Forest-fire-simulation

Indice

Introduzione

Questo progetto è una possibile implementazione dell'algoritmo Forrest fire in C che utilizza Open MPI, una implementazione di MPI. Tale problema è ben descritto qui: https://en.wikipedia.org/wiki/Forest-fire_model. In breve, questa implementazione parallelizza la simulazione di un incendio boschivo, con i seguenti requisiti:

1. Una cella bruciata si traforma in una cella vuota.
2. Un albero brucia se almeno uno dei suoi vicini sta bruciando.
3. Un albero si incendia con probabilità b anche se nessun vicino sta bruciando.
4. In una cella vuota può nascere un albero con probabilità p.

La simulazione può terminare o quando tutta la foresta è vuota, oppure quando si raggiunge un numero di iterazioni S dell'algoritmo (dato in input dall'utente al programma).

Descrizione della soluzione proposta

La soluzione proposta prevede che l'utente dia in input la dimensione della matrice (N ed M) e il numero di step dell'algoritmo (S). Il processo master (rank 0$, inizializza la matrice riempiendo casualmente una cella con una dei seguenti valori:

- T: nella cella c'è un albero.
- E: la cella è vuota.
- B: l'albero sta bruciando.

Il master node poi procede a dividere in modo equo le righe della matrice fra i processi disponibili ed ogni processo lavorerà su una porzione della matrice. Ogni processo poi comunica con i processi vicini le righe agli estremi della sottomatrice ricevuta, in modo da verificare se l'albero dovrà bruciare oppure no. Se la sottomatrice ricevuta da un processo è almeno 3 righe, allora il processo potrà procedere a computare una parte della simulazione senza fare comunicazione (tranne quando andrà a considerare i bordi), questo meccanismo permette di velocizzare la simulazione. Ogni processo poi terrà conto ad ogni iterazione delle celle vuote e le comunicherà al master node, che farà terminare la simulazione quando il numero di celle vuote è pari al numero di celle totali della matrice. Altrimenti la simulazione termina al raggiungimento di S.

Dettagli dell'implementazione

In questa sezione saranno dettaglite le parti più significative del codice dell'implementazione.

Inizializzazione della foresta

```
void forest_initialization(char *forest,int num_row,int num_col){
    for(int i = 0; i<num_row; i++){
        for(int j = 0; j<num_col; j++){
            int rand_num = 1 + (rand() % 100);
            if(rand_num > 70)
                forest[i* num_col + j] = 'T';
            else if(rand_num <= 50)
                forest[i * num_col + j] = 'E';
            else
                forest[i * num_col + j] = 'B';
        }
    }
}
```

Il seme utilizzato da rand è il tempo, allo scorrimento di ogni cella viene generato un numero compreso fra 0 e 100 e se il numero è maggiore di 70 viene inserito un albero, se è minore o uguale a 50 sarà una cella vuota altrimenti sarà un albero che sta bruciando.

Comunicazione asincrona dei bordi delle sottomatrici

```
        if(myrank == 0){
            //invio l'ultima riga al processo successivo
            MPI_Isend(&sub_forest[send_counts[myrank] - n],n,MPI_CHAR,myrank +
1,0,MPI_COMM_WORLD,&request[0]);
            //ricevo la prima riga dal processo successivo
            MPI_Irecv(bottom_row,n,MPI_CHAR,myrank +
1,0,MPI_COMM_WORLD,&request[1]);
        } else if(myrank == numtasks - 1){
            //invio la prima riga al processo precedente
            MPI_Isend(sub_forest,n,MPI_CHAR,myrank -
1,0,MPI_COMM_WORLD,&request[1]);
            //ricevo ultima riga dal processo precedente
            MPI_Irecv(top_row,n,MPI_CHAR,myrank -
1,0,MPI_COMM_WORLD,&request[0]);
        } else { // tutti gli altri processi
            //invio la prima riga al processo precedente
            MPI_Isend(sub_forest,n,MPI_CHAR,myrank
-1,0,MPI_COMM_WORLD,&request[1]);
            //invio l'ultima riga al successivo
            MPI_Isend(&sub_forest[send_counts[myrank] - n],n,MPI_CHAR,myrank +
1,0,MPI_COMM_WORLD,&request[0]);

            //ricevo la riga superiore dal processo precedente
            MPI_Irecv(top_row,n,MPI_CHAR,myrank -
1,0,MPI_COMM_WORLD,&request[0]);
```

```
            //ricevo la riga inferiore dal processo successivo
            MPI_Irecv(bottom_row,n,MPI_CHAR,myrank +
1,0,MPI_COMM_WORLD,&request[1]);
        }
```

La comunicazione scelta è non bloccante, perchè nel mentre vengono scambiati i messaggi fra i processi, è possibile fare della computazione (non sempre), andando quindi a rendere più efficiente l'algoritmo e poi sincronizzare i processi solo quando sono necessari i messaggi. Ogni processo, tranne il processo con rank 0 e l'ultimo (rank = numtasks - 1), inviano la prima riga al processo precedente, in quanto ha bisogno del bordo inferiore per controllare i vicini, mentre inviano l'ultima riga al successivo, in quanto ha bisogno del bordo superiore per controllare i vicini. In ricezione invece, l'n-esimo processo deve ricevere dal processo precedente la riga superiore e dal processo successivo la riga inferiore. I valori ricevuti vengono salvati rispettivamente in `top_row` e `bottom_row` per essere utilizzati successivamente per il controllo. Il processo con rank 0 invece riceve solamente il bordo inferiore e invia l'ultima riga della sua sotto matrice al processo successivo, mentre l'ultimo processo invia la prima riga della sua sottomatrice al processo precedente e riceve l'ultima riga del processo precedente.

Calcolo della simulazione indipendente dalle righe ricevute

Questa parte del codice sarà eseguita senza aspettare che le send e le receive siano terminate. In particolare, ogni processo potrà fare una computazione in maniera indipendente dagli altri processi, se la sottomatrice che ha ricevuto ha più di 3 righe. Poichè in questo caso il controllo delle celle delle righe centrali non dipende dai bordi della matrice che devo ricevere dagli altri processi.

```
int my_row_num = send_counts[myrank] / n;

if(my_row_num >= 3){ // se ho più di 3 righe possso iniziare già a computare le
righe non ai bordi
            for(int i = 1; i<my_row_num - 1; i++){
                for(int j = 0; j<n; j++){
                    if(sub_forest[i * n + j] == 'B')
                        sub_matrix[i * n + j] = 'E'; //Se è già Burned allora ora
la cella diventa vuota
                    else if(sub_forest[i * n + j] == 'E'){
                        int rand_num = 1 + (rand() % 100); // se è vuota, allora
con probabilità prob_grown può crescere un albero nella cella
                        if(rand_num <= prob_grow){
                            sub_matrix[i * n + j] = 'T';
                        } else
                            sub_matrix[i * n + j] = 'E';
                    }else if(sub_forest[i * n + j] == 'T'){

check_neighbors(sub_forest,sub_matrix,my_row_num,n,i,j,prob_burn);
                    }
                }
            }
        }
```

Quindi partendo dalla seconda riga della matrice, fino alla penultima, posso già calcolarmi tutti i valori delle celle. Per il calcolo dei valori della foresta nell'iterazione i-esima, viene utilizzata una matrice di supporto `sub_matrix` dove vengono inseriti i nuovi valori della martrice, che al termine dell'iterazione sarà scambiata con `sub_forest` per il nuovo step. Nel caso in cui nella cella che stiamo considerando abbiamo un albero, allora viene chiamata la funzione `check_neighbors`.

```
void check_neighbors(char *forest,char *matrix2,int num_row,int num_col,int i,int j,int prob_burn){
    //controllo a destra e a sinistra

    //se controllando l'elemento a destra sono ancora della riga (quindi il resto della divisione deve essere 0 altrimenti vuol dire che sono all'ultimo elemento)
    if(((i * num_col)  + j + 1) % num_col != 0 && forest[((i * num_col) + j + 1)] == 'B')
        matrix2[i * num_col + j] = 'B';
    //se controllando l'elemento a sinistra sono ancora della riga (quindi il resto della divisione deve essere diverso da n-1 altrimenti vuol dire che sono al primo elemento della riga)
    if(((i * num_col)  + j - 1) % num_col != num_col-1 && forest[((i * num_col) + j - 1)] == 'B')
        matrix2[i * num_col + j] = 'B';

    //controllo sopra e sotto

    //verifico se il vicino della riga inferiore sta bruciando
    if(((i * num_col) + j + num_col) < num_row * num_col && forest[((i * num_col) + j + num_col)] == 'B')
        matrix2[i * num_col + j] = 'B';

    //verifico se il vicino della riga superiore sta bruciando
    if(((i * num_col) + j - num_col) >=0 && forest[((i * num_col) + j - num_col)] == 'B')
        matrix2[i * num_col + j] = 'B';

    //se nessun vicino è burned allora con probabilità prob_burn può diventare burned
    int rand_num = 1 + (rand() % 100);
    if(matrix2[i * num_col + j] == 'B' || rand_num < prob_burn){ //verifico anche se negli if precedenti non l'ho già bruciato, altrimenti poi se non viene soddisfatto il secondo valore dell'OR inserirei un albero in una cella dove uno stava bruciando
        matrix2[i * num_col + j] = 'B';
    } else{
        matrix2[i * num_col + j] = 'T';
    }
}
```

Questa funzione, per la cella che stiamo considerando, va a verificare lo stato dei vicini a destra, sinistra, sopra e sotto. Se uno di questi sta bruciando, allora la cella diventerà B. Altrimenti se nessuno sta bruciando con probabilità `prob_burn`, l'albero brucerà.

Controllo delle righe ai bordi

Per la computazione delle righe ai bordi vi è bisogno delle righe degli altri processi, quindi vengono utilizzate due `MPI_Wait`, rispettivamente su `request[0]` e `request[1]`, dopodichè si può procedere a controllare la prima e l'ultima riga della sottomatrice di ogni processo. Questo viene fatto utilizzando la funzione `check_borders` che prende in input le righe superiore e inferiore ricevute e va ad eseguire i controlli.

```
void check_borders(char *forest, char *matrix2,char *top_row,char *bottom_row,int
i,int j, int num_row, int num_col,int prob_burn){
    //controllo a destra e a sinistra
    if(((i * num_col)  + j + 1) % num_col != 0 && forest[((i * num_col) + j + 1)]
== 'B')
        matrix2[i * num_col + j] = 'B';
    if(((i * num_col)  + j - 1) % num_col != num_col-1 && forest[((i * num_col) +
j - 1)] == 'B')
        matrix2[i * num_col + j] = 'B';

    //se sto considerando la prima riga, allora faccio il controllo con top_row
    if(i == 0){
        if(top_row[i * num_col + j] == 'B')
            matrix2[i * num_col + j] = 'B';
        if((i * num_col) + j + num_col < num_row * num_col){
            if (forest[((i * num_col) + j + num_col)] == 'B'){
                matrix2[i * num_col + j] = 'B';
            }
        }
            //se abbiamo una sola riga dobbiamo fare il confreonto sia con top
che con bottom
            //ci assicuriamo anche che non siamo alla fine della matrice
        else if((bottom_row[i * num_col + j] == 'B'))
            matrix2[i * num_col + j] = 'B';

    //se sono all'ultima allora controllo con bottom_row
    } else if(i == num_row - 1){
        if(bottom_row[j] == 'B')
            matrix2[i * num_col + j] = 'B';
        //verifico se il vicino della riga superiore sta bruciando
        if(((i * num_col) + j - num_col) >=0 && forest[((i * num_col) + j -
num_col)] == 'B')
            matrix2[i * num_col + j] = 'B';
    }

    //se nessun vicino è burned allora con probabilità prob_burn può diventare
burned
    int rand_num = 1 + (rand() % 100);
    if(matrix2[i * num_col + j] == 'B' || rand_num < prob_burn){ //verifico anche
se negli if precedenti non l'ho già bruciato, altrimenti poi se non viene
soddisfatto il secondo valore dell'OR inserirei un albero in una cella dove uno
stava bruciando
        matrix2[i * num_col + j] = 'B';
    } else{
        matrix2[i * num_col + j] = 'T';
```

```
    }
}
```

La funzione è simile a quella precedente, ed inoltre bisogna considerare anche il caso in cui il processo abbia ricevuto una sola riga, poichè in tal caso, quando leggiamo la riga 0, dobbiamo confrontare i valori nelle cella sia con la riga superiore ricevuta che con quella inferiore.

Terminazione della simulazione del caso la foresta sia vuota

Per terminare la simulazione nel caso la foresta sia vuota, ogni processo incrementa un contatore di celle vuote, che azzera poi ad ogni nuova iterazione.

```
MPI_Reduce(&empty_count,&empty_recv_count,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
        if(myrank == 0){
            if(empty_recv_count == m*n){
                all_empty = 1;
            }
        }

        MPI_Bcast(&all_empty,1,MPI_INT,0,MPI_COMM_WORLD);
        if(all_empty == 1){
            if(myrank == 0)
                printf("Foresta vuota, numero di iterazioni: %d\n",i);
            break;
        }
```

Si è utilizzata la `MPI_Reduce`, in modo che viene fatta la somma dei contatori ricevuti da ogni processo(`empty_count`), se il valore della somma è uguale al numero di elementi totali della matrice, allora viene impostato a $true$ un flag `all_empty` e il processo master farà la broadcast del flag. Ogni processo controllerà il valore del flag e se lo trova ad 1 si arresterà e terminerà la computazione (il processo master mostrerà sul terminale dopo quante iterazioni la foresta è risultata vuota).

Output della simulazione

Al termine degli step o quando la foresta è vuota viene eseguita una `MPI_Gather`, dove il master riceverà tutte le sottomatrici e stamperà sia su terminale che su file la foresta al termine dell'algoritmo.

```
MPI_Gatherv(sub_forest,send_counts[myrank],MPI_CHAR,forest,send_counts,displ,MPI_
CHAR,0,MPI_COMM_WORLD);
```

Correttezza dell'algoritmo

Per facilitare il test della correttezza dell'algoritmo si è realiazzata una versione sequenziale dell'algoritmo e una versione parallela che stampa la foresta ad ogni iterazione, in modo da poter controllare se in ogni step dell'algoritmo le foreste sono sempre uguali. Entrambi i programmi inseriscono su un file lo stato della foresta ad ogni iterazione. Per rendere più veloce il test si è realizzato un piccolo script bash che esegue i due programmi e poi con il comando `diff` verifica se i due file sono uguali.

Esempio per controllare la correttezza su una matrice di dimensione 50x50 con 50 step e un numero di processori pari a 4 (per il programma parallelo). Dalla root directory del progetto eseguire i seguenti comandi.

```
cd test_correttezza/
./check_correctness.sh --row 50 --column 50 --steps 40 --processors 4
#oppure
./check_correctness.sh -r 50 -c 50 -s 40 -p 4
```

L'output dell'esecuzione dei programmi viene riportato nei file `output_sequenziale` e `output_parallelo`. Se i due file contengono lo stesso output, cioè significa che nelle S iterazioni, lo stato della foresta era sempre uguale, fra programma eseguito con p processori e fra il programma sequenziale. Ciò ci assicura la correttezza del programma parallelo.

Benchmarks

Prima di andare nel dettaglio nel mostrare i dati del benchmark effettuato è opportuno fare le seguenti premesse. I benchmark sono stati eseguiti su un cluster di macchine virtuali create su Google Cloud. In particolare, il cluster è stato creato con 4 macchine di tipo e2-standard-8. Questo tipo di macchine hanno si 8vCPU, ma in realtà al sistema operativo solo 4vCPU vengono esposte. Di conseguenza, per avere dati più realistici e sfruttare solamente i core reali disponibili della macchina, il test è stato eseguito con procesessori che variano da 1 fino a 16 (senza usare oversubscribe). Tutti i tempi di esecuzione che saranno mostrati nel seguito, sono la media dei tempi su 10 esecuzioni consecutive della simulazione.

Scalabilità forte

Nel test della scalabilità forte, vengono riportati i tempi dell'esecuzione della simulazione su una matrice di dimensione 3000*3000, eseguendo 100 step e con le seguenti probabilità:

```
    int prob_burn = 50;  // probabilità che un albero si incendi
    int prob_grow = 50;  //probabilità che un albero cresca nella cella vuota
```

Nella tabella di seguito vengono riportati nel dettaglio i dati con anche il relativo speedup al crescere dei processori. Nel calcolo del tempo è stato escluso il tempo impiegato dal MASTER node per inizializzare la matrice all'inizio della simulazione, tale inizializzazione prende 8,3082s (calcolato su 10 esecuzioni).

PROCESSORI	TEMPO (sec.)	SPEEDUP
1	31,3127	1.00000
2	15,9797	1,9595
3	10,8956	2,8738
4	8,55186	3,6615

PROCESSORI	TEMPO (sec.)	SPEEDUP
5	6,85979	4,5646
6	5,84798	5,3544
7	5,10273	6,1364
8	4,65547	6,7260
9	4,19364	7,4667
10	3,82968	8,1763
11	3,58176	8,7422
12	3,32661	9,4127
13	3,14370	9,9604
14	2,98696	10,483
15	3,58176	10,997
16	3,32661	11,708

Scalabilità debole

Nel test per la scalabilità debole invece, viene preso il tempo dell'esecuzione aumentando il numero delle righe e il numero di processori su cui viene eseguita la simulazione. In particolare, il numero di colonne è fissato a 500, mentre le righe sono np*200 (np=numero dei processori) e con un numero di step dell'algoritmo pari a 100.

Scalabilità debole

PROCESSORI	TEMPO (sec.)	EFFICIENZA
1	0,0149	100
2	0,0296	50,3378
3	0,0443	33,6343
4	0,0595	25,0420
5	0,0738	10,1897
6	0,0895	16,6480
7	0,1037	14,3683
8	0,1197	12,4477
9	0,1372	11,2283
10	0,1485	10,0275
11	0,1623	9,18052
12	0,1774	8,39909

PROCESSORI	TEMPO (sec.)	EFFICIENZA
13	0,1913	7,78881
14	0,2059	7,23652
15	0,2216	6,72382
16	0,2366	6,29754

Conclusioni

Come si può evincere dal test della scalabilità forte, l'algoritmo sicuramente riesce a trarre benefici dall'esecuzione in parallelo della simulazione. Da notare però che la diminuzione del tempo è forte soprattuto fino ad arrivare a 8 processori, dopo la diminuzione del tempo tende sempre di più ad appiattirsi. Questo è sicuramente dovuto al fatto che aumentando i processori, va ad aumentare la comunicazione, creando overhead, infatti aumentano sempre di più i processori che devono eseguire l'operazione di MPI_Reduce e di MPI_Bcast (che sono comunque bloccanti a differenza delle send e receive utilizzate) impattando negativamente sul tempo di esecuzione. Quindi per l'algoritmo in questione, un ottimo compromesso fra tempi e costo da investire in processori, vale sicuramente la pena fermarsi ad 8 processori. Anche il test della scalabilità debole conferma tale ipotesi, infatti come si può vedere dalla tabella, quando si aumentano i processori da 8 in su, il tempo inizia ad aumentare significativamente, mentre fino ad 8 processori l'aumento risulta accettabile.