Programmazione Concorrente, Parallela e su Cloud
Relazione del progetto

1. Presentazione della soluzione proposta

La segregazione razziale è una pratica che consiste nella restrizione dei diritti civili su base razzista. Essa riguarda la separazione delle persone nella vita quotidiana e può ripercuotersi su varie attività che i cittadini di una comunità mista svolgono insieme. Nel 1971, l'economista americano Thomas Schelling creò un modello basato su agenti che suggeriva che il comportamento involontario potrebbe anche contribuire alla segregazione. In particolare, tale modello ha mostrato che alcuni individui potrebbero addirittura giungere in uno stato di auto-segregazione anche quando non hanno alcun desiderio esplicito di farlo.

Lo scopo di questo progetto è quello di realizzare una simulazione del modello di Schelling con l'utilizzo di MPI, una libreria del linguaggio C che permette lo scambio di messaggi tra processi.
In particolare, inizialmente si genererà una griglia dove, in maniera casuale, saranno posizionati degli agenti di due tipi differenti, lasciando la possibilità di avere anche spazi vuoti. Questa griglia, rappresentata da una matrice di dimensione fissata, verrà suddivisa in diverse parti in funzione del numero dei processi partecipanti e ogni parte verrà inviata al corrispettivo processo. Successivamente tutti i partecipanti si scambieranno la prima e l'ultima riga della propria sottomatrice, inviando la prima riga al processo predecessore e l'ultima al successore. Le righe "di confine" saranno utili successivamente ai processi per conoscere lo stato delle celle che confinano con la propria sottomatrice e prendere delle decisioni sugli spostamenti degli agenti basandosi anche su queste. Successivamente inizierà il vero e proprio lavoro che ogni processo dovrà svolgere.

In maniera ciclica ogni processo eseguirà una serie di operazioni allo scopo di risolvere il modello di Schelling. Tali operazioni riguardano la scansione della sottomatrice alla ricerca di celle occupate da agenti insoddisfatti e di spazi vuoti dove posizionare tali agenti. Ricordiamo che un'agente viene ritenuto insoddisfatto se e solo se il numero dei suoi vicini dello stesso tipo, in percentuale rispetto al totale dei vicini, è inferiore ad una soglia prestabilita. Dopo aver scansionato la sottomatrice se non sono stati ritrovati agenti insoddisfatti o, inversamente, se ci sono agenti insoddisfatti ma c'è assenza di celle vuote in cui posizionare questi ultimi il processo invia a tutti gli altri il segnale di voler terminare e la terminazione avviene in maniera definitiva solo se tutti hanno la stessa intenzione. Nel caso non si decida di terminare, ovvero, se qualche processo ha ancora agenti insoddisfatti e celle vuote nella propria sottomatrice, il processo effettua gli scambi spostando ogni agente insoddisfatto in una cella libera della sua sottomatrice. Al termine del ciclo, prima di ripartire con una nuova scansione, effettua lo scambio di righe "di confine" tra i processi con la stessa procedura descritta in precedenza. Per evitare alcuni casi critici in cui non si riesce mai a raggiungere la soluzione è stato aggiunto un limite di cicli massimo effettuabili entro il quale i processi tentano di risolvere il problema.

2. Descrizione dettagliata del progetto

Il codice sorgente del progetto è inserito in un unico file “segregazione.c”. La soluzione si trova principalmente all’interno della funzione main, la quale richiama altre funzioni che ho realizzato e che spiegherò successivamente all’occorrenza. La logica del programma è strutturata in due blocchi logici, come anticipato anche nell’introduzione: il primo riguarda la generazione e distribuzione della matrice e il secondo la verifica della soddisfazione degli agenti e del loro spostamento. Scendiamo di più nel dettaglio sull’implementazione dei due blocchi.

Per la logica di generazione e distribuzione della matrice il processo root (nel nostro caso con rank 0) non dichiara una matrice come blocco unico in quanto ogni processo lavorerà solo sulla propria sottomatrice, comunicando con i vicini per trovare la soluzione globale, e dunque la dichiarazione di una matrice unica avrebbe rappresentato uno spreco di memoria. Dunque, il processo con rank 0 fa uso di un buffer di dimensioni pari a COLONNE (la dimensione di una riga, nonché il numero di colonne della matrice) che conterrà i valori pseudocasuali generati con la funzione rand, inizializzata con un seed differente ad ogni esecuzione in modo tale da generare sempre valori differenti ad ogni avvio. Ogni riga generata verrà inviata direttamente ai rispettivi processi e il buffer sarà riutilizzato per le restanti righe. I valori generati sono: 0 per indicare la cella vuota, 1 per indicare il primo agente e 2 per il secondo. Per tutti i test svolti la funzione rand è stata inizializzata invece con il rank del processo root in modo da generare gli stessi numeri pseudocasuali ad ogni esecuzione.

Ogni processo, come già anticipato, dichiara una sottomatrice AxB, le cui dimensioni vengono ricavate in maniera differente in base all’operazione indicata dalla costante OP. Per l’operazione di test sulla strong scalability (OP “strong”), il numero di righe A della sottomatrice è pari al valore ottenuto dalla divisione (senza resto) tra il numero di righe della matrice iniziale (RIGHE) e il numero di processi, mentre il numero di colonne B è lo stesso della matrice iniziale (COLONNE). Precisiamo che per questa operazione, qualora il numero di righe della matrice non sia divisibile perfettamente per il numero di processi ma si generi un resto n, i primi n processi (a partire dallo 0) dichiareranno una sottomatrice con una riga in più (A+1). Per l’operazione di test sulla weak scalability del programma (OP “weak”) il numero di righe A della sottomatrice è pari al valore della costante RIGHE, che in questa modalità assume significato diverso in quanto rappresenta il numero di righe da generare per ogni processo e non quelle dell’intera matrice, calcolate moltiplicando questa costante per il numero di processi. Il valore di colonne B della sottomatrice resta indicato come prima dalla costante COLONNE.

Ritornando alla generazione della matrice, per evitare l’aggiunta di ulteriori controlli ad ogni valore, la generazione delle prime righe che spettano al processo root viene effettuata per prima e le righe ottenute vengono salvate direttamente nella sua sottomatrice. Per le righe restanti, ogni qualvolta si completa la generazione di tutti i valori della riga, questa si invia direttamente al processo a cui spetta, tenendo anche conto di chi ne dovrà ricevere A e chi A+1. La strategia di invio riga per riga non è casuale in quanto inizialmente l’intento era di unire le diverse righe da inviare per ridurre il numero di send, utilizzando le funzioni MPI_Pack e MPI_Unpack o creando un nuovo tipo di dato vettoriale che avrebbe incluso al suo interno più righe. Il problema di queste alternative è che riducevano il numero di send ma avevano un prezzo aggiuntivo da pagare. L’utilizzo di Pack ed Unpack avrebbe portato ad un tempo maggiore per le operazioni di copia dei dati delle righe in un buffer unico prima dell’invio e ad un ulteriore tempo per spacchettare i dati arrivati a destinazione siccome si tratta di spedire decine di migliaia di righe. Per l’invio di più righe sfruttando la dichiarazione di un nuovo tipo di dato invece si sarebbe dovuto ricorrere alla definizione di due tipi differenti: uno per le sottomatrici con A righe e uno per le sottomatrici con A+1 righe. Questa seconda opzione, seppur migliore prestazionalmente rispetto a pack ed unpack avrebbe portato ad avere delle sottomatrici unidimensionali (siccome le righe erano salvate in maniera sequenziale su di un unico array) e dunque l’accesso a questo tipo di dato sarebbe stato meno agile rispetto all’accesso con una classica matrice.

Successivamente, tutti i processi tranne quello root ricevono le righe della matrice che gli spettano e le salvano nella propria sottomatrice.
Infine, tutti i processi eseguono la funzione "exchangeRow" che prende in input quattro array sendFirst, recvFirst, sendLast e recvLast, il rank del processo chiamante, il numero di processi e l'array di MPI_Request che servirà successivamente per controllare che i dati siano stati ricevuti correttamente. Precisiamo che sendFirst e sendLast sono rispettivamente la prima e l'ultima riga della sottomatrice da inviare agli altri processi e recvFirst e recvLast sono gli array che conterranno rispettivamente, la prima riga del successore (rank+1) e l'ultima riga del predecessore (rank-1). La funzione esegue l'invio e la ricezione in modalità non bloccanti con la particolarità che il processo 0 non riceve la riga dal suo predecessore e l'ultimo processo non riceve la prima riga dal suo successore.

Il secondo blocco logico del programma è l'aspetto cruciale della soluzione e contiene la logica per la verifica della soddisfazione degli agenti e del loro spostamento che viene eseguita da ogni processo sulla propria sottomatrice e sulle righe ottenute dai vicini. Questa fase è strutturata ciclicamente ed in modo tale da compiere un numero massimo di giri S per evitare di rimanere bloccati a causa di una soluzione impossibile. Ad ogni giro:

- Per prima cosa azzera il contatore degli agenti insoddisfatti che tiene traccia del totale di quelli riscontrati all'interno della sottomatrice ad ogni giro. Successivamente controlla di aver ricevuto i dati degli scambi delle righe di confine effettuati nella fase iniziale di distribuzione e alla fine di ogni giro. Notiamo che il processo 0 non controlla di aver ricevuto dal suo predecessore e l'ultimo processo non controlla di aver ricevuto dal suo successore poiché il problema non richiedeva una matrice circolare.
- Avvia la procedura di controllo dei conflitti dove: Il processo scorre la sottomatrice tenendo traccia delle celle vuote tramite una coda circolare di puntatori e delle celle insoddisfatte con uno stack di puntatori. Precisiamo che le due strutture dati utilizzate sono volutamente diverse in quanto se si utilizzasse per entrambi la stessa tipologia di struttura dati si sarebbe potuto incorrere in situazioni cicliche di scambio in cui una cella insoddisfatta si spostava sempre tra gli stessi due spazi vuoti. Inoltre, queste strutture dati hanno una dimensione pari a quella della sottomatrice assegnata al processo in quanto bisogna tener conto dei due casi peggiori in cui la sottomatrice ha tutte celle vuote oppure tutte le celle sono occupate da agenti insoddisfatti.
  Il controllo dei conflitti è diviso in tre blocchi principali: Il primo effettua i controlli sulla prima riga della sottomatrice e viene eseguito per ogni tipologia di questa, il secondo su tutte le righe centrali qualora ce ne siano almeno tre, il terzo sull'ultima riga qualora ce ne siano almeno due. Questa divisione serve ad ottimizzare i tempi in quanto elimina diversi controlli per la riga di appartenenza di ogni cella esaminata. La suddivisione è anche logica siccome la prima riga della sottomatrice viene confrontata anche con l'ultima riga del predecessore (ad eccetto del processo 0, la cui sottomatrice non ha un predecessore), l'ultima riga della sottomatrice viene confrontata anche con la prima riga del successore (ad eccetto dell'ultimo processo, la cui sottomatrice non confina con nessun successore) ed infine le righe centrali non effettuano confronti con le righe confinanti.
  Le celle insoddisfatte sono quelle che presentano una percentuale di conflitti maggiore rispetto alla soglia impostata. Nel programma viene utilizzata la costante SOGLIA per indicare la percentuale minima di vicini dello stesso tipo che l'agente deve avere per non generare un conflitto (che lo porterà a volersi spostare). La percentuale di soglia dei conflitti utilizzata è dunque l'inverso della SOGLIA indicata (100-SOGLIA).
  Descriviamo ancora più nel dettaglio la procedura di controllo dei conflitti:
  - La procedura conteggia con la variabile "numVicini" il numero di celle vicine (non vuote) che circondano la cella attuale e verifica se presentano un valore differente da quest'ultima. In tal caso si tiene traccia del numero di differenze trovate per quella cella con un contatore "conflitti".
  - Dopo aver controllato tutte le celle confinanti si calcola la percentuale dei conflitti generati rispetto al numero di vicini posseduti. Se la percentuale di conflitti è maggiore della soglia

indicata si fa puntare la cella insoddisfatta dal puntatore nello stack tramite la funzione “push” e si incrementa il contatore di celle insoddisfatte della sottomatrice.
    - Dopo la scansione di una cella si riportano a 0 i contatori dei conflitti e dei vicini, riutilizzati per la prossima cella non vuota da analizzare.
- Dopo aver controllato tutta la sottomatrice, se non sono stati trovati agenti insoddisfatti, se la sottomatrice è vuota (ha tutti 0) oppure non ha spazi vuoti (si necessita di almeno uno spazio per effettuare gli scambi) esegui la funzione MPI_Allreduce inviando 1, altrimenti invia 0.
    - Se la somma calcolata a partire dai valori inviati da ogni processo è pari al numero di task allora tutti i processi hanno intenzione di terminare e dunque terminano, altrimenti proseguono.
    - La terminazione avviene per tutti i processi insieme altrimenti si rischierebbe che un processo ancora in attività potrebbe invalidare “a cascata” le sottomatrici dei processi vicini che hanno già terminato in qualche giro precedente
- Se ci sono agenti insoddisfatti e spazi vuoti nella sottomatrice avvia la procedura di spostamento degli agenti. Questa procedura viene eseguita ciclicamente fino a quando tutte le celle indicate come insoddisfatte all’interno del giro attuale non vengono spostate in una cella vuota. Per prima cosa si chiama la funzione “pop” per lo stack dei puntatori alle celle irregolari e “popQueue” per la coda degli spazi vuoti, funzioni che eliminano il primo carattere dalla struttura dati e lo restituiscono. Successivamente si sostituisce il contenuto della cella vuota con quello della cella irregolare e si riporta a 0 il contenuto della vecchia cella irregolare (spostando di fatti l’agente insoddisfatto). Infine, si completa l’operazione chiamando la funzione “pushQueue” per aggiungere alla coda il nuovo spazio libero creato in una posizione diversa.
- Dopo aver effettuato tutti gli spostamenti ogni processo chiama la funzione “exchangeRow”, mostrata in precedenza, per effettuare lo scambio delle righe di confine.
- L’incremento del numero di giri viene fatto a fine giro da tutti i processi in contemporanea, questo perché i processi potrebbero procedere a velocità differenti, in base al numero di scambi effettuati, di controlli da svolgere ad ogni giro o alla dimensione della sottomatrice e potrebbe accadere che un processo termini i propri giri totali prima di altri, i quali potrebbero continuare a modificare la loro sottomatrice rendendo scorretta la soluzione fornita dai confinanti.

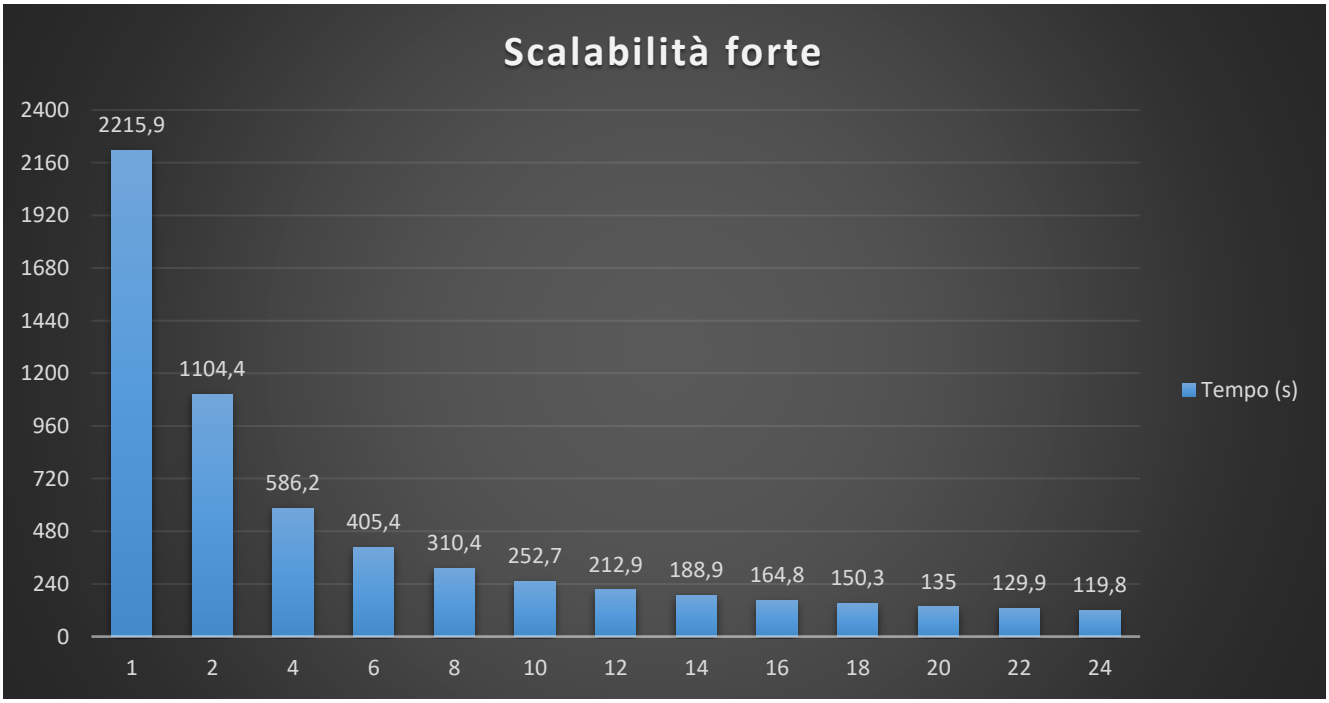
Quando tutti i processi avranno raggiunto un numero massimo di giri S, definito da una costante, o avranno deciso di terminare insieme il programma terminerà. In fase di terminazione ogni processo tiene conto dell’esito delle sue operazioni, se queste hanno avuto successo (dopo l’ultima scansione il numero delle celle insoddisfatte nella propria sottomatrice è pari a zero) imposta a 1 una variabile locale che indica questo stato altrimenti la lascia a 0. Successivamente tutti i processi chiamano la funzione MPI_Reduce inviando il valore dello stato di terminazione locale. La reduce consente al processo root di comprendere se la soluzione ottenuta da ogni processo è ottima globalmente o meno e stampa l’esito.

3. Test effettuati

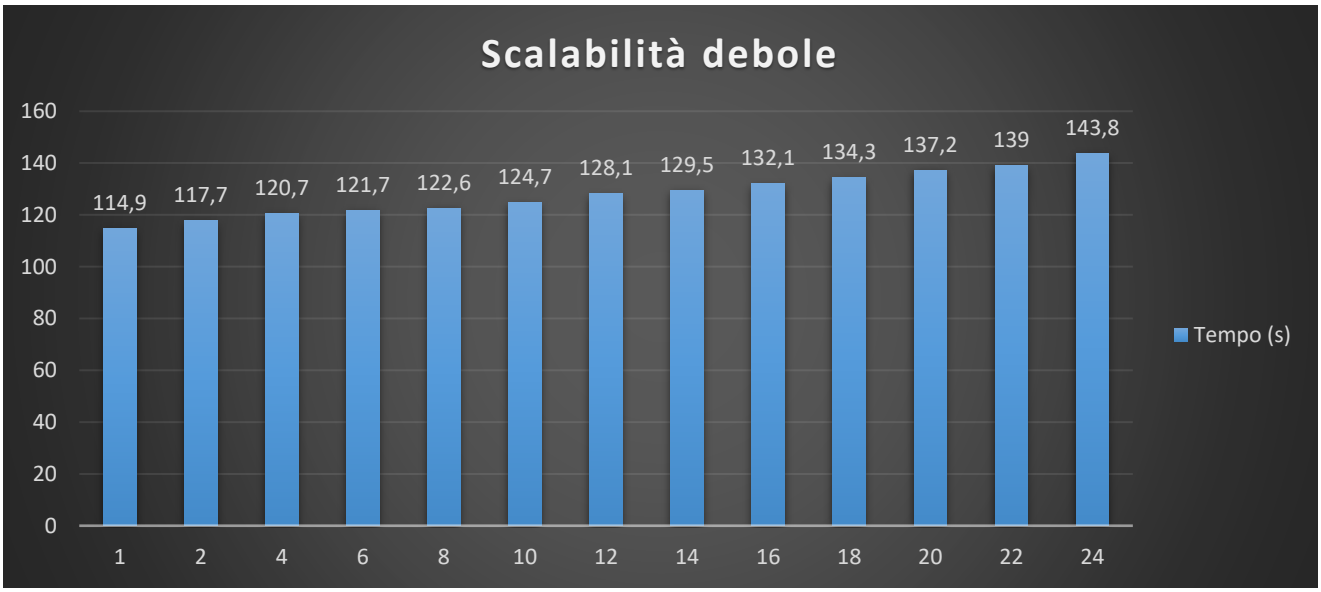
Il primo test è stato svolto in locale con l'utilizzo di Docker e ha avuto come scopo quello di provare la correttezza del programma. Questa è stata provata attraverso diverse esecuzioni con un numero crescente di processi, con la matrice di dimensione fissa e valori invariati. Le diverse esecuzioni hanno avuto sempre lo stesso esito positivo, trovando la soluzione al problema sulla matrice data, impiegando un numero differente di giri a seconda del numero di processi adoperati.

I due test successivi sulla scalabilità forte e debole del programma sono stati effettuati su di un cluster AWS composto da tre nodi di tipo t2.2xlarge; ognuno dei quali con 8 processori e 32 gigabyte di RAM, per un totale di 24 processori e 96 gigabyte di RAM. Entrambi i test sono stati svolti con un numero crescente di processi, rispettivamente: 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24. Anche in questo caso la matrice aveva valori invariati ad ogni esecuzione.
Le istanze utilizzate sono tre in quanto l'account Students di AWS non consentiva l'utilizzo di più istanze in contemporanea rispetto a quelle utilizzate siccome inizialmente l'intenzione era di utilizzarne quattro. Configurando tutte le macchine nella stessa maniera, nel momento in cui si eseguiva il comando mpirun con quattro nodi e qualsiasi valore di processi da 1 a 32 il comando indicava un errore di autenticazione di un nodo sempre differente diverso dal root (nonostante utilizzassi una connessione SSH senza autenticazione) e il comando non terminava mai la sua esecuzione se non in maniera forzata. Eliminando l'IP di una qualsiasi istanza diversa dalla root dall'hostfile, il comando mpirun riprendeva a funzionare normalmente e ad eseguire fino a 24 processi, limite imposto dagli slot indicati nel file.

Per i test sulla scalabilità forte è stata utilizzata una matrice con dimensione fissa, con un numero di righe e colonne pari a ventimila unità. Il numero di giri massimo del programma è stato invece fissato a cento e tutte le esecuzioni considerate sono quelle che hanno effettuato tutti i giri limite. Mostriamo il diagramma con i tempi (in secondi) registrati per ogni esecuzione:



Per i test sulla scalabilità debole la dimensione della matrice variava in funzione dei processi, in particolare per ogni esecuzione sono state assegnate mille righe da ventimila valori per ogni processo. Anche in questo caso il numero di iterazioni è stato fissato a cento e sono stati registrati i tempi delle esecuzioni che hanno svolto tutti i giri. Mostriamo il diagramma dei tempi risultante:



Conclusioni

In conclusione, dopo aver mostrato i diagrammi sulle tempistiche dei test su strong e weak scalability possiamo affermare che:

Il tempo per la strong scalability risulta direttamente proporzionale alla dimensione della sottomatrice assegnata ad ogni processo, legata al numero dei processi coinvolti. Inoltre, bisogna tener conto dell'aumento del tempo dovuto allo scambio delle righe di confine che crescono proporzionalmente con il numero di processi utilizzati. Un altro fattore che influisce sui tempi e anch'esso legato al numero di processi utilizzati è la generazione dei conflitti tra sottomatrici vicine in quanto queste vengono modificate da due processi differenti. Questi conflitti, per ovvie ragioni, non si verificano con l'esecuzione con un singolo processo e dunque bisogna tener conto anche di questa sottile differenza. In generale, i tempi registrati per le diverse esecuzioni sono soddisfacenti in quanto già con l'introduzione del parallelismo notiamo un miglioramento del 50%, miglioramento che arriva fino al 95,6% con l'utilizzo di 24 processi.

I tempi per la scalabilità debole, a differenza di quelli per la scalabilità forte, tendono ad essere molto simili in quanto la dimensione totale della matrice varia in base al numero dei processi e ognuno di questi ha sempre la stessa dimensione del problema. La differenza sostanziale dei tempi riscontrati tra le varie esecuzioni risiede dunque nella generazione e l'invio delle righe per ogni processo, che aumentano in maniera proporzionale a questi. I tempi registrati differiscono di circa ventinove secondi tra l'esecuzione ad un processo, che ricordiamo non comporta nemmeno gli scambi di righe di confine e la generazione di conflitti con queste, e quella con ventiquattro processi risultata la più lunga con un peggioramento del 20%.