# PARALLELIZZAZIONE DI UN AUTOMA CELLULARE – “Water Simulation” SIMULAZIONE DELLA FLUIDODINAMICA PERTINENTE ALL’ACQUA

## *Richiesta (Originale):*

The project consists of the parallel implementation in MPI (and OpenMP) of a Cellular Automaton or other structured grid numerical models (es: Predator - Prey, Heat Equation, Sandpile model, Wator, Game of Life, etc. ) according to the suggestions given in class. Moreover, it is required to write a report on the used methodology, performance tests, etc. Remember that it is a good idea to implement a first (sequential) version in C.

The parallelization consists to split the matrix in "slices", allocating each slice on a process. This requires the bordering of each subarray, so that the extreme portions of each submatrix have the data which is physically located on adjacent processes. In this regard, recall that a division into horizontal slices allows to send the borders (rows) of submatrices, simply send and receive, while a vertical subdivision requires sending of columns (thus, use of MPI derived datatypes). This applies to a static allocation: pay attention to a possible dynamic allocation (hard though!), where the borders are not contiguous. In addition, you can also use send / receive non-blocking for the sending of the borders, the update of the "inside" portion of the submatrix, and once "finished" the receiving of the borders (with MPI_Wait), updating of the border portion of the sub-matrix ( see Models - lesson examples ) .

Tip: It's a standard method to allocate (even statically) the whole array for each process. The MASTER process will ONLY send the data portion of each process (start-end of the sub-matrix ) . See the example mpi_heat2D.c of the MPI tutorial! OSS: Although you can implement the project only with blocking sends and receives, solutions will be positively evaluated that involve the use of "advanced" MPI features, such as virtual topologies, derived data types, non-blocking send-receives, etc. In addition, the use of dynamic allocation for the local sub-matrices will be assessed "positively" for the final evaluation.

Finally, the adoption of a suitable display interface (with Allegro, OpenGL, QT, etc ) will be further evaluated in a positive way.

## *Scelta dell’automa cellulare e libreria grafica:*

Il progetto di automa cellulare in MPI, inerente al corso di Algoritmi Paralleli e Sistemi Distribuiti, consiste nella **simulazione due dimensionale dell’acqua seguendo una semplificazione dei principi della fluidodinamica.** Naturalmente, simulare perfettamente la fluidodinamica comporta innumerevoli considerazioni che renderebbero il progetto incredibilmente complesso. Per tali motivazioni, nasce l’ausilio della programmazione parallela mediante automa cellulare per ridurre notevolmente complessità computazionali e velocizzare la simulazione desiderata.

Per quanto concerne *l’output* grafico, ho scelto la libreria ***Allegro5*** e implementazione source code mediante *IDE Visual Studio.*

## *Che cos'è un automa cellulare?*

**Un automa cellulare** è un modello matematico usato per descrivere l'evoluzione di sistemi complessi discreti (numerabili), studiati in ambito computazionale.

Un automa cellulare nasce mediante una **griglia costituita da celle.**

- **La griglia** può avere una qualunque dimensione finita.
- **Le celle** possono assumere un insieme finito di stati, ad esempio "vivo", "morto", "preda", "predatore" …ecc.

Per ogni cella è necessario definire *l'insieme delle celle "vicine"*, il che varia in base alla progettazione desiderata.

Ad un certo tempo t=0, si assegna ad ogni cella uno **stato**. L'insieme di questi stati costituisce lo **stato iniziale** dell'automa cellulare. Dopo un tempo prefissato, ogni cella sarà soggetta all'applicazione delle **regole dell'automa cellulare** conseguendo l'eventuale **cambiamento di stato.**

L'idea fu originariamente sviluppata da **Stanislaw Ulam** e da **John von Neumann** nei primi anni Cinquanta ed è fiorita con lo sviluppo delle teorie di computazione e le strutture hardware. Un classico esempio di automa cellulare è **il gioco della vita** ideato dal matematico inglese **John Conway.**

## *Che cos'è MPI?*

*Message Passing Interface* (**MPI**) è un protocollo di comunicazione per computer **a memoria distribuita**. È di fatto lo standard per la comunicazione tra nodi appartenenti ad un cluster di computer che eseguono un programma parallelo. MPI, rispetto ad altre librerie utilizzate per il calcolo parallelo, ha il vantaggio di essere molto portatile (implementata per tantissime architetture parallle) e veloce (ottimizzato per ogni architettura). Il progetto è stato svolto mediante OpenMPI.

**OpenMPI** è una implementazione di MPI che combina varie tecnologie e risorse per costituire la migliore libreria MPI disponibile.

## *Che cos'è Allegro5?*

**Allegro** è una libreria open source per la creazione di videogiochi. Sviluppata in **C**, fornisce delle funzioni per la gestione della grafica 2D, manipolazione delle immagini, stampa di testo a schermo, riproduzione audio, lettura degli input e timers. Il nome è un acronimo ricorsivo di ***A**llegro **L**ow **L**evel **G**ame **RO**utines.*

**Allegro5** rappresenta l'ultima versione della libreria Allegro. Si tratta di una completa riprogettazione sia della parte API che delle operazioni interne grafiche. In particolare, Allegro5 gode di scelte implementative tali da renderla multithread-safe.

## *Specifiche implementative*

Prima di approfondire le specifiche effettuate, vorrei dare una visualizzazione, sottoforma di *.gif*, del ***risultato seriale finale,*** così da avere una migliore comprensione della versione parallela, mediante l'ausilio di MPI.
Animazione disponibile al seguente link:
*https://drive.google.com/file/d/1s37SHXdRERXDSUV43r5xExmNsSS3rCmq/view?usp=sharing*

1. ***Variabili globali***

```
const float MinMass = 0.0001;
const float MaxMass = 0.5;
const float MaxCompress = 2;
const float MaxSpeed = 30;
const float MinFlow = 0.01;


const float MinDraw = 0.01; //For water colors based on mass
const float MaxDraw = 1.1;

const int map_dim = 40; //40 poiché 40 è divisibile per 2, 4 e 8. 3 Comuni cardinalità di microprocessori per CPU
//const int map_height = 50;

const int AIR = 0;
const int GROUND = 1;
const int WATER = 2;

//Variabili per il parallelismo MPI
const int root = 0;
int numOfProcesses;

//Allegro settings
const int display_dim = 880;
ALLEGRO_DISPLAY* display;

//Coherence between the matrixes and the allegro display
const int blockNumberPerSide = map_dim;
const int spacing = display_dim / blockNumberPerSide;
```

- ***Stati possibili:***
    1) *AIR* = Cella nera, letteralmente "*aria",* permetterà all'acqua il libero passaggio.
    2) *GROUND* = Cella gialla, rappresenta un "muro" dove l'acqua non può attraversare. Mediante più celle GROUND è possibile creare contenitori, deviatori…ecc.
    3) *WATER* = Cella con "acqua", colorazione dinamica in base alla massa di acqua della cella in questione. (*Vedere sezione dedicata alla colorazione*)

- ***MinMass*** e **MaxMass** sono due parametri necessari per la scelta implementativa della fluidodinamica. Infatti, per poter simulare nel modo più verosimile possibile il comportamento dell'acqua, è necessario adottare un approccio *Pressione-Massa*. Mediante la conservazione della massa dell'acqua per ogni cella, è possibile controllare il flusso e stabilire quanta massa di acqua può supportare una cella (*MaxMass),* avendo come massa minima *MinMass* per esser una cella nello stato WATER.
***MaxCompress*** indica quanta massa aggiuntiva al *MaxMass* una cella possa contenere. È utile per simulare la distribuzione più appiattita dell'acqua.

- ***MaxSpeed*** rappresenta la velocità massima in cui l'acqua possa muoversi.

- ***MinFlow*** specifica il flusso minimo di trasferimento di massa necessario.

- ***MinDraw*** e ***MaxDraw*** delimitano l'intervallo di colorazione delle celle *WATER (**vedere sezione 3. Metodo waterColor**).*

- ***Variabili globali per il parallelismo MPI:*** Processo *root* sarà il processo con *rank* 0 mentre *numOfProcesses* conterrà il numero di processori utilizzati in fase di *MPIExecution.*

- ***map_dim:*** Indica la dimensione della matrice logica. Ho come dimensione 40x40 poiché per dimensioni troppo elevate, la fluidità dell'acqua verrebbe compromessa (per questioni legate al mio hardware). Inoltre, 40 è una dimensione multiplo del numero di *core* più comuni in circolazione, utile per la fase di partizionamento dei dati.

- ***Impostazioni display Allegro:*** Come dimensioni del display per l'output grafico, ho scelto *880x880.* ***blockNumberPerSide*** rappresenta semplicemente il numero di celle per righe che non è altro che la *map_dim,* ***spacing*** rappresenta invece la dimensione di ogni cella della matrice logica sul display Allegro.

  $$spacing = \frac{display\_dim}{blockNumberPerSide} = \frac{880}{40} = 22$$

  Dando vita a celle di dimensione *22px x 22px sull'output grafico.*

## *2. Scelte strutturali*

La simulazione voluta si basa sulla relazione tra massa e pressione. Grazie ad esse, infatti, è possibile replicare la capacità dell'acqua di assumere la forma del recipiente e di riempire quest'ultimo. A tale scopo, **sono necessarie tre matrici**:

1) Matrice ***blocks,*** per la memorizzazione dello stato delle celle dove $Stato \in \{WATER, GROUND, AIR\}$.

2) Matrice ***mass,*** contiene, in corrispondenza parallela ad ogni cella *WATER,* la massa d'acqua per tale cella. A tale scopo, mediante trasferimento di massa (o *Flusso)* è possibile simulare il movimento dell'acqua.

3) Matrice ***new_mass***, permette di implementare lo scambio di matrice con *mass* per le evoluzioni di stato. Memorizzerà lo stato successivo all'attuale per ogni cella applicando le ***regole dinamiche del flusso** (vedere sezione 5. **Regole di Waterflow**).* Viene inizializzato pari alla matrice mass.

## *3. Funzioni*

- ***matrixAllocation***
  Alloca un array due dimensionale in modo contiguo in memoria. L'allocazione dinamica di una matrice in modo ***contiguo in memoria*** è fondamentale nel calcolo parallelo. Basti pensare all'inefficienza della comune allocazione per righe, legata al fatto che ogni riga si trovi in aree di memoria differenti. L'allocazione contigua di un array due dimensionale aumenta notevolmente l'efficienza.

- ***initContainerMap***
  Si tratta di un metodo di costituzione dello ***stato iniziale*** delle celle. Prende come parametri le tre matrici (*vedere sezione dedicata alle scelte strutturali)* e sceglie alcune celle per i *GROUND,* alcune per il flusso continuo d'acqua *WATER (con celle parallele mass inizializzate a MaxMass)* e le restanti per *AIR.*

- ***untilESCisPressed***
  Come posso terminare l'esecuzione della ***Sezione Parallela (Sezione 4****.)*? Ho adottato un approccio di *MPI_Bcast,* ossia un broadcast da parte del *root process* ai restanti processi dell'avvenuta richiesta di terminazione. Nel particolare, il *root* controlla se il tasto "*ESC"* venga premuto, in caso positivo, invia un intero "*buf*" che assume valore pari a *INT_MAX* ai restanti processi. Ogni processo ritornerà un *booleano* con valore pari a false se $buf = INT\_MAX$ terminando il ciclo.

- ***waterColor & map***
  waterColor restituisce la colorazione sottoforma di standard *RGB* per la cella *WATER* in base alla massa per tale cella. Riceve un *float m* che rappresenta la massa ed effettua una mappatura della componente *B = BLUE* dello standard *RGB.* La mappatura, mediante funzione *map,* prende il valore di *massa* e lo mappa da un intervallo $[0.01, 1]$ ad un intervallo $[255, 200]$ per la componente *BLU* e ad un intervallo $[240, 50]$ per la componente *RED (nel caso di massa < 1).* Risultato finale consiste in celle tendenti al *BIANCO* per $massa \to MinMass$ e celle tendenti al *BLU SCURO per* $massa \to MaxMass.$ Denotando, dunque, una colorazione blu scura laddove vi è maggiore concentrazione di massa d'acqua.

- ***print***
  Metodo di stampa sul display *Allegro*. Semplicemente riceve le due matrici *blocks* e *mass* e controlla lo stato di ogni cella:
  1) *WATER,* stampa la cella d'acqua invocando ***waterColor*** per ottenere la colorazione opportuna in base alla massa della cella.
  2) *GROUND,* stampa la cella *gialla* (*RGB = (245, 245, 60)).*
  3) *AIR,* stampa la cella *nera (RGB = (0, 0, 0)).*

- ***get_stable_state_b***
  Nasce il problema di come simulare la pressione verticale dell'acqua con un accumulo, ad esempio, d'acqua in un contenitore. Tale metodo permette di calcolare la quantità di massa rimanente nella cella inferiore a causa di una risalita di flusso.

## *4. Sezione parallela*

- ***MPI_Init***
  Permette di inizializzare l'ambiente di esecuzione MPI.

- ***MPI_Comm_rank***
  Determino il rank del processo chiamante nel mio comunicatore, ossia *MPI_COMM_WORLD.* Il rank del processo chiamante viene memorizzato nella variabile *rank.*

- ***MPI_Comm_size***
  Determino il numero di processi nel mio comunicatore *MPI_COMM_WORLD* e lo memorizzo in *numOfProcesses*.

- ***Allocazione iniziale prima della partizione dati***
  Il *root* process alloca dinamicamente in modo contiguo le matrici *blocks, mass e new_mass.* Dopodiché richiama *initContainerMap*, inizializza Allegro, *Allegro_Display e Allegro_Keyboard (per l'input da tastiera),* infine, stampa lo stato iniziale delle celle.

- ***Sottomatrici per ogni processo***
  La scelta della partizione delle matrici base (*blocks, mass e new_mass)* doveva permettere il minore *overhead* possibile, sia in termini di comunicazioni che di efficienza. A tale scopo, scelsi ***il partizionamento a blocchi*** dove ogni processo gode di una propria sottomatrice per ogni matrice base, chiamate rispettivamente *subBlocks, subMass & subNewMass.* Le dimensioni delle sottomatrici dipendono dal numero di processi usufruiti in fase di *MPI_Execution,* pertanto, è intuibile dover dividere le *map_dim* righe di ogni matrice base fra *numOfProcesses* processi. Ogni sottomatrice avrà, dunque, dimensioni pari a $\frac{map\_dim}{numOfProcesses} x\ map\_dim$.

- ***MPI_Scatter***
  Come posso inviare i blocchi delle matrici base ai rispettivi processi? *MPI_Scatter* ci viene d'aiuto, il *root* process partiziona le matrici base in modo equivalente specificando come &*send_buffer* la matrice base da inviare (3 Scatter poiché ho matrici *blocks, mass e new_mass),* e &*send_count* la quantità di dati (celle) da inviare, ossia $\frac{map\_dim}{numOfProcesses} * map\_dim$, e come &*recv_buffer* la sottomatrice corrispondente per ogni processo.

- ***Comunicazione fra bordi delle sottomatrici***
  Ora che le matrici base sono correttamente partizionate fra i *numOfProcesses*, come posso conservare le informazioni delle righe sovrastanti e/o sottostanti ai bordi delle sottomatrici? Avere informazioni sui bordi è vitale per la corretta implementazione delle *regole di Waterflow. Ad esempio,* immaginiamo di volere controllare il vicino superiore di una cella sulla prima riga della sottomatrice *subBlocks* del processo *1. subBlocks* NON contiene informazioni pertinenti alla riga sovrastante, allora nasce la necessità di conservare, per ogni processo, riga sovrastante

e sottostante per i processi NON ai bordi superiore e inferiore delle matrici base, mentre per il processo *0,* conservare la riga sottostante all'ultima riga delle proprie sottomatrici e, per il processo *numOfProcesses – 1*, conservare la riga sovrastante alla prima riga delle proprie sottomatrici.

A tale scopo, alloco:

- ***Due vector*** per i bordi inferiore/superiore delle *subMass.*
- ***Due vector*** per i bordi inferiore/superiore delle subBlocks.

- ***Scambio bordi***

  Ogni processo deve inviare ai processi adiacenti i propri bordi per le questioni di coerenza dei dati specificate nel punto precedente. Usufruisco di una *MPI_ISend* che mi permette di inviare il rispettivo bordo mediante comunicazione *non-bloccante*, utile per fattori di efficienza, ridotto overhead e possibile grazie al fatto che tale bordo non debba essere riutilizzato dal mittente. Tra i parametri del *MPI_ISend abbiamo*

  - ****buf:*** Specifica quale riga (bordo) sto inviando
  - ***Count:*** ossia la quantità di dati che viene inviato (*map_dim)*
  - ***Dest:*** rank del destinatario. Nel caso di tutti i processi tranne *numOfProcesses – 1,* invieremo il bordo inferiore al processo successivo. Nel caso di tutti i processi tranne *root,* invieremo il bordo superiore al processo precedente.
  - ***Tag:*** Mi permette di associare un tag alla comunicazione per correlare *l'invio* con la *ricezione.*

  Per la ricezione invece, usufruisco di *MPI_Recv* che aggiorna la propria variabile *s* di stato (*MPI_Status)* quando riceve correttamente la riga. Nel caso invece dei processi *root* e *numOfProcesses – 1,* bordo superiore (*per root)* viene riempito di zeri e bordo inferiore (*per numOfProcesses – 1)* viene riempito di zeri.

### *5. Regole di Waterflow*

Ad ogni iterazione del *while,* oltre all'invio dei corrispettivi bordi, abbiamo l'evoluzione di stato delle celle. In particolare, ogni processo controlla se la cella *subBlocks*(*x,y)* sia una cella *WATER* e memorizza all'interno della variabile *remaining_mass* la massa della cella *subMass*(*x,y).*

Dopodiché applico in successione le seguenti *regole di waterflow:*

1) $\boldsymbol{Se\ la\ cella\ inferiore\ \neq GROUND}$, allora: ottengo la massa che confluirà nella cella inferiore mediante il metodo *get_stable_state_b (calcolato sulla massa rimanente sommata alla massa della cella inferiore)* e lo memorizzo in *Flow.* Moltiplico *Flow per 0.5,* ciò rende il trasferimento di flusso più fluido. Vincolo *Flow* ad essere $0 \leq Flow \leq \min(MaxSpeed, remaining\_mass)$ e aggiungo alla *subNewMass* della cella inferiore il *Flow* calcolato, sottraendolo dalla subNewMass della cella corrente. Aggiornamenti di massa sono memorizzate in *subNewMass*. Continuo alla condizione successiva se $remaining\_mass > 0$.

2) $\boldsymbol{Se\ la\ cella\ a\ sinistra\ \neq GROUND}$, allora:
$Flow = \frac{subMass(x,y) - subMass(x,y-1)}{4}$, ossia *Flow* assume un quarto il valore ottenuto dalla differenza tra massa corrente e massa a sinistra. Moltiplico *Flow per 0.5,* ciò rende il trasferimento di flusso più fluido. Vincolo *Flow* ad essere $0 \leq Flow \leq remaining\_mass$ ed effettuo il trasferimento di massa tra le due celle. Aggiornamenti di massa sono memorizzate in *subNewMass*. Continuo alla condizione successiva se $remaining\_mass > 0$.

3) $\boldsymbol{Se\ la\ cella\ a\ destra\ \neq GROUND}$, effettuo il medesimo procedimento rispetto alla condizione precedente ma in questo caso con la cella adiacente a destra. Continuo alla condizione successiva se $remaining\_mass > 0$.

4) *Infine,* $\boldsymbol{Se\ la\ cella\ superiore\ \neq GROUND}$, allora: ottengo la massa che confluirà nella cella superiore mediante *remaining_mass – get_stable_state(calcolato su remaining_mass + mass della cella superiore). Vincolo Flow ad essere* $0 \leq Flow \leq (MaxSpeed, remaining\_mass)$ ed effettuo il trasferimento di *Flow* tra cella attuale e cella superiore. Aggiornamenti di massa sono memorizzate in *subNewMass*.

A questo punto avrò, per ogni processo, *subNewMass* con i nuovi valori di massa per le celle con *WATER*. Ora effettuo lo scambio tra *subNewMass e subMass* così da aggiornare quest'ultimo con l'evoluzione ottenuta. Infine, ad ogni cella con *subMass* maggiore di *MinMass,* assegno lo stato di *WATER* alle rispettive *subBlocks.*

6. ***Terminazione sezione parallela e stampa***
   Usufruisco di ***MPI_Barrier*** sul mio comunicatore *MPI_COMM_WORLD* per attendere il completamente di tutti i processi. Dopodiché effettuo due *MPI_Gather*[(1)] da parte del *root:*
   1) Richiamo i partizionamenti *subBlocks* "*raccogliendoli"* nella matrice base originale *blocks.*
   2) Richiamo i partizionamenti *subMass* nella matrice base originale *mass.*

   Infine, il *root* richiama la funzione *print*, passando le matrici *blocks* e *mass.*

   Terminata l'esecuzione del *while loop* mediante tasto *"ESC",* dealloco le strutture dati e termino l'ambiente d'esecuzione MPI con *MPI_Finalize.*

Risultato finale sottoforma di *.gif:*
*https://drive.google.com/file/d/1GhueJ-T6iZuh_dLJm0JzN73O7ZO_DQL6/view?usp=sharing*

*(1) MPI_Gather è una comunicazione many-to-one e ha il funzionamento opposto dello Scatter. Effettua un raggruppamento dei risultati dei processi, dove il root li raccoglie per un risultato finale.*

## Risultati – *SpeedUp & Efficienza*

Per il calcolo dello ***Speedup***, ossia $\frac{tempo\ seriale}{tempo\ parallelo}$, ho catturato il ***tempo medio***[(2)] necessario per una evoluzione di stato (*dimensione delle matrici pari a 40x40*):

- ***Tempo medio seriale:*** 6351 *millisecondi*.
- ***Tempo medio parallelo:*** 1964 *millisecondi*, con 4 processori. 3693 *millisecondi* con 2 processori,

$$\boldsymbol{Speedup\ (4\ processori)} = \frac{0.006351s}{0.001764s} = \ 3.6003$$

$$\boldsymbol{Speedup\ (2\ processori)} = \frac{0.006351s}{0.003293s} = \ 1.9286$$

Possiamo notare un notevole *Speedup* in entrambe le casistiche parallele, denotando un ulteriore miglioramento nel caso in cui vi sono quattro processori. Questo perché con quattro processori diminuiamo la quantità di dati che ogni processore andrà ad esaminare e, nonostante maggiore overhead dovuto alle ulteriori comunicazioni, generiamo tempi di evoluzione minori rispetto alla controparte con due processori.

Altra interessante osservazione è la tendenza dello *speedup* a *NON* essere perfettamente lineare. Questo perché esistono un insieme di fattori di rallentamento nel parallelismo, tra cui *latenza,* dovuto all'accesso in memoria e *comunicazioni* (*Send, Receive, Broadcast…ecc.)*

*(2) Tempo di ogni iterazione MPI ottenuto mediante la differenza fra start_time (prima dello scambio dei bordi) ed end_time nella fase di assegnazione degli stati WATER alle celle rispettive. Tempo ricavato mediante MPI_WTime(). Mentre per la versione seriale, tempi ottenuti mediante la funzione high_resolution_clock() dell'header chrono per ogni iterazione.*

Per il calcolo dell'**efficienza**, ci avvaliamo della formula (3):

$$E = \frac{S}{p} = \frac{Tempo\ Seriale}{numeroProcessori * Tempo\ Parallelo}, allora:$$

$$Efficienza\ (4\ processori) = \frac{0.006351s}{4 * 0.001764s} = 0.900008$$

$$Efficienza\ (2\ processori) = \frac{0.006351s}{2 * 0.003294s} = 0.964025$$

Ricordiamo una *buona efficienza* avviene con $E \to 1$.

Notiamo inoltre, dai corollari dell'Efficienza, che all'aumentare del numero di processori, diminuisce l'efficienza poiché aumenta il *parallel overhead.*

$$parallel\ overhead = T_0 = numeroProcessori * T_p - T_s, \textit{ allora}$$

$$T_0(4) = 4 * 0.001764s - 0.006351s = 0.000705$$

$$T_0(2) = 2 * 0.003294s - 0.006351s = 0.000237, \textit{ pertanto}$$

$$T_0(4) > T_0(2), allora\ E(4) < E(2).$$

E dai risultati precedenti, i dati coincidono.

## *Hardware utilizzato*

*CPU:* Intel Core i5 – 6300HQ @2.30GHz – Quad-core
*RAM:* 8,00GB *DDR4*

## *Sitografia*

***Specifiche MPI -*** *https://www.mpi-forum.org/docs/*

***Physical Modelling for Cellular Automatas -***
*https://tomforsyth1000.github.io/papers/cellular_automata_for_physical_modelling.html*

***Allegro5 Documentation -*** *https://github.com/liballeg/allegro_wiki/wiki*