Matteo Muraca, mat. 201074, CdL Informatica

OpenMP Spring Bonus Report

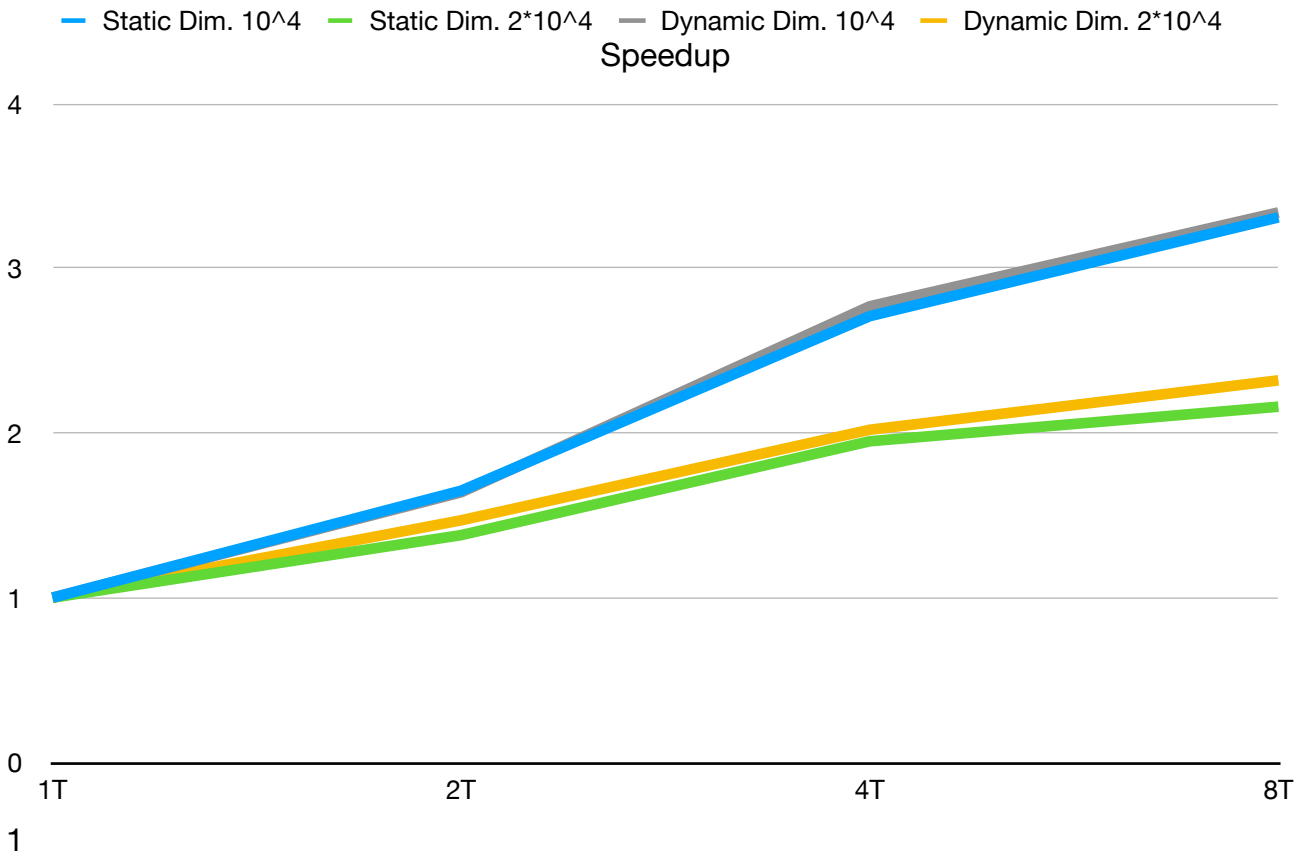
Il dispositivo utilizzato è un MacBook Pro 2019, dotato di un Intel i5-8257U con 4+4 thread e frequenza di 1,4GHz. Per testare i metodi basta compilare ed eseguire springbonus.cpp.

Esercizio 1 - parallel 2D Matrix computation

Per quanto riguarda le due matrici la massima dimensione che ho testato è di 20000, in quanto numeri più grandi generavano errori di killing dei processi.

Dimensione 10^4	Static	Speedup	Dynamic	Speedup
1 Thread (Seriale)	837,6 ms	1	837,6 ms	1
2 Threads	507,3 ms	1,65	509,7 ms	1,64
4 Threads	308,9 ms	2,71	302,8 ms	2,77
8 Threads	252,8 ms	3,31	251,1 ms	3,34

Dimensione $2*10^4$	Static	Speedup	Dynamic	Speedup
1 Thread (Seriale)	4145,7 ms	1	4145,7 ms	1
2 Threads	3004,0 ms	1,38	2819,9 ms	1,47
4 Threads	2121,6 ms	1,95	2054,0 ms	2,02
8 Threads	1916,3 ms	2,16	1784,0 ms	2,32



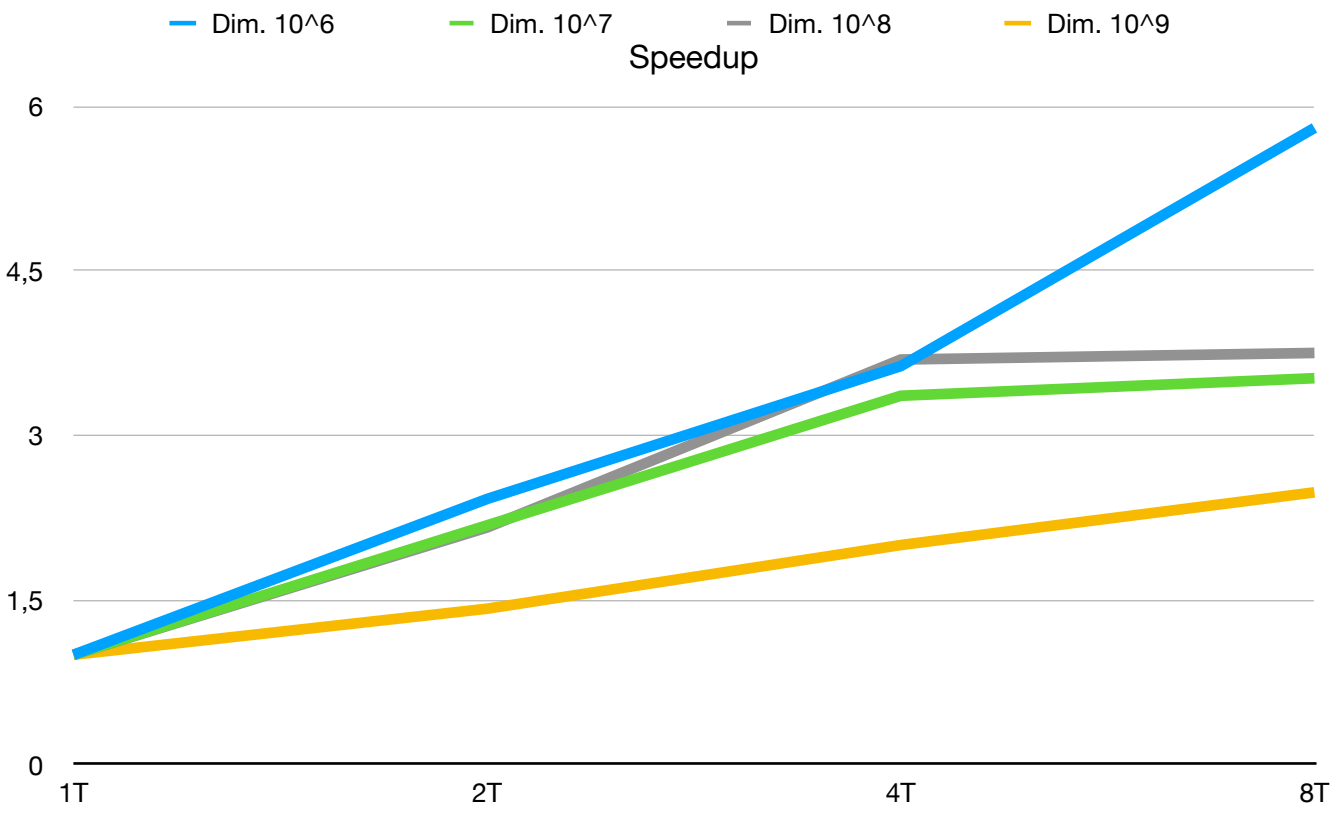
1

Esercizio 2 - Vectors Computation

In questo esercizio ho generato i due vettori A e B in maniera casuale, con una funzione da me scritta nella libreria myTools.h (fine report) e basata su rand_r.
Inoltre ho utilizzato la riduzione per calcolare il prodotto scalare tra i vettori B e C.

Tempo	Dimensione 10^6	Dimensione 10^7	Dimensione 10^8	Dimensione 10^9
1 Thread	2,9 ms	22,5 ms	305,6 ms	28397,9 ms
2 Threads	1,2 ms	10,3 ms	141,4 ms	20025,0 ms
4 Threads	0,58 ms	6,7 ms	82,8 ms	14245,7 ms
8 Threads	0,5 ms	6,4 ms		11436,4 ms

Speedup	Dimensione 10^6	Dimensione 10^7	Dimensione 10^8	Dimensione 10^9
2 Threads	2,42	2,18	2,16	1,42
4 Threads	3,63	3,36	3,69	2,00
8 Threads	5,80	3,52	3,75	2,48



Esercizio 3 - Calculation of PI

Tutti i dati riguardo tempi e speedup sono visionabili sullo spreadsheet allegato.

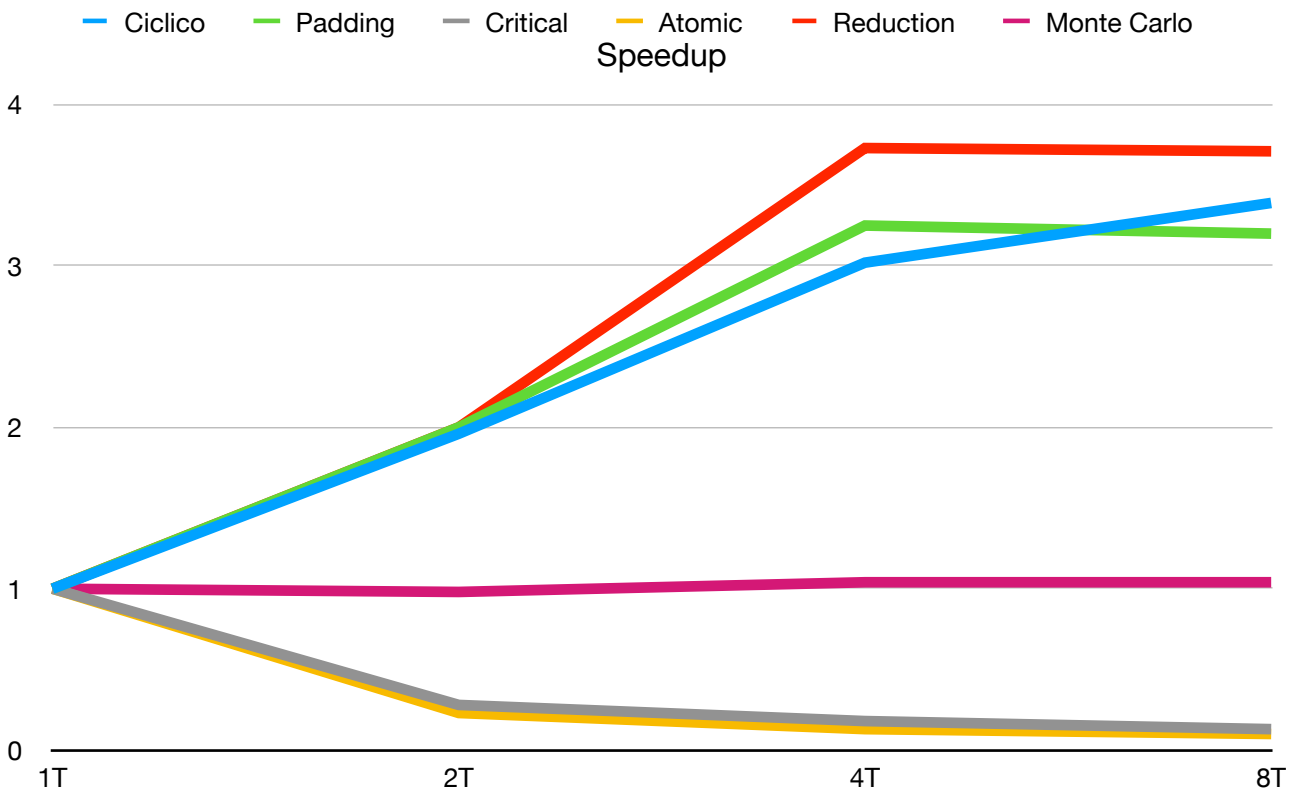
Il primo metodo è il classico algoritmo ciclico, basato sull'approssimazione dell'integrale. La scrittura dei risultati avviene su un vettore di somme, ogni thread ha la propria casella, e la somma finale viene fatta in sequenziale.

Nel secondo metodo il padding elimina le dispute nella scrittura in cache ed incrementa le prestazioni utilizzando, anziché un vettore su cui i thread scrivono in contemporanea, una matrice in cui ognuno scrive sulla prima colonna.

I due successivi metodi si basano su un'unica variabile somma anziché un vettore o matrice, dunque ho utilizzato i paradigmi critical e atomic: essi rallentano notevolmente l'algoritmo rispetto al classico ciclico, poiché i thread devono fermarsi per verificare se la variabile non è utilizzata dagli altri, e si può notare la grande differenza di esecuzione di un'operazione atomica rispetto ad una sezione critica.

Il metodo con riduzione parte dal principio ideato nei precedenti, ma si ottengono tempi di esecuzione molto minori, poiché ogni thread scrive su una variabile privata, ed in seguito il main thread esegue la somma di tutte queste variabili.

Il metodo Monte Carlo differisce dagli altri per la modalità in cui calcola il valore del pi.
La probabilità che un punto si trovi in un cerchio inscritto in un quadrato è data dal rapporto tra le aree di cerchio e quadrato, e sarà approssimabile al rapporto tra n di punti che ricadono nel cerchio e n di tentativi totali: con r=1 e l=2, avremo P(punto) = π/4, con la formula inversa si ricava π. L'errore è molto casuale in quanto dipende da un random. Anche in questo caso ho usato il randomNumber di myTools, con un seed per ogni thread.

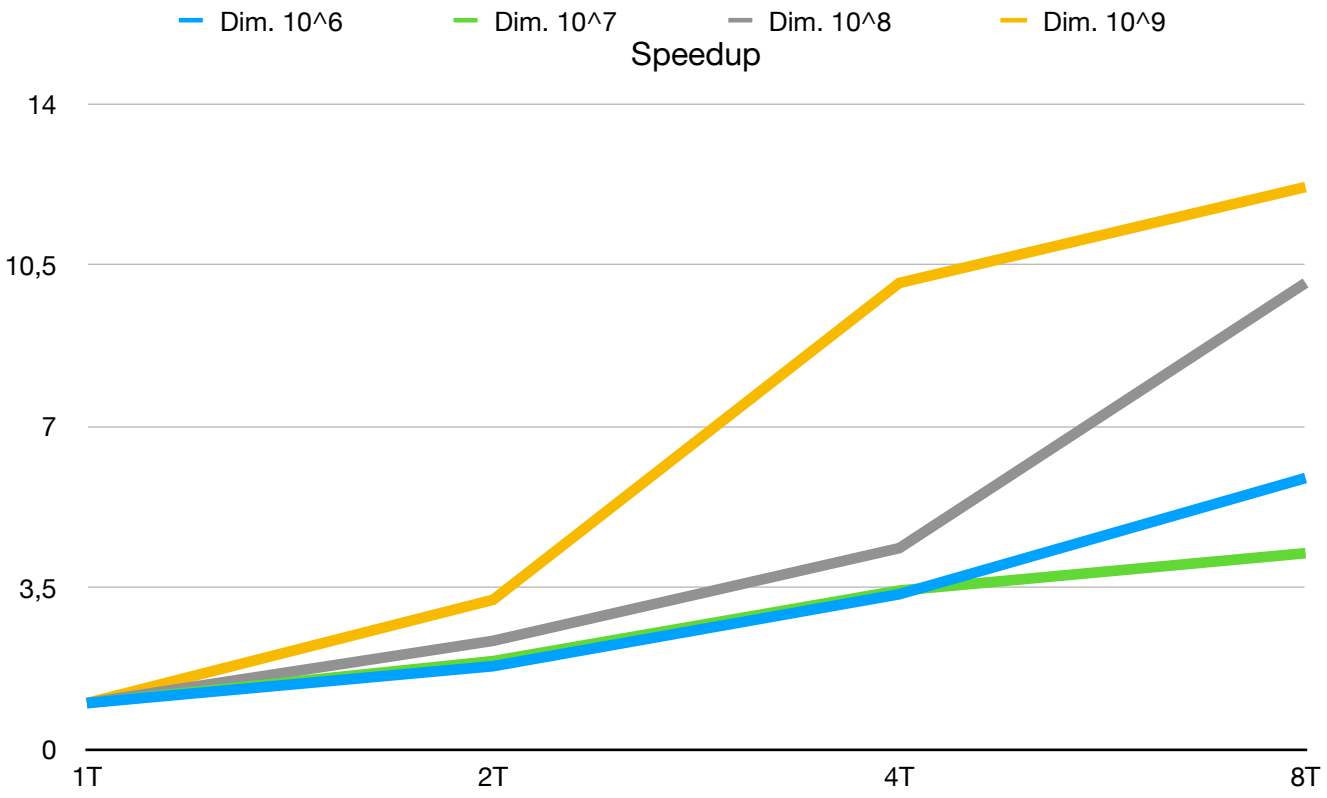


Esercizio 4 - Find an element in a vector

Nel caso di un vettore riempito senza uno specifico criterio, il metodo ottimale per trovare un numero al suo interno è la ricerca lineare. Dunque, ho scritto un metodo che compie la ricerca lineare dello stesso numero casuale in uno stesso vettore generato casualmente.
I tempi riportati nelle tabelle sottostanti si riferiscono allo scorrimento dell'intero vettore, senza trovare il valore.

Tempo	Dimensione 10^6	Dimensione 10^7	Dimensione 10^8	Dimensione 10^9
1 Thread (Seriale)	4,7 ms	37,8 ms	319,5 ms	7387,9 ms
2 Threads	2,6 ms	19,8 ms	136,0 ms	2283,2 ms
4 Threads	1,4 ms	11,0 ms	73,2 ms	730,7 ms
8 Threads	0,8 ms	8,9 ms	62,5 ms	606,3 ms

Speedup	Dimensione 10^6	Dimensione 10^7	Dimensione 10^8	Dimensione 10^9
2 Threads	1,80	1,91	2,35	3,24
4 Threads	3,36	3,44	4,36	10,11
8 Threads	5,88	4,25	5,11	12,19

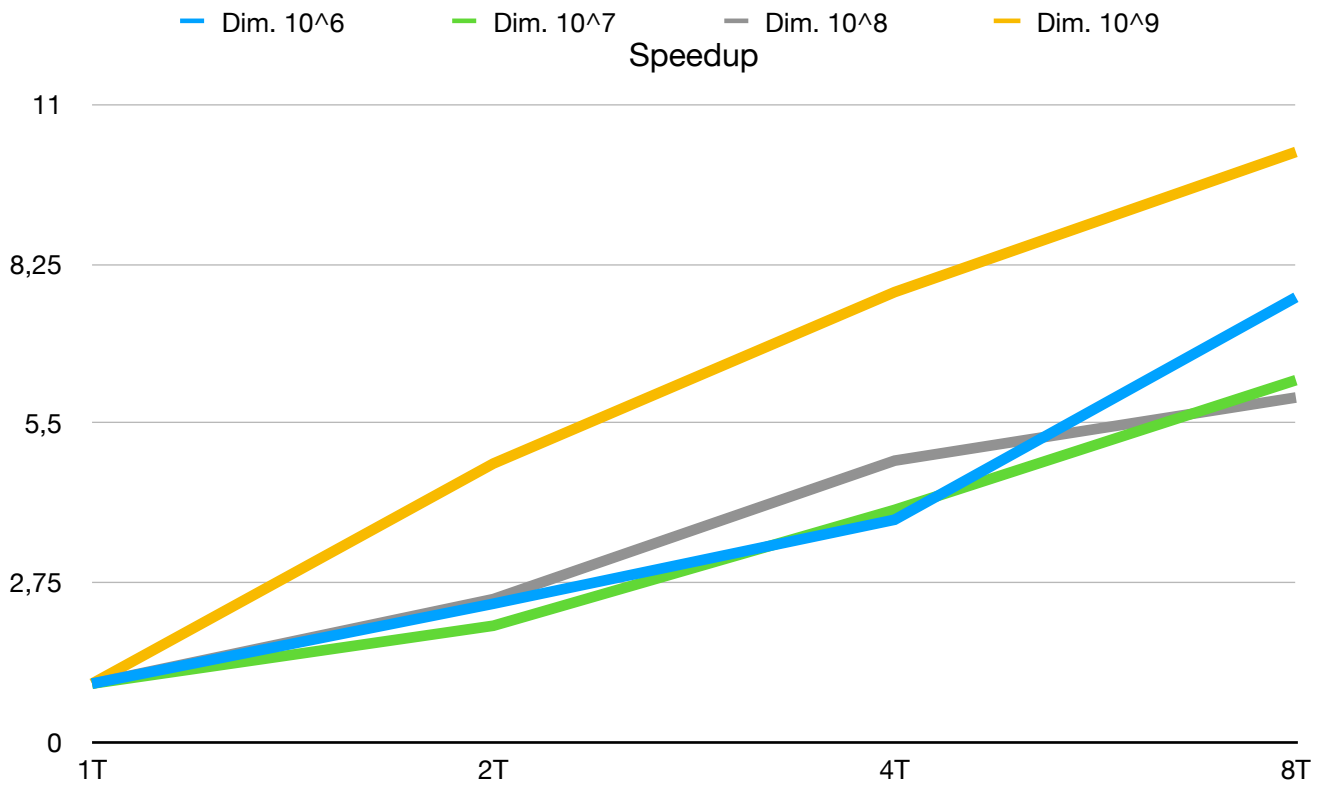


Matteo Muraca, mat. 201074, CdL Informatica

Le tabelle sottostanti riguardano la ricerca di un elemento situato in posizione dim-1.
Si può notare anche qui come lo speedup sia direttamente proporzionale alle dimensioni.

Tempo	Dimensione 10^6	Dimensione 10^7	Dimensione 10^8	Dimensione 10^9
1 Thread (Seriale)	6,9 ms	36,8 ms	317,4 ms	6019,0 ms
2 Threads	2,9 ms	18,4 ms	128,7 ms	1255,9 ms
4 Threads	1,8 ms	9,2 ms	65,5 ms	775,7 ms
8 Threads	0,9 ms	5,9 ms	53,4 ms	591,3 ms

Speedup	Dimensione 10^6	Dimensione 10^7	Dimensione 10^8	Dimensione 10^9
2 Threads	2,38	2,00	2,46	4,80
4 Threads	3,83	4,00	4,85	7,76
8 Threads	7,67	6,24	5,94	10,18

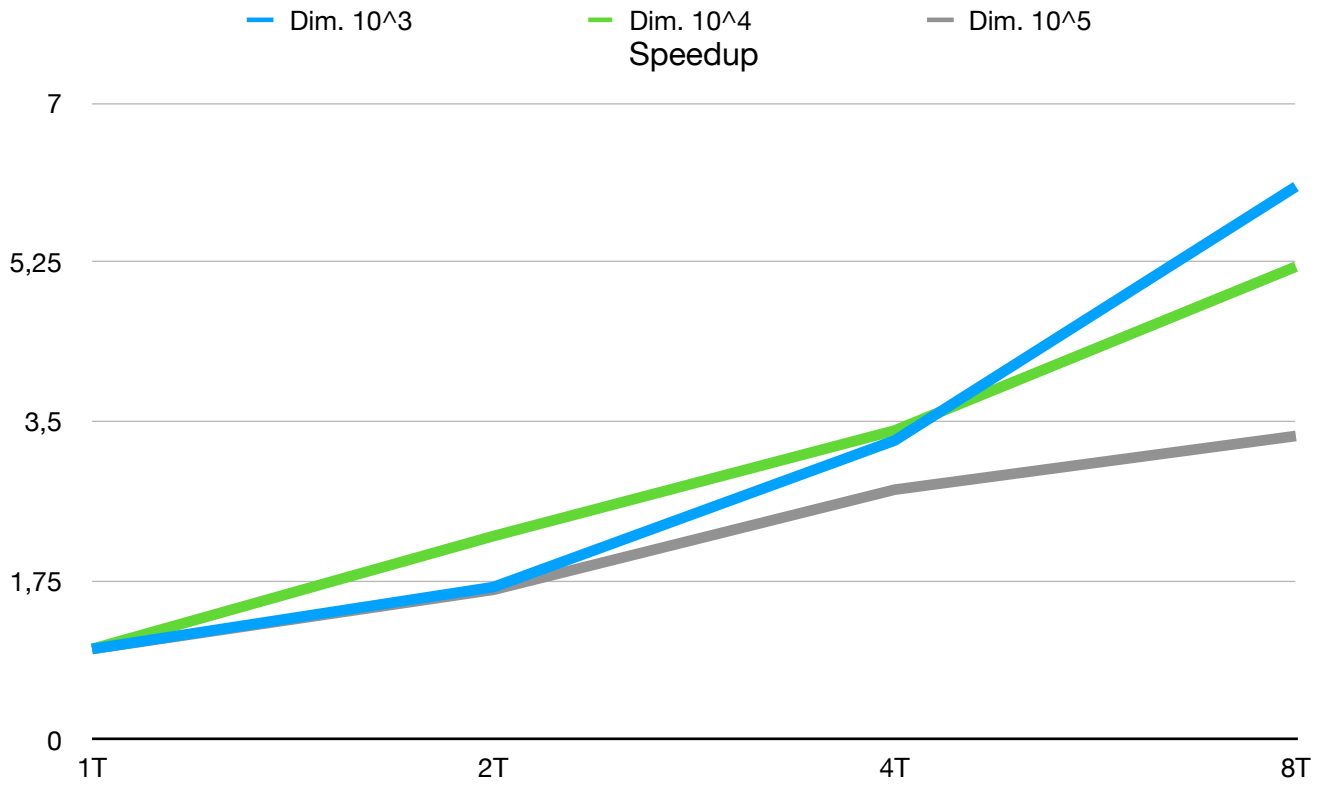


Esercizio 5 - Conway's Game of Life

Data una dimensione dim, la matrice viene generata con dimensione (dim+2)*(dim+2), con una cornice che resta vuota e non viene controllata, che serve ad evitare di implementare codice in più per il controllo delle celle sui bordi. I tempi misurati si riferiscono alla computazione di tutta la nuova matrice e sostituzione della precedente.

Tempo	Dimensione 10^3	Dimensione 10^4	Dimensione 10^5
1 Thread (Seriale)	7,9 ms	549,9 ms	57021,2 ms
2 Threads	4,7 ms	245,7 ms	34528,6 ms
4 Threads	2,4 ms	161,5 ms	20732,3 ms
8 Threads	1,3 ms	105,8 ms	17085,3 ms

Speedup	Dimensione 10^3	Dimensione 10^4	Dimensione 10^5
2 Threads	1,68	2,24	1,65
4 Threads	3,29	3,40	2,75
8 Threads	6,08	5,20	3,34



myTools.h

Ho scritto una semplice libreria contenente alcuni metodi utili nei vari esercizi.

Contiene i seguenti metodi:
•vectorOut, metodo che semplicemente dà in standard output un vettore di tipo T;
•deleteMatrix, metodo che elimina dalla memoria una matrice dinamica;
•deleteMatrixParallel, un deleteMatrix che usa il calcolo parallelo: ho notato un notevole miglioramento di prestazioni;
•randomVec, genera un nuovo vettore con valori casuali;
•randomVecParallel, un randomVec che sfrutta il calcolo parallelo;
•randomNumber, genera un numero casuale sfruttando il rand_r, in due varianti, la prima crea un seed e la seconda ne riceve uno in input;
•inputNumThreads, il nome dice tutto, funzione scritta per evitare copia e incolla;
•ready, riceve in input la risposta a domande come “sei pronto?” o “hai capito?”.

Oltre ai metodi, ho incluso la costante PI24, ovvero pi greco con le prime 24 cifre decimali.