrimenti il tempo totale dell'esecuzione non diminuisce, ma anzi, può addirittura aumentare a causa dei tempi persi nella sincronizzazione o per piccoli effetti di starvation dovuti ad uno scheduling non adatto del sistema operativo.

Riferimenti bibliografici

[1] LifeWiki, the wiki for Conway's Game of Life. http://www.conwaylife.com/wiki/index.php

[2] Simulazione del Game of Life in javascript. http://www.gwydir.demon.co.uk/giles/javalife.htm

[3] The Munster Skeleton Library Muesli Homepage. http://www.wi.uni-muenster.de/pi/forschung/Skeletons/index.html

[4] P. Ciechanowicz, M. Poldner, H. Kuchen, "The Munster Skeleton Library Muesli - A Comprehensive Overview. http://www.wi.uni-muenster.de/imperia/md/content/erciscontent/publications/ercis_wp_no_7.pdf

[5] MPICH2 Homepage. http://www.mcs.anl.gov/research/projects/mpich2/index.php

[6] MPICH2 Installer's Guide. http://www.mcs.anl.gov/research/projects/mpich2/documentation/files/mpich2-1.2.1-installguide.pdf