Esercizio 1

Per migliorare le prestazioni dell’algoritmo Merge Sort, è possibile introdurre l’utilizzo di un ForkJoinPool.

Estendiamo la classe MergeSort con RecursiveAction, per utilizzare il metodo compute(), che si occuperà di dividere le operazioni in task più piccoli, da eseguire più agevolmente. Bisogna poi impostare una soglia sotto la quale impedire di eseguire ulteriori divisioni: in questo modo, non si rischia che l’overhead causato dai vari task sia maggiore del tempo necessario ad eseguire effettivamente l’algoritmo di sort.

Per la divisione in task, viene utilizzato il metodo statito invokeAll() che si occupa prima di divedere in task (fork) e in seguito di aspettare i task per poi poterli riunire (join), sollevando il compito dal programmatore.

**protected** **void** **compute**()

{

**if** (lo >= hi)

**return**;

**final** **int** **mid** = lo + (hi - lo) / 2;

**final** **int** **lenght** = hi - lo;

**if**(lenght>***SOGLIA***)

*invokeAll*(**new** MergeSort(a, helper, lo, mid), **new** MergeSort(a, helper, mid+1, hi));

**else**

{

*mergesort*(a, helper, lo, mid);

*mergesort*(a, helper, mid + 1, hi);

}

*merge*(a, helper, lo, mid, hi);

}

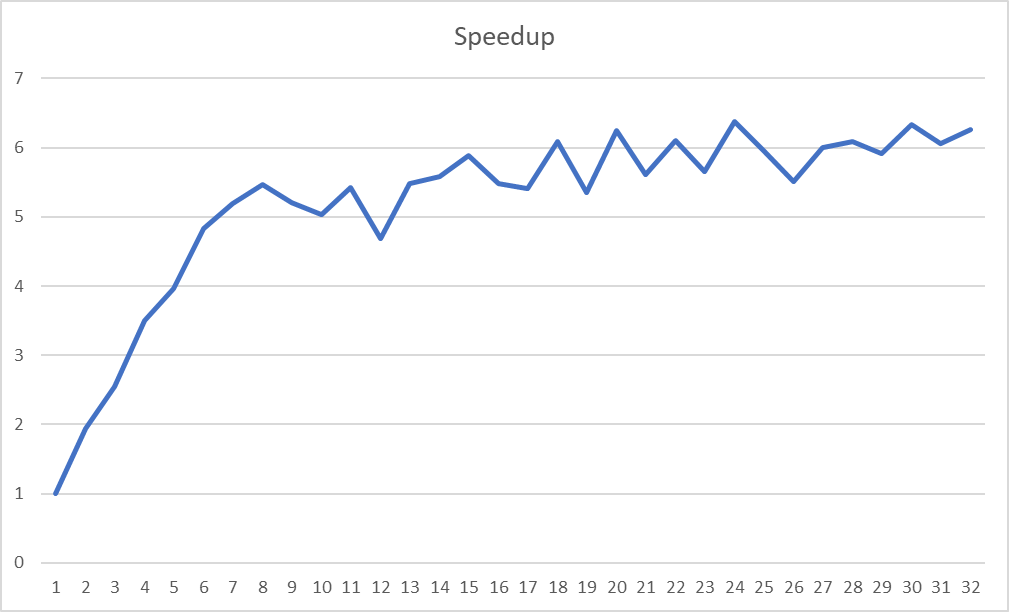
Esercizio 2

Per eseguire le misurazioni di tempo, è necessario implementare una classe Timer per registrare i tempi di esecuzione, da cui poi calcolare lo speedup. Bisogna poi eseguire l’applicazione variando, ad ogni esecuzione, il numero di thread utilizzati per completare il disegno.

Dalle misurazioni otteniamo la seguente tabella:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Thread** | **Tempo** | **Speedup** | **Thread** | **Tempo** | **Speedup** |
| 1 | 13672 | 1 | 17 | 2528 | 5,4082278 |
| 2 | 7023 | 1,9467464 | 18 | 2248 | 6,0818505 |
| 3 | 5365 | 2,5483691 | 19 | 2556 | 5,3489828 |
| 4 | 3898 | 3,5074397 | 20 | 2187 | 6,2514861 |
| 5 | 3445 | 3,9686502 | 21 | 2435 | 5,6147844 |
| 6 | 2830 | 4,8310954 | 22 | 2238 | 6,1090259 |
| 7 | 2636 | 5,1866464 | 23 | 2417 | 5,6565991 |
| 8 | 2499 | 5,4709884 | 24 | 2143 | 6,3798413 |
| 9 | 2627 | 5,2044157 | 25 | 2294 | 5,9598954 |
| 10 | 2718 | 5,0301692 | 26 | 2484 | 5,5040258 |
| 11 | 2522 | 5,4210944 | 27 | 2277 | 6,0043917 |
| 12 | 2917 | 4,6870072 | 28 | 2244 | 6,0926916 |
| 13 | 2492 | 5,4863563 | 29 | 2314 | 5,9083838 |
| 14 | 2448 | 5,5849673 | 30 | 2158 | 6,3354958 |
| 15 | 2324 | 5,8829604 | 31 | 2254 | 6,065661 |
| 16 | 2494 | 5,4819567 | 32 | 2183 | 6,2629409 |

Questo è invece il grafico ottenuto dai dati nella tabella:



Esercizio 3

Per testare le prestazioni dell’applicazione, utilizziamo un Flight Recorder. Per le misurazioni, ho deciso di registrare la versione con 1 thread, 8 thread e 32 thread.

In tutte e 3 le prove, la memoria è principalmente utilizzata da array di interi.

Immagine che contiene screenshot, testo

Descrizione generata con affidabilità molto elevata

*1 thread*

Immagine che contiene testo, screenshot, mappa

Descrizione generata con affidabilità elevata

*8 thread*

Immagine che contiene screenshot

Descrizione generata con affidabilità molto elevata

*32 thread*