PC-2021/22 Random maze solver

Prasanna Silva
hondamunige.silva1@stud.unifi.it

Abstract

Il metodo più semplice, ma poco efficiente, per risolvere un labirinto è cercare la soluzione in modo del tutto casuale. Questo report si concentra sull'implementare ed analizzare l'algoritmo di risoluzione di labirinti in modo casuale parallelizzato tramite OpenMP.

1. Introduction

La risoluzione di un labirinto in maniera casuale è un algoritmo che non è scalabile in dimensione, vedremo se è possibile migliorare le prestazioni tramite la parallelizzazione. In questo report analizzeremo le prestazioni di un algoritmo di risoluzione casuale di labirinti paralizzato tramite l'utilizzo del processore, attuando quindi lo stesso algoritmo contemporaneamente su più thread. L'algoritmo è (quasi) **"embarrassingly parallel"**, perchè è possibile risolvere il labirinto paralellamente da piu 'particelle', ma le particelle neccessitano di una communicazione.

2. Implementazione

2.1. Generazione di labirinti

Per provare l'algoritmo parallelizzato i labirinti sono stati creati utilizzando l'algoritmo Randomized depth first search con implementazione interattiva, in seguito è stato presentato lo pseudocodice dell'algoritmo. Una volta creato, il labirinto viene memorizzato su un'array di carattere.
I labirinti scelti per testare l'algoritmo sono labirinti quadrati di grandezza **15, 25, 30, 35, 40**

Algorithm 1 Randomized depth-first search

Choose the initial cell
mark it as visited and push it to the stack
while The stack is not empty **do**
    Pop a cell from the stack and make it a current cell
    if The current cell has any neighbours which have not been visited **then**
        1.Push the current cell to the stack
        2.Choose one of the unvisited neighbours
        3.Remove the wall between the current cell and the chosen cell
        4.Mark the chosen cell as visited and push it to the stack
    end if
end while

2.2. Algoritmo risolutore di labirinti

Per risoluzione casuale si intende che la scelta presa dall'algoritmo a un dato punto del labirinto risulti senza nessun criterio e puramente casuale. A ogni passo dell'algoritmo seleziona una delle quattro celle vicine, se la cella selezionata non è un ostacolo allora la cella selezionata viene inserita all'interno dello stack per il path. I vicini della cella selezionata vengono poi inseriti

nello stack dei vicini, altrimenti l'algoritmo rimbalza sull'ostacolo e torna alla cella precedente. L'algoritmo si ripete fin quando la cella scelta non è la cella d'arrivo.

Algorithm 2 Sequential Maze solver

```
Choose the initial cell
Push it to the stack
Select the neighborhood of the initial cell and push to the stack
while the current cell is not the target cell do
    Select a random neighbour of the current cell
    if neighbour is not a wall then
        Push on the path stack
        Clear the stack
        Select the neighborhood of the current cell
        Push to the stack
    else if neighbour is a wall then
        Bounce on the wall and return to previous cell
    end if
end while
```

L'algoritmo 2 mostra il codice di risolutore di labirinto nella versione sequenziale.

La fig. 1 mostra il tempo di risoluzione al crescere della grandezza del labirinto utilizzando la versione sequenziale.

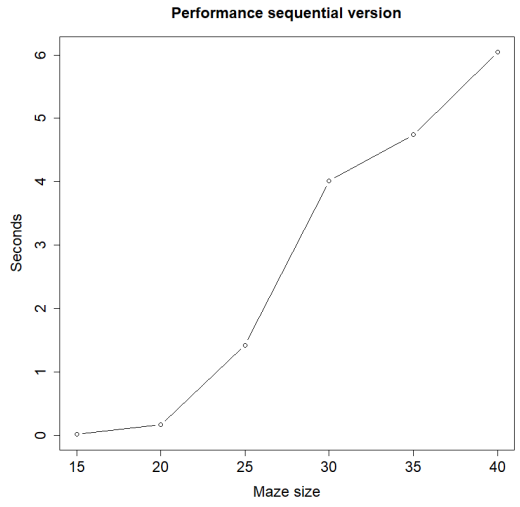

Figure 1. Prestazione versione sequenziale

2.3. Algoritmo risolutore di labirinti parallelizzato

L'idea è di far eseguire tale codice sequenziale su tutti i thread contemporaneamente; quando un thread trova la soluzione la comunica agli altri che pertanto possono smettere di cercarla. Per fare ciò utilizziamo una variabile condivisa tra i thread, posta inizialmente a false e appena un thread risolve il labirinto la variaibile viene posta a true. Ogni ciclo dell'algoritmo di risoluzione del labirinto controlla che la variabile condivisa sia settata a true, se si verifica tale condizione la ricerca si ferma.

Algorithm 3 Parallel Maze solver

```
Choose the initial cell
Push it to the stack
Select the neighborhood of the initial cell and push to the stack
while the current cell is not the target cell do
    if found is true then
        break
    end if
    Select a random neighbour of the current cell
    if neighbour is not a wall then
        Push on the path stack
        Clear the stack
        Select the neighborhood of the current cell
        Push to the stack
    else if neighbour is a wall then
        Bounce on the wall and return to previous cell
    end if
end while
```

```
1  #pragma omp parallel shared(maze, start_position,finish_position,
2              column, row, found) private(path)
3  {
4      bool resolved = false;
5      int id = omp_get_thread_num();
6
7      resolved = MazeSolver(maze, path,start_position,finish_position,
8                      column, row,found);
9
10
11     #pragma omp critical
12     {
13         if(resolved && !found)
14         {
15             found = true;
16             cout << "Thread "<<id<<" found the path"<<endl;
17             PathDisplay(maze, path);
18             drawMaze(maze, row, column);
19         }
20     }
21 }
```

Listing 1. Parallelizzazione

Il codice mostra la parallelizzazione dell'algoritmo di risoluzione su più thread tramite OpenMP . Le variaibili *maze, start_position, finish_position, column , row e* ***found*** sono condivise tra i thread. Tutte tranne ***found*** non vengono

modificate dai thread, infatti, l'unica variaibile che viene modificata da un thread, qualora risolvesse il labirinto, è *found*.

Tale variabile viene passata alla funzione *MazeSolver* ossia l'algoritmo 3 presentato in preccedenza. Quando un thread risolve il labirinto entra in una sezione critical dove pone la variaible *found* a *true*, avvertendo così gli altri thread che smettereanno di cercare la soluzione. La variabile *Path* è privata dato che ogni thread ha un suo percorso e vogliamo visualizzarla una volta trovata. La sezione critical, non è per la modifica della variaibile condivisa *found*, ma per far stampare il percorso del thread che ha trovato la soluzione.

La Fig. 2 mostra che il tempo di risoluzione, al crescere della grandezza del labirinto, mantiene un andamento simile a quello sequenziale. Nella sezione Prestazioni verrà mostrato una comparazione tra le due versioni.

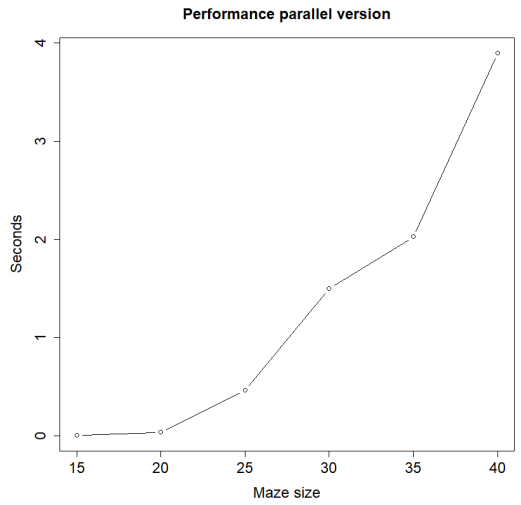

Figure 2. Prestazione versione parallela

3. Prestazioni

In questa sezione mostreremo le prestazioni delle due versioni e eseguiremo dei test per quanto riguarda la versione parallelizzata con un numero di thread differenti. Nello specifico, utilizzeremo 4,8,16,32 thread. La macchina su cui vengono fatti i test ha 8 core in totale, pertanto, l'utilizzo di 16 e 32 thread significa che ci sarà un incremento di schedulazione che riteniamo peggiori le prestazioni. Le grandezze dei labirinti utilizzate per calcolare le prestazioni sono 15x15, 20x20,25x25,30x30,35x35, 40x40. Ogni prova è stata effettuata cinquanta volte, viene poi fatta la media. La risoluzione dell'algoritmo è del tutto casuale, pertanto, è necessario effettuare molteplici test per avere un tempo significativo. Non sono state utilizzate grandezze superiori a 40 x 40 poichè possiamo supporrere che l'andamento sia simile a quello visto per i labirinti 35x35 e 40x40, quindi sempre crescente.

3.1. Confronto tra sequenziale e parallellizzazione

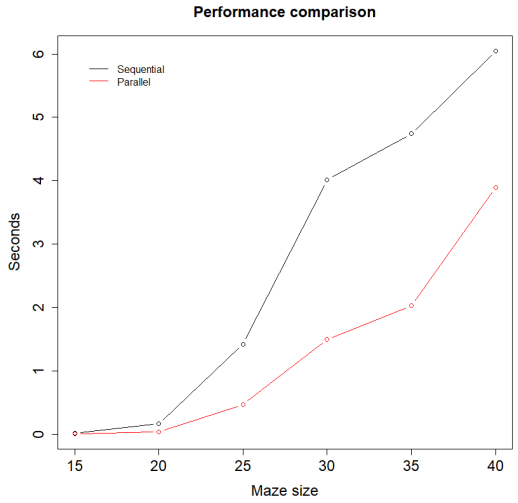

Figure 3. Prestazione

La Fig. 3 mostra la comparisione tra la versione parallel e quella sequenziale, si nota che quella parallelizzata ha un andamento simile a quella sequenziale e che rimane sempre sotto la curva della sequenziale. Inoltre, fino a labirinti 20 x 20 di grandezza non si nota differenze significativa.

3.2. Analisi su numero thread variaibile

Eseguiamo un'analisi in cui si utilizza la versione parallelizzata con 4, 8, 16 e 32 thread. Si osserva che i labirinti di grandezza fino 25x25 hanno tutti lo stesso andamento. Invece, per labirinti più grandi di 25x25 si nota che l'incremento della schedulazione, dovuta all'utilizzo di un numero di thread non disponibile fisicamente, crea un rallentamento sempre più grande. In-

Miglioramento delle prestazioni				
Labirinti	4 Thread	8 Thread	16 Thread	32 Thread
15x15	+275%	+242%	+225%	+223%
20x20	+533%	+422%	+603%	+824%
25x25	+198%	+307%	+358%	+281%
30x30	+357%	+268%	+176%	+155%
35x35	+217%	+234%	+139%	+96%
40x40	+198%	+155%	+150%	+95%

Table 1. Miglioramento rispetto alla versione sequenziale.

oltre osserviamo che un numero di thread minore rispetto ai thread disponibile riesce a dare delle prestazioni migliori; questo è dovuto molto probabilmente al minor tempo di creazione e distruzione dei thread.

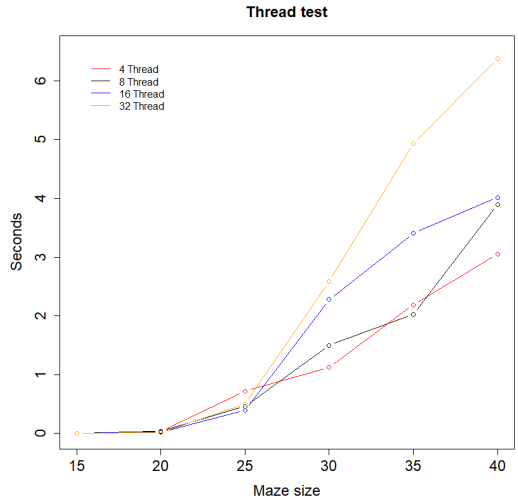

Figure 4. Prestazione

4. Conclusione

In conclusione, nella tab. 1, riportiamo l'incremento delle prestazioni dovute alla parallelizzazione rispetto a quelle nella versione sequenziale. La versione con 8 thread riporta una miglioramento del x2 in media.