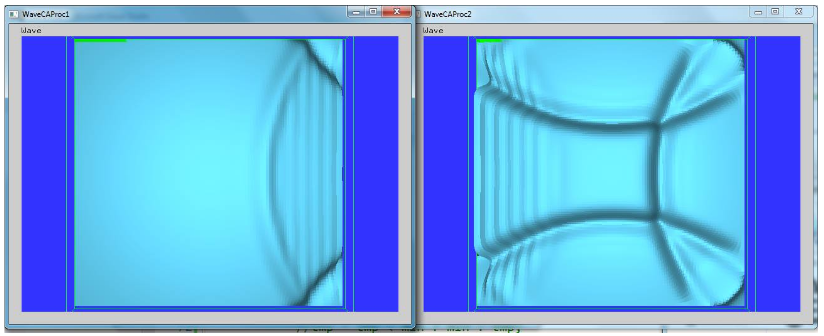
Ring o Star.

# Parallelizzazione di un automa cellulare su cluster MPI

## Progetto di fine corso di Algoritmi paralleli e sistemi distribuiti



|  |  |
| --- | --- |
| Studenti: Aldo Marzullo  Luigi Olivella  Sommario  [Introduzione 2](#_Toc422251347)  [Breve descrizione del paradigma computazionale: 2](#_Toc422251348)  [Implementazione 3](#_Toc422251349)  [La libreria OpenCAL 3](#_Toc422251350)  [La funzione di transizione 4](#_Toc422251351)  [La parallelizzazione e il sistema di comunicazione 5](#_Toc422251352)  [I tipi derivati 6](#_Toc422251353)  [Comunicazioni persistenti 6](#_Toc422251354)  [Implementazione del Cluster 7](#_Toc422251355)  [Elasticità per la gestione dati 8](#_Toc422251356)  [Procedura principale 10](#_Toc422251357)  [Performance 10](#_Toc422251358)  [Considerazioni finali: 13](#_Toc422251359) | Docente: William Spataro |

# Introduzione

Il nostro sistema vuole simulare la propagazione di un’onda a fronte circolare di un fluido generata da una perturbazione iniziale applicata in un qualsiasi punto dello spazio.

L’evoluzione della simulazione viene calcolata in modo parallelo utilizzando un sistema di parallelizzazione distribuito basato sull’utilizzo dello standard MPI.

Lo scopo finale del progetto vuole essere quello di sfruttare il potere di calcolo delle macchine connesse nella rete locale in modo da poter eseguire calcoli più dettagliati nel minor tempo possibile.

Ulteriore feature aggiunta è la possibilità di osservare la simulazione sui display di cui ogni macchina è provvista. Questo dà la possibilità di osservare il fenomeno su uno spazio maggiore, aggirando le limitazioni dovute al potere computazionale della singola macchina.

### Breve descrizione del paradigma computazionale:

La scelta del paradigma da utilizzare ricade sugli Automi Cellulari, che si prestano molto bene alla definizione del problema.

Il modello è stato così definito:

< Z, Q, X, 𝛕, 𝝈, P >

Z è lo spazio in cui l’automa è definito. Formalmente la nostra scelta ricade su uno spazio bidimensionle del tipo:

Q è il prodotto cartesiano di tutti i sottostati che definiamo nell’insieme S.

I sottostati utilizzati sono:

- Q\_old: ovvero il sottostato che rappresenta la situazione dell’automa al tempo

- Q\_old\_2\_steps: Che rappresenta la situazione dell’automa al tempo

P è l’insieme dei parametri globali, che definiscono lo step temporale, l’ intensità della perturbazione iniziale e il punto in cui viene applicata, la dimensione dello spazio.

**X** è il vicinato considerato dalla nostra funzione di transizione, in particolare X è il vicinato di Von Neumann, utilizzato per l’evoluzione degli stati.

La funzione di transizione consiste nel calcolo della quantità di fluido contenuto in una cella al tempo. Essa è definita come la discretizzazione dell’equazione del moto ondoso ed è così definita:

# Implementazione

### La libreria OpenCAL

La libreria OpenCAL è la soluzione scelta per realizzare gli automi cellulari. Infatti tramite l’uso di tale libreria modellare il concetto di Automa Cellulare con i suoi sottostati e funzione di transizione è molto semplice ed immediato. Essa è scritta nel linguaggio di programmazione strutturato C, scelta che è stata fatta per perseguire una migliore performance rispetto al linguaggio orientato agli oggetti C++, ed è stata concepita per essere multipiattaforma essendo quindi possibile utilizzarla sia sotto Windows, Sistemi Linux e Mac OS.

L’automa cellulare è descritto nella classe *waveCA.*

Il nostro automa necessita di due sottostati per la memorizzazione delle informazioni relative alla sua evoluzione. In particolare è necessario che l’automa tenga l’informazione relativa al valore della massa di fluido contenuta in ciascuna cella sia al passo corrente che al passo precedente.

Concretamente i due stati *q\_old* e *q\_old\_2\_steps* sono implementati tramite l’utilizzo di quattro matrici, due per ogni sottostato:

La prima, *curr,* è una matrice di sola lettura. Essa contiene le informazioni relative allo stato corrente dell’automa che verranno utilizzate per il calcolo dei nuovi valori.

La seconda, *next,* è la matrice su cui verranno scritti i nuovi valori calcolati.

Al termine dell’applicazione della funzione di transizione le due matrici vengono scambiate in modo che la matrice in scrittura assuma il significato di *stato corrente* e sia pronta per contenere i nuovi valori al prossimo step temporale.

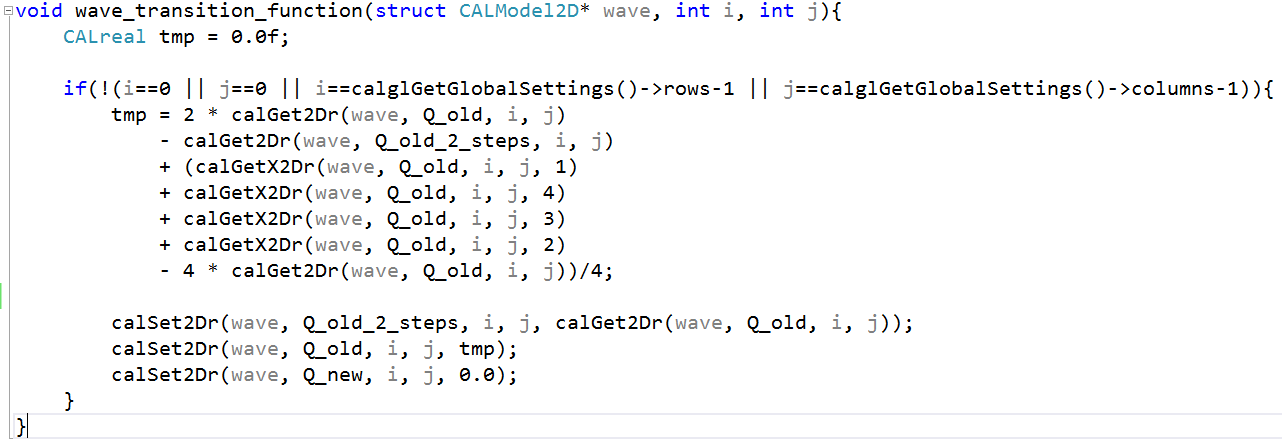
### La funzione di transizione

La funzione di transizione rappresenta il core dell’evoluzione del nostro automa cellulare. Essa deriva dalla combinazione dei due effettivi processi elementari che determinano l’evoluzione dello stato di ogni singola cella sulla base dello stato del vicinato.

Il primo processo elementare determina in effetti la quantità di fluido della cella corrente sulla base di una rappresentazione discreta dell’equazione del moto ondoso, la quale viene riportata di seguito:

Come si osserva, il nuovo stato della cella si evolve in base al suo stato corrente e allo stato dei suoi vicini sia al tempo corrente che allo step di tempo precedente.

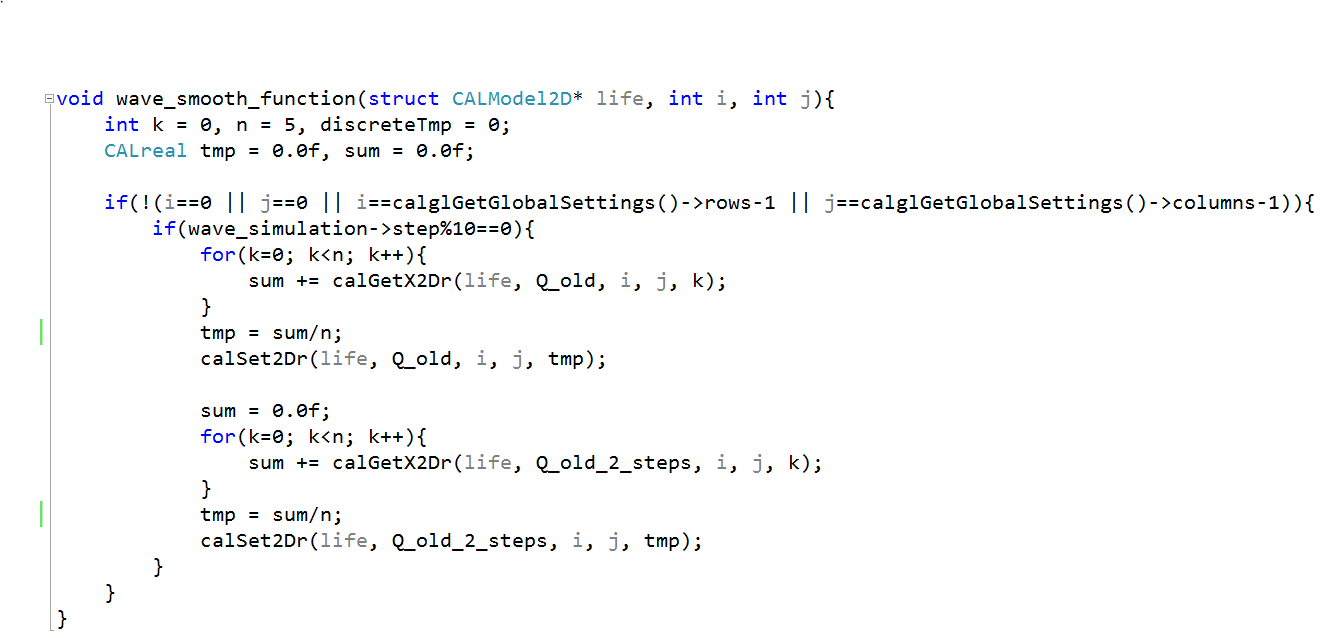
Essendo lo spazio dell’automa definito da un vettore bidimensionale, l’accesso allo stato dei vicini si ottiene in tempo costante per entrambi i sottostati *q\_old* e *q\_old\_2\_steps.* Riportiamo di seguito l’implementazione della funzione:



In particolare, *tmp* sarà la quantità di fluido contenuto nella cella, che verrà copiata tramite l’istruzione *calSet2Dr* nella matrice in scrittura del sottostato *q\_old.*

Il secondo processo elementare, chiamato *funzione di smooth,* permette di rendere più realistica la simulazione “smussando” i valori fra la cella corrente e il suo vicinato in base alla media delle celle che compongono il vicinato stesso. Questo consente in maniera semplice di colmare il gap di approssimazione inevitabilmente dovuto alla rappresentazione discreta di valori altrimenti rappresentati su uno spazio continuo.

Ne riportiamo di seguito l’implementazione:



Si noti, come d’accordo con l’equazione, che i valori di fluido contenuti nella cella sono troncati ad un certa soglia per impedire la crescita incontrastata delle quantità.

# La parallelizzazione e il sistema di comunicazione

Come meglio specificato nella sezione precedente, la simulazione avviene su uno spazio bidimensionale di dimensione NxM.

L’idea alla base della parallelizzazione è quella di ricreare su ogni macchina connessa alla rete locale una matrice rappresentante una porzione di quella originale. Ogni automa si evolverà quindi indipendentemente su ogni nodo secondo le regole definite nella funzione di transizione.

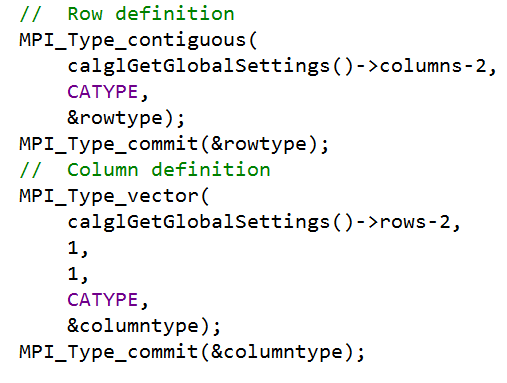
Poiché la simulazione sia coerente, è necessario però fare in modo che ogni automa risulti parte di un’unica grande entità, introducendo una comunicazione fra i nodi tramite l’utilizzo delle funzioni di messaggistica asincrona *I\_Send e I\_Recv* messe a disposizione dalla nostra implementazione di MPI.

In particolare quello che i processi si comunicano sono quelli che vengono comunemente chiamati *bordi fantasma*.

La matrice locale ad ogni processo, è infatti in realtà di dimensione NxM + 2 dove le due colonne e righe sono inizializzate, ad ogni esecuzione della funzione di transizione, ai valori presenti nella colonna corrispondente del processo adiacente (secondo la topologia definita dal file di configurazione del sistema). Ogni automa potrà quindi evolversi indipendentemente tenendo conto però della situazione delle colonne nei punti di giuntura con i processi vicini.

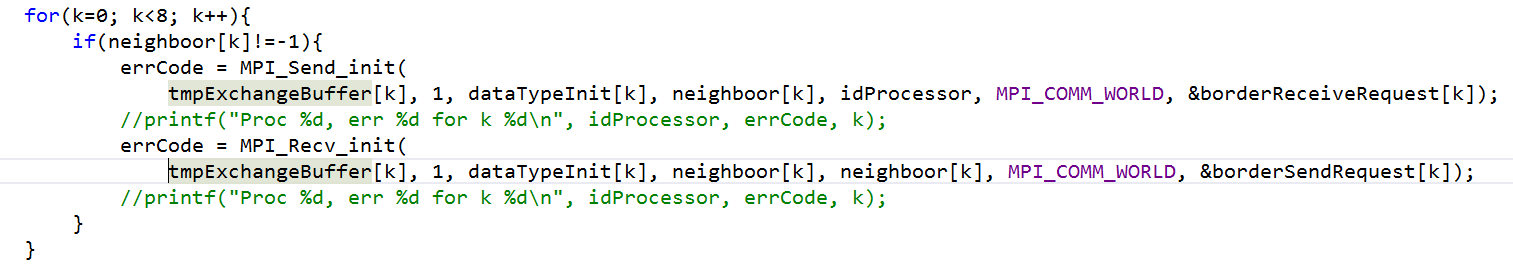
# I tipi derivati

Lo scambio dei bordi è reso più intuitivo tramite la creazione di un apposito tipo derivato definito dall’unione di M celle contigue. Vengono infatti definiti nuovi tipi per lo scambio dei bordi, essi sono il tipo “riga” e il tipo “colonna”.



# Comunicazioni persistenti

Poiché lo scambio dei bordi è un’operazione che si effettua con ripetuta continuità tra gli steps del nostro automa, l’utilizzo delle comunicazioni persistenti ci consente una gestione più efficiente delle risorse non dovendo ogni volta allocare e deallocare risorse per lo scambio dei dati. Qui segue il codice con cui si creano gli oggetti “request” dove saranno poi utilizzati per la fase di scambio dei bordi.



# Implementazione del Cluster

L’automa può essere distribuito su un cluster di macchine connesse in rete. Questo viene fatto tramite l’utilizzo delle funzioni messe a disposizione da MPI, le quali permettono di trattare diverse macchine fisiche connesse sulla stessa rete, come un unica grande macchina capace di sfruttare la potenza di calcolo di tutte le altre.

Il codice infatti può essere scritto pensando solo in termini di processi locali, per poi espandere il range senza dover modificare nulla, aggiungendo solo le apposite opzioni nel comando di avvio.

Nel nostro caso l’obiettivo della parallelizzazione è quello di poter simulare l’automa su una matrice più grande da quella calcolabile da uno dei singoli elaboratori in nostro possesso.

In particolare due istanze diverse dell’automa sono distribuite su più macchine secondo una topologia a griglia definita nel file “*configuration.txt”.* L’automa si evolve secondo le sue leggi locali, ma riceve in input informazioni sullo stato evolutivo delle macchine vicine.

Il file di configurazione si presenta secondo queste caratteristiche:

CellularAutomataRows: 100

CellularAutomataColumns: 100

CellularAutomataSteps: 1000

ProcessorsNumber: 9

MasterProcess: 0

Rows: 3

Columns: 3

SlaveDisplacement:

1 2 3

4 5 6

7 8 9

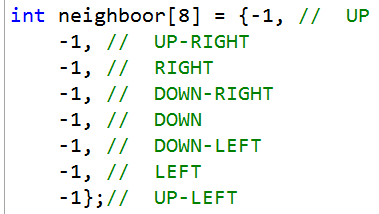
Executable: waveClusterMPI.exe

Dove in ordine, oltre ai dati relativi alla dimensione dell’automa, vengono specificati il numero di processi su cui eseguire il programma, il rank del processo master e la topologia secondo la quale è definito il vicinato (ad esempio il processo 5, i quale potrebbe anche risiedere su una macchina diversa, è vicino dei processi 2, 4, 6, 8).

# Elasticità per la gestione dati

Per consentire un codice più strutturato e semplice da leggere, evitando innumerevoli righe di codice, si è scelto di utilizzare apposite strutture dati in modo tale da ridurre il volume di codice ma dando allo stesso tempo più elasticità.

Tali strutture consistono in array per la gestione del “vicinato” di ogni processo. La prima che si può notare, che è poi la base per le altre è costituita da

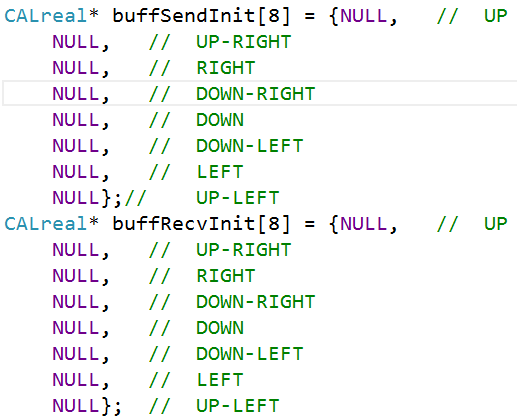


Essa dice ad ogni processo dove trovare i suoi vicini. Ogni cella del vettore dice se nella posizione i-esima vi è il vicino corrispondente, altrimenti il valore -1 indica che in quella direzione non vi è nessun vicino.

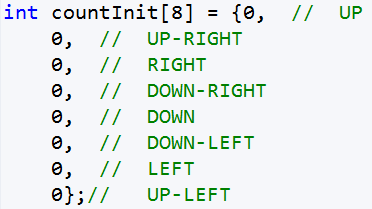
Esempio: la prima posizione neighboor[0]=2 indica che il processo corrente sopra di esso (si immaginino degli schermi messi vicini quanti sono i processi) ha il processo numero 2.

Seguono le strutture dati che contengono i buffer di invio/ricezione per ogni processo. La modalità di gestione è la stessa vista per la struttura “neighboor”. In tali strutture, valori diversi da “NULL” indicano la presenza di vicini e quindi che si può utilizzare tali strutture per la comunicazione dei dati.

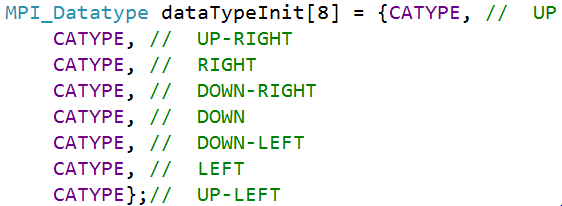
Tali strutture che seguiranno sono tutte collegate degli indici. Infatti neighboor[i], buffSendInit[i] e buffRecvInit[i] si riferiscono a valori utilizzati per la comunicazione con l’ i-esimo vicino.



La struttura “countInit” specifica le dimensioni dei buffer per la comunicazione con l’i-esimo vicino.



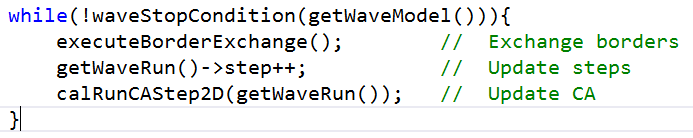
Infine “dataTypeInit” specifica il tipo di dato che bisogna scambiarsi con l’i-esimo vicino. In seguito alla fase di inizializzazione per ogni cella verra scelto il valore di data type corretto, se deve essere un tipo semplice MPI\_FLOAT, o un tipo derivato “rowtype” o “columntype”.



Esempio per la comunicazione con il vicino di destra i tipi di dati che bisogna scambiarsi sono le colonne ai bordi, quindi si utilizzerà un datatype di tipo “columntype”.

# Procedura principale

Viene ora presentato il ciclo principale che consente lo svolgimento dell’intero sistema.



In pratica questo ciclo viene eseguito da ogni processo insieme alla parte grafica OpenGL (che però per non causare freeze dell’intero sistema viene gestita in un Pthread a parte). Il ciclo sostanzialmente esegue le istruzioni finché non si verifica la condizione di stop dell’automa cellulare (fin quando non si sono raggiunti gli steps prestabiliti), quindi ad ogni iterazione si provvede ad aggiornare i bordi, si incrementa lo step dell’automa e si chiama la funzione di libreria OpenCAL *“calRunCAStep2D()”* che provvede ad applicare la funzione di transizione.

# Performance

Analizziamo dunque le prestazioni della simulazione:

Consideriamo la simulazione su un singolo processo con uno spazio dimensionale di 500 righe per 500 colonne. Il tempo atteso per una simulazione di questo genere lanciata su una singola macchina sarà sicuramente alto, senza contare che la simulazione stessa potrebbe non essere realistica a causa degli scatti grafici dovuti alla scarsa potenza della macchina su cui l’automa viene testato.

Pensiamo però di distribuire la simulazione su più calcolatori in modo tale che ciascuno di essi visualizzi ed elabori solo una porzione della matrice 500x500 (ad esempio la stessa simulazione su due calcolatori richiede che ogni nodo calcoli e visualizzi una 250x500) rendendo la simulazione più fluida con conseguente abbattimento dei tempi di esecuzione.

Il tempo sequenziale su una matrice 500x500 per 1000 step temporali risulta essere:

Lanciamo quindi la stessa simulazione su due macchine fisiche distinte, su una matrice 250x500. Poiché la simulazione avviene sincronizzando gli step temporali, consideriamo solo il tempo di esecuzione su uno dei due calcolatori.

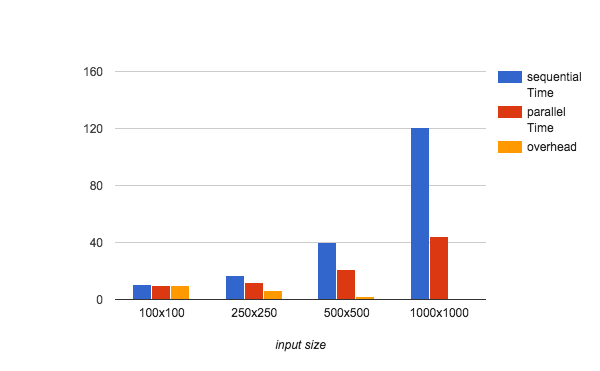
Si noti come i risultati delle simulazioni rispecchino quelli attesi.

Maggiori informazioni possono essere acquisite calcolando due importanti indici: Lo speedup e l’efficienza:

Lo speedup ci dà una stima dell’incremento della velocità di esecuzione nella versione parallela del programma rispetto alla sua versione sequenziale.

Come si può osservare lo speedup ottenuto non è proprio lineare come ci si poteva attendere, ma il valore ottenuto ci fa pensare che il sistema sia comunque ben parallelizzato e molto scalabile: infatti all’aumentare della dimensione dell’input, quindi della dimensione dello spazio, ci aspettiamo che il rapporto fra il tempo sequenziale e quello parallelo sia sempre costante (proprio grazie alla proprietà del sistema di dividere il problema in un sottoproblema di dimensione notevolmente inferiore da risolvere su calcolatori differenti).

Di seguito riportiamo il grafico relativo a quattro esecuzioni su un input crescente. Vengono messi a confronto il tempo seriale con il tempo di esecuzione parallelo, con particolare attenzione al tempo di overhead:



E’ interessante notare come il tempo di overhead (calcolato come ) venga completamente abbattuto al crescere dell’input.

Consideriamo quindi la sua efficienza, che dovrebbe fornirci una stima del contributo di ogni processo (nel nostro caso di ogni singola macchina) alla soluzione del problema:

Sappiamo che un’algoritmo è ben parallelizzato se la sua efficenza è almeno maggiore di .

Poichè dunque 0,95 > 0,50 allora il sistema risponde bene al calcolo distribuito.

# Considerazioni finali:

I dati relativi alla performance sono rassicuranti: sebbene non abbiamo raggiunto uno speedup lineare i valori garantiscono una buona scalabilità del sistema, sia al crescere della dimensione dell’input, sia al crescere del numero di processi.

La clusterizzazione è stata realizzata con il minimo dispendio di risorse in quanto, grazie alla stuttura di MPI, abbiamo potuto concentrarci sulla realizzazione del sistema su singola macchina per poi estendere la simulazione solo nella fase finale del lavoro.