1-D Wave Equation in MPI

Salvatore Calderaro

19 gennaio 2021

Università degli Studi di Palermo

Sommario

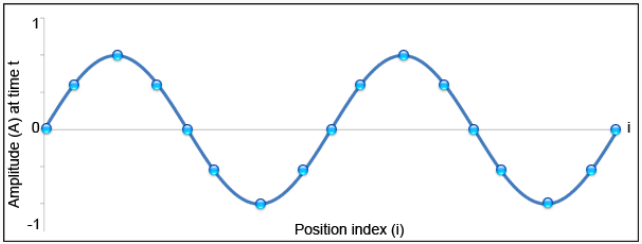
Descrizione del problema

L'equazione d'onda

In uno spazio monodimensionale, l'equazione d'onda può essere espressa come segue:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

Quello che si vuole calcolare è l'ampiezza lungo una stringa uniforme e vibrante dopo che è trascorso un determinato periodo di tempo.



L'equazione d'onda

L'equazione può essere risolta usando il metodo delle differenze finite e si ottiene:

$$u_j^{k+1} = 2u_j^k - u_j^{k-1} + c^2 \frac{\Delta t^2}{\Delta x^2}(u_{j+1}^k - 2u_j^k + u_{j-1}^k)$$

dove k rappresenta l'indice temporale e j quello spaziale.

L'ampiezza dunque dipenderà dai timestep precedenti $(k, k-1)$ e dai punti vicini $(j-1, j+1)$.

Versione Seriale

Inserimento dei parametri

Dopo aver settato i parametri base, tramite la funzione *init_param()* è permesso all'utente l'inserimento dei seguenti dati:

- numero di punti (tp);
- numero di timesteps (ns).

Le variabili maggiormente utilizzate durante il calcolo sono i seguenti tre array:

1. *values*: contiene i valori dell'ampiezza dell'onda al tempo t
2. *old_values*: contiene i valori dell'ampiezza dell'onda al tempo $t - dt$;
3. *new_values*: contiene i valori dell'ampiezza dell'onda al tempo $t + dt$.

Creazione della sinusoide iniziale

Tramite la funzione *create_line()*, viene creata l'onda iniziale basandosi su una curva di tipo sinusoidale seguendo la seguente formula:

$$a[i] = \sin\left(2\pi \cdot \frac{k}{tmp}\right)$$

con $i = 1, \ldots, tp$, $tmp = tp - 1$. k inizialmente è settato a 0 e viene incrementato di 1 ad ogni iterazione. In questo modo, il punto iniziale e quello finale avranno ampiezza 0 e dunque si ha un'onda stazionaria.

Aggiornamento dei valori per tutti i timesteps

L'aggiornamento dei valori viene fatto mediante la funzione *update()*. Una volta fissati

$$dt = 0.3 \quad c = 1 \quad dx = 1 \quad \tau = c \cdot \frac{dt}{dx}$$

i punti iniziali e finali vengono posti a 0, gli altri vengono calcolati come segue:

$$new\_values[j] = (2.0 \cdot values[j]) - old\_values[j] + (sqtau \cdot (values[j-1] - (2.0 \cdot values[j]) + values[j+1]))$$

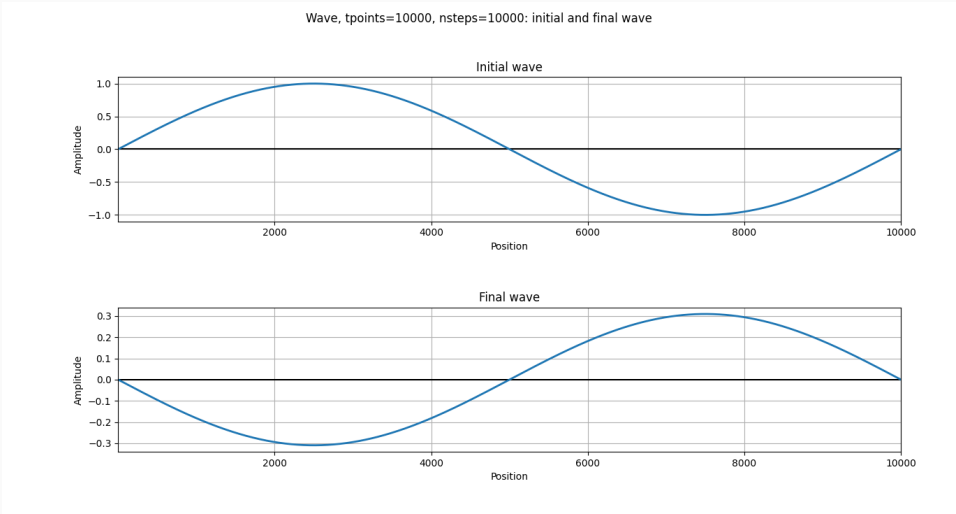
dove *sqtau*$= \tau^2$. Dopo di ciò i valori presenti in *values* vengono copiati in *old_values* e quelli presenti in *new_values* in *values*.

Visualizzazione dei risultati

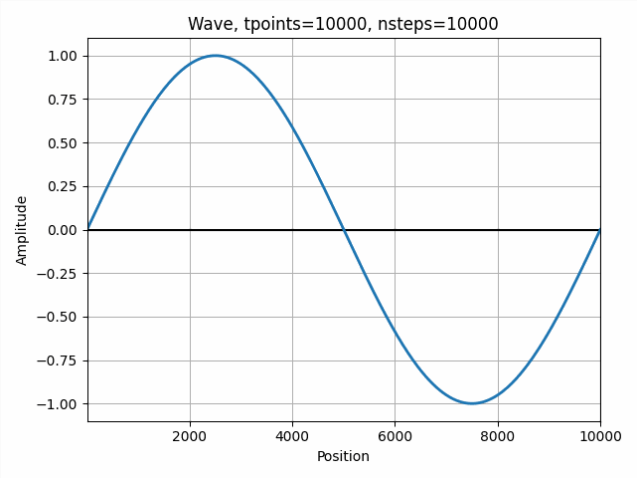
Dopo l'aggiornamento dei valori per ogni timestep vengono prodotti:

1. file txt contenente i valori finali dell'ampiezza dell'onda;
2. grafico contenente l'onda iniziale e quella finale;
3. gif che mostra l'andamento dell'onda.

Visualizzazione dei risultati



Visualizzazione dei risultati



Versione Parallela

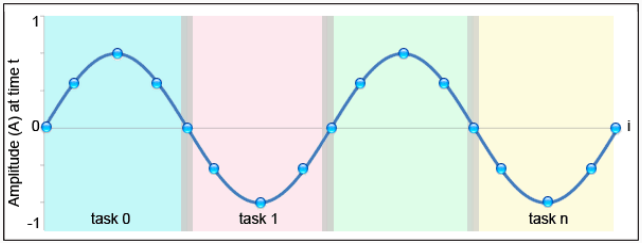
Modello utilizzato

La parallelizzazione può essere effettuata mediante un modello **SPMD** (Single Program Multiple Data) in cui:

- il **master** invia le informazioni iniziali ai workers e dopo aver processato anche la sua parte attende per aggregare i risultati provenienti da tutti i **worker**;
- i **worker** calcolano i valori dell'ampiezza per un numero specificato di timestep comunicando con i **worker** vicini.

Strategia di parallelizzazione

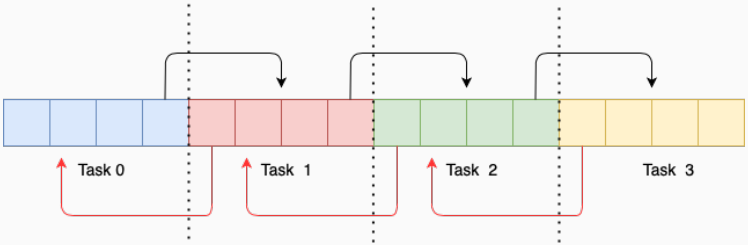
Per parallelizzare il codice, l'array contenente le ampiezze dei punti dell'onda è partizionato in subarray e quest'ultimi sono distribuiti ai vari task.



Ogni task processerà circa lo stesso numero di punti. Tutti i punti richiedono la stessa quantità di lavoro e quindi vi è **load balancing**. Effettuando dunque una divisione a blocchi, ogni task processerà una serie contigua di punti.

Comunicazione

Le comunicazioni fra i processori sono necessarie solamente ai bordi tra una partizione dell'array ed un'altra.



Oltre a queste, abbiamo infine le comunicazioni che intercorrono tra i **worker** e il **master** per l'invio dei dati finali e la loro relativa aggregazione.

Descrizione generale del programma

Il programma esegue i seguenti step:

1. controllo del numero di task inseriti da linea di comando;
2. inserimento dei parametri da parte dell'utente e trasmissione dei dati dal **master** ai **worker**;
3. creazione della sinusoide iniziale (ogni processore calcolerà la sua porzione);
4. aggiornamento dei valori dell'ampiezza per ogni timestep;
5. aggregazione da parte del **master** dei risultati inviati dai **worker**.

Insrimento dei parametri e trasmissione da parte del master

Tramite la funzione *init_master()* è permesso all'utente l'inserimento dei dati (il numero di punti e di timesteps). Dopo tutti i controlli del caso sulla correttezza dei dati inseriti, il **master** li trasmette ai **worker**. Quest'ultimi riceveranno i dati grazie alla funzione *init_workers()*.

Creazione della sinusoide iniziale

Per la creazione della sinusoide iniziale viene usata la funzione *init_line()* che esegue seguenti step:

1. calcolo del numero di punti che dovranno essere processati da ogni task (*nmin*) e del numero di eventuali exstrapoint (*nleft*) (qualora il numero di punti non fosse divisibile esattamente per il numero di task);
2. ogni processore:
   - 2.1 stabilisce il numero di punti (*npts*) da calcolare compreso di eventuali extrapoint;
   - 2.2 calcola i valori della sinusoide iniziale cosi come fatto per il codice seriale;
   - 2.3 copia i valori appena calcolati nell'array *old_values*.

Aggiornamento dei valori per ogni timestep

L'aggiornamento dei valori per ogni timestep viene effettuato tramite la funzione *update()* in cui:

1. vengono identificati i vicini *left* e *right* del **worker** corrente, mediante la funzione *identify_left_right_processors()*;
2. scambio dei dati con il **worker** sinistro:
   - invio a *left* il primo punto;
   - ricevo da *left* il suo ultimo punto.
3. scambio dei dati con il **worker** destro:
   - invio a *right* l'ultimo punto ;
   - ricevo da *right* il suo primo punto.
4. il **worker** aggiorna la sua porzione di array seguendo lo stesso algoritmo utilizzato per la versione seriale.

Raccolta ed aggregazione finale dei risultati

In conclusione:

1. tramite la funzione *output_master()*, il **master** riceve da ciascun **worker**:
   - un array contenente l'indice di partenza e il numero di punti elaborati;
   - un array contenente i valori finali calcolati.

   e conoscendo tali informazioni può collocare i dati nella posizione corretta e restituire cosi l'array con i risultati finali.
2. tramite la funzione *output_workers()*, ciascun **worker** invia al **master**:
   - un array contenente l'indice di partenza e il numero di punti elaborati;
   - un array contenente i valori finali calcolati.

Analisi dei risultati

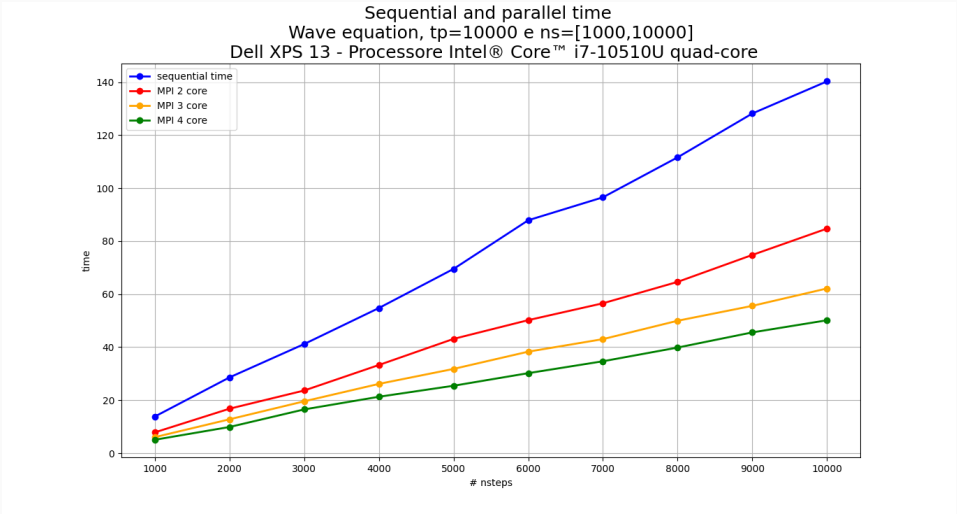
Analisi effettuate

Una volta calcolati i tempi (seriale e parallelo) sono stati realizzati dei grafici inerenti **tempo**, **speedup** ed **efficienza** per i seguenti tre casi:

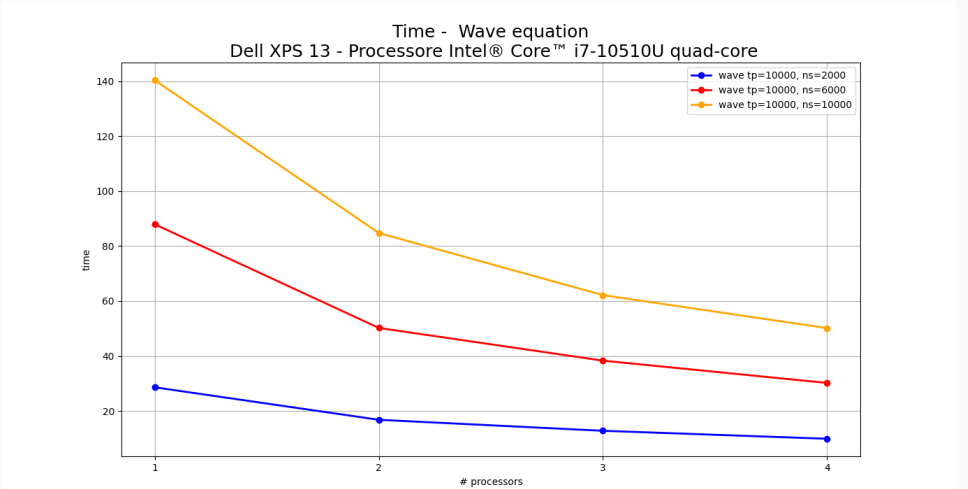
1. numero di punti fissato ($tp = 10000$) e numero di timestep variabile ($ns = [1000, 10000]$) con step di 1000;
2. numero di timestep ($ns = 10000$) fissato e numero di punti variabile ($tp = [1000, 10000]$) con step di 1000;
3. numero di timestep ($ns = 10000$) fissato e numero di punti variabile ($tp = 10, 100, 1000, 10000$).

Per la misura dei tempi i programmi sono stati eseguiti su di un *Dell XPS 13 - Intel® Core™ i7-10510U quad-core* sotto le medesime condizioni.

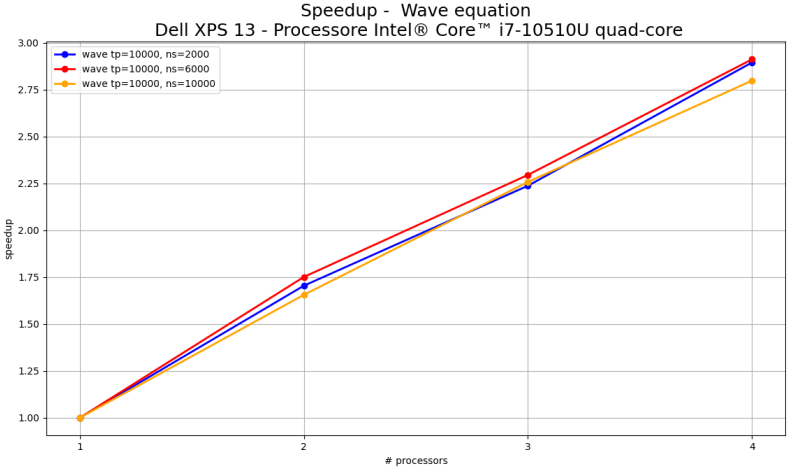
Numero di punti fissato e timestep variabile - Time



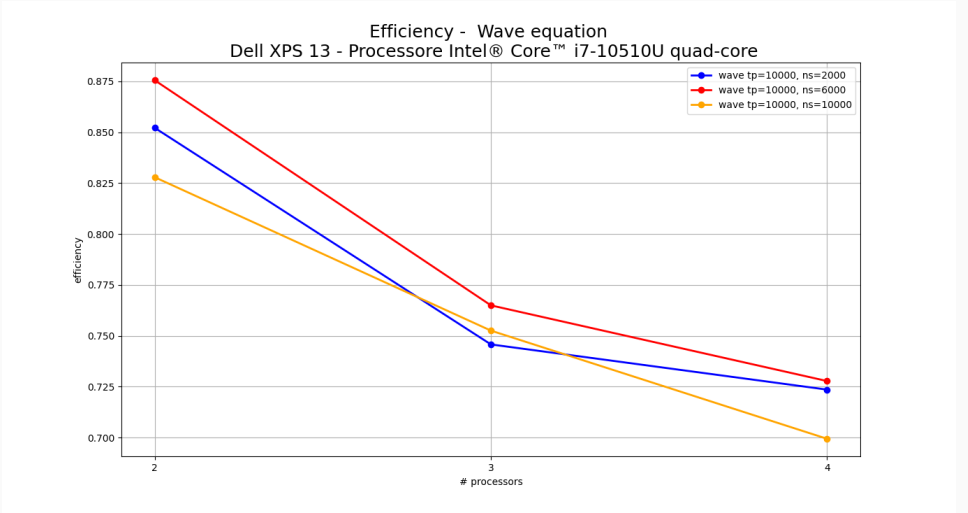
Numero di punti fissato e timestep variabile - Time



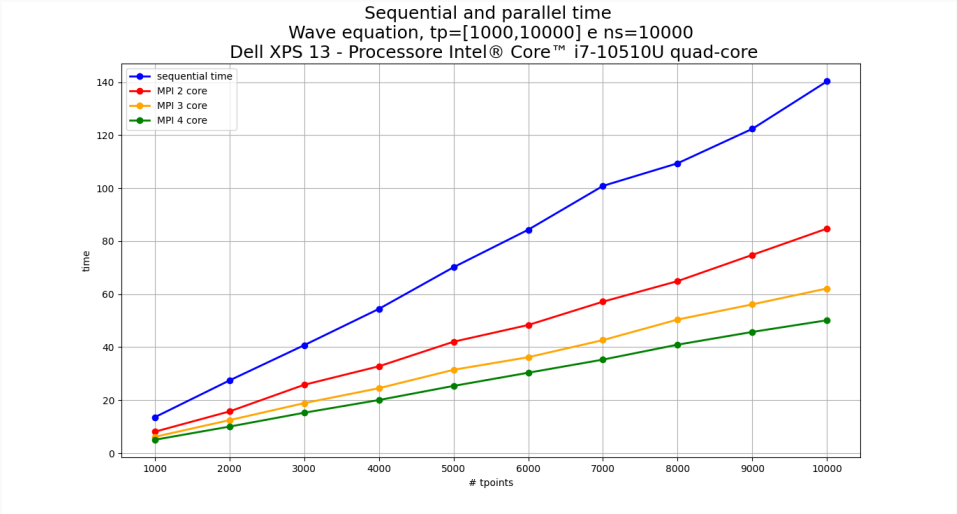
Numero di punti fissato e timestep variabile - Speedup



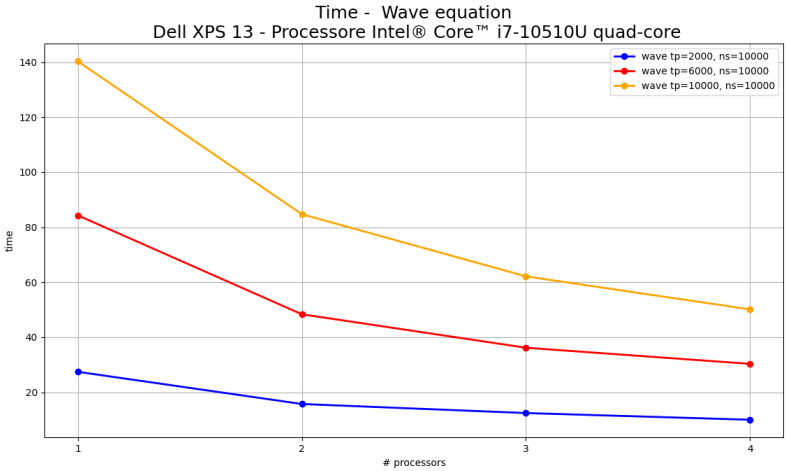
Numero di punti fissato e timestep variabile - Efficiency



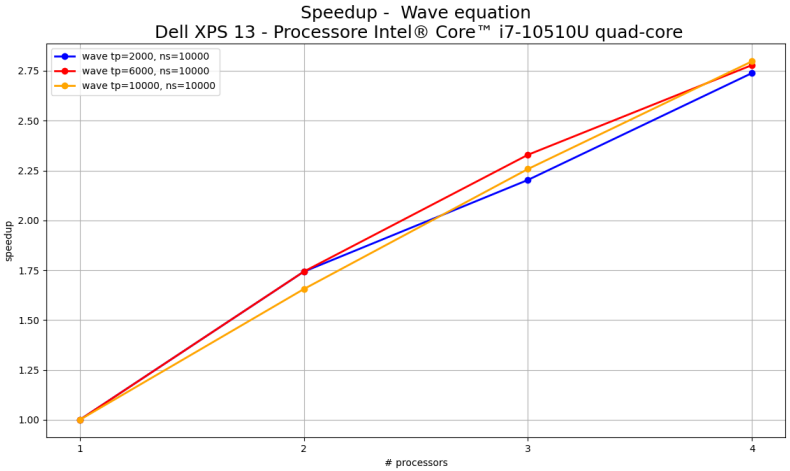
Numero di timestep fissato e numero di punti variabile - Time



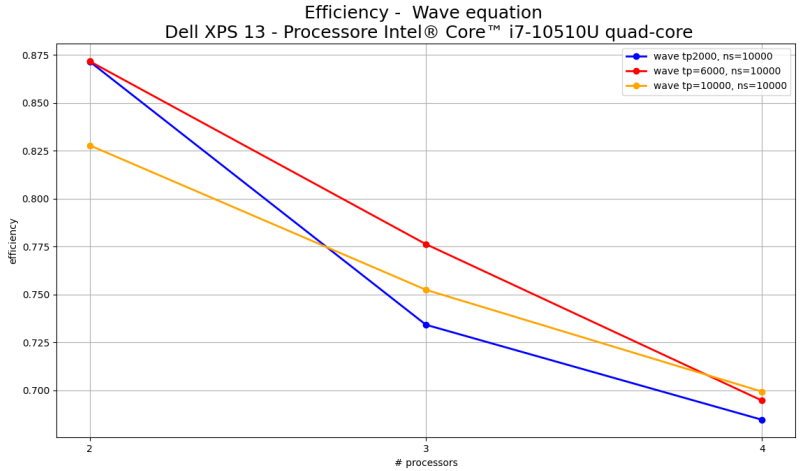
Numero di timestep fissato e numero di punti variabile - Time



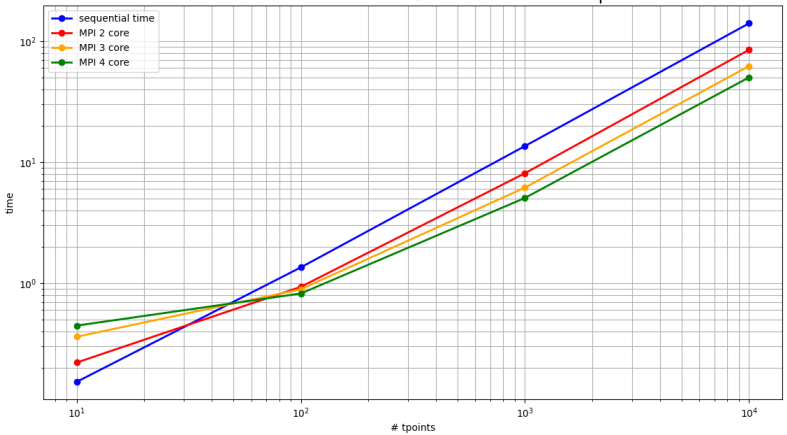
Numero di timestep fissato e numero di punti variabile - Speedup



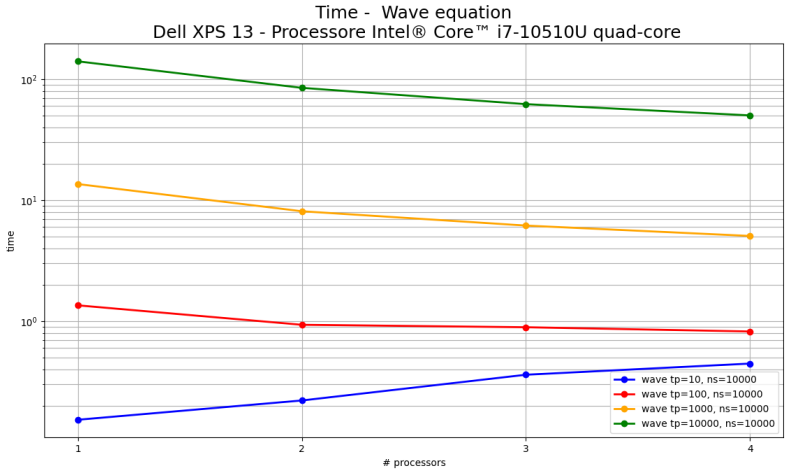
Numero di timestep fissato e numero di punti variabile - Efficiency



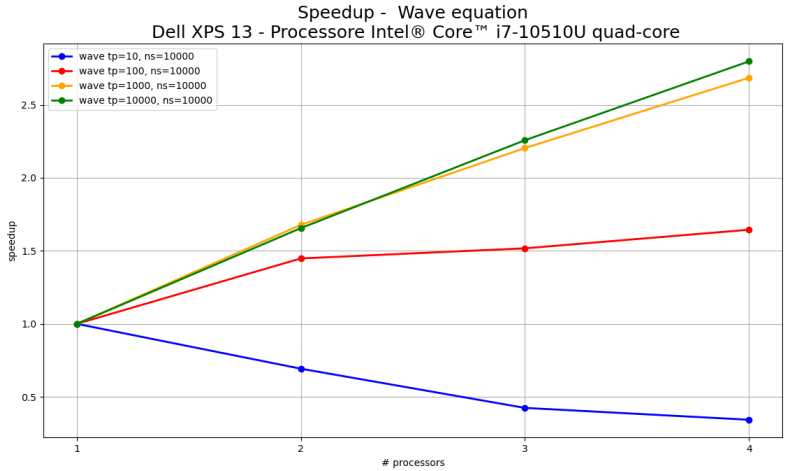
Numero di timestep fissato e numero di punti variabile (log scale)- Time



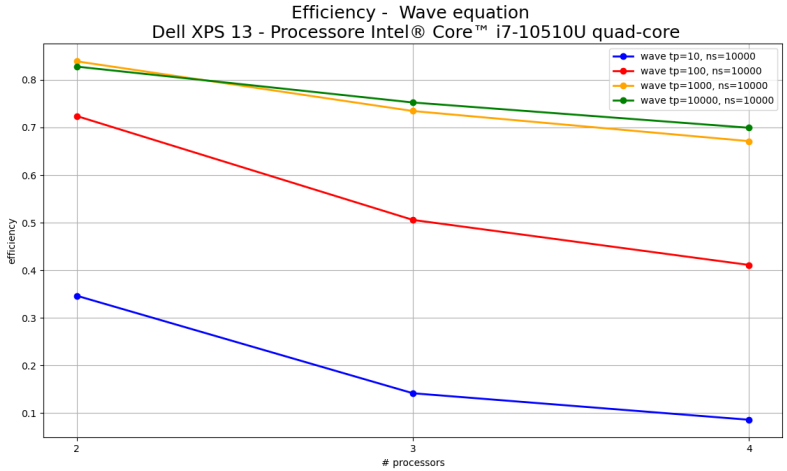
Numero di timestep fissato e numero di punti variabile (log scale) - Time



Numero di timestep fissato e numero di punti variabile (log scale) - Speedup



Numero di timestep fissato e numero di punti variabile (log scale) - Efficiency



Conclusioni

Conclusioni

Conclusioni

In conclusione, grazie all'utilizzo del calcolo parallelo si è riuscito ad abbassare il tempo di esecuzione del programma. Analizzando i grafici mostrati nelle slide precedenti si evince che il numero di processori ideale che si dovrebbe utilizzare è due , in quanto per tale valore si raggiungono i valori più alti di *efficienza*. Tuttavia anche con un numero di processori pari a tre e quattro si ottengono buoni di valori di efficienza.

Il codice sorgente è reperibile alla seguente repository GitHub:
https://github.com/Calder10/Serial-and-Parallel-Wave-Equation-Python.

Grazie per l'attenzione !