Mattia Villani
502347
mattiavillani94@gmail.com

Metodo di Jacobi

> The application finds the solution of a system of linear equations using the Jacobi iterative method

Le librerie usate sono Fastflow ed pthread con linguaggio C++. Un file di script bash aggiuntivo ed un parser in python sono stati realizzati ed usati per creare ed analizzare i risultati.

Design

Consideriamo allora *N* (lunghezza del vettore x_i) come la misura della complessità del problema, ed *K* come il numero di iterazioni effettuate, con *K>0*. Assumiamo inoltre di considerare il tempo utente e quindi non preoccuparci dei tempi di IO che potrebbero derivare dalle operazioni di lettura da file.
Il metodo di Jacobi è composto dalle seguenti fasi:

- F1: Setting up del programma. *Tempo costante* t_{F1}.
- F2: Lettura del problema.
  - F2a: Inizializzazione di A. *Tempo* N^2.
  - F2b: Inizializzazione di b. *Tempo N.*
- F3: Setting up del problema
  - F3a: calcolo di *J*
    - *Teoricamente avrebbe complessità* N^3*, essendo moltiplicazione di due matrici di cardinalità NxN. Però J è data dal prodotto di una matrice diagonale per una matrice non sparsa. Abbiamo quindi tempo* N^2.
  - F3b: calcolo di *p*
    - *Analogamente al punto precedente, anziché avere tempo* N^2*, poiché prodotto di una matrice quadrata per un vettore, abbiamo tempo N poiché la matrice è diagonale.*
  - F3c: init di x_0. *Tempo N.*
- F_loop: ripetuto *K* volte, fino a che F5 non è soddisfatto.
  - F4: Calcolo di x_{i+i} a partire da x_i.
    - F4a: calcolo di $J*x_i$. *Tempo* N^2.
    - F4b: calcolo di x_i, sommando *p* al risultato parziale F4a. *Tempo N.*
  - F5: Verifica della condizione di terminazione. *Tempo N.*
    - Ovvero verificare se la differenza tra x_{i+1} ed x_i sia apprezzabile.
- F6: Output del risultato. *Tempo N.*
- F7: liberazione della memoria. *Tempo costante*

Quindi, per ricapitolare, quello che avremmo in questo approccio lineare è un tempo $\Theta(t_{F1} + N^2 + K \cdot N^2 + t_{F7}) = \Theta((K+1) \cdot N^2) = \Theta(K \cdot N^2)$ che è dominato da K (ovvero numero di iterazioni) ed N^2 ovvero la moltiplicazione di *J* per la *x* corrente.

Mattia Villani
502347
mattiavillani94@gmail.com

Verso il parallelismo

Possiamo iniziare a pensare di parallelizzare le operazioni che portano ad avere tempi alti. Dobbiamo innanzitutto chiarire però che *K* non è manipolabile, o meglio, *K* dipende dalla precisione che vogliamo dalla soluzione, ma non da l'algoritmo usato. Inoltre, *K*, rimane sempre come fattore, in quanto il calcolo di ogni x_i non è parallelizzabile con il precedente (di fatto il precedente è necessario per il suo calcolo). Questo ci porta a concludere che dobbiamo concentrarci nel cercare di ridurre il costo di una singola iterazione dell'algoritmo e non il costo dell'algoritmo in generale (che sarà dato da K per il costo della singola iterazione). Quindi dobbiamo cercare di parallelizzare la moltiplicazione matriciale. Più in particolare, a noi interessa parallelizzare il prodotto di una matrice di NxN per un vettore di dimensione N.
Supponiamo allora di introdurre un grado di parallelismo *w* e di distribuire in qualche maniera fair il carico della moltiplicazione. Il costo che otterremo per una singola iterazione sarebbe quindi $t = (N^2/w)$ e quindi in generale il costo totale sarebbe $t \approx (K \cdot N^2/w)$.
Potremmo sfruttare la parallelizzazione della moltiplicazione matriciale anche per calcolare la *J*. Ricordandoci però che introducendo una parallelizzazione, gli overhead potrebbero essere variabili in funzione di *w*, avremmo ora un tempo più simile alla seguente formulazione:

$$t \approx (t_{F1}(w) + N^2/w + K \cdot N^2/w + t_{F7}(w))$$

Come parallelizzare la moltiplicazione matriciale

Prendiamo in considerazione quindi la moltiplicazione di una matrice *A* di NxN con un vettore *v* di dimensione N. Consideriamo inoltre di avere a disposizione *w* workers.
Per paralizzare potremmo distribuire tra i vari workers delle task tipo il calcolo di A_i*v per un totale di N task. Ogni task ha un tempo di completamento quindi dell'ordine di N. Tuttavia, questo significa trasmettere ogni task singola ad un worker, e quindi avere dei costi di comunicazione proporzionali ad N.
In alternativa potremmo pensare di assegnare ad i workers direttamente un range di indici su cui operare (ovvero esplicitare quello che il parallel_for già fa) ed trasmettere ad i workers una quantità di task proporzionale ad w. Nel caso in cui i costi di trasmissione siano alti, questo ci potrebbe aiutare ad abbassare molto i tempi.
Per giocare con *Fastflow* il modello scelto è la farm.
Introduciamo quindi le entità *Collector* ed *Emitter*. Esse si occupano di distribuire ed accogliere i risultati delle task nei confronti dei workers. Il tempo di tale farm per un singolo ciclo del metodo di Jacobi sarebbe quindi

$$t_{farm} \approx t_c(w) + w \cdot t_e(w) + N^2/w$$

assumendo che t_c, t_e, siano rispettivamente i tempi del collector e del emitter (comprensivi dei tempi di trasmissione) ed assumendo che che il carico di lavoro sia fair.
In conclusione le nostre aspettative rispetto al tempo sono le seguenti:

$$t \approx (t_{F1}(w) + N^2/w + K \cdot (t_c(w) + w \cdot t_e(w) + N^2/w) + t_{F7}(w))$$

Facendo però attenzione a inizializzare la farm all'inizio del programma.

Mattia Villani
502347
mattiavillani94@gmail.com

Per semplificare il codice, ed assumendo *K* >> *w*, la fase di inizializzazione di *J*, ed *p* non è necessario sia parallela. Inoltre, essendo t_{F1}, t_{F7}, t_c ed t_e proporzionali ad *w*, se assumiamo di avere *K,N* >> *w*, si avrebbe tali tempi trascurabili, per cui, il tempo che ci si potrebbe aspettare è grosso modo quello indicato precedentemente, ovvero il seguente:

$$t \approx K \cdot N^2 / w$$

Parallelizzazione annidata

Potremmo anche giocare su come ottenere la parallelizzazione: fino ad ora abbiamo fatto riferimento a worker di una farm, ma potremmo provare anche a concedere ad ogni worker di parallelizzare il suo lavoro su *p* threads. Questo si potrebbe ottenere mescolando una farm in cui ogni worker usa un parallel for. In questo caso, *w* sarebbe definito come *numero workers * grado di parallelismo interno*.

Plain threads

Come alternativa alla implementazione tramite Fastflow di una farm, è proposta una versione con plain threads realizzati con i *pthread*s. In questo caso, i worker sarebbero i thread, però non è prevista la presenza di un emitter e di un collector. La sincronizzazione potrebbe esser fatta tramite un contatore che invita i threads a eseguire la moltiplicazione ed un altro contatore che “fa l’appello” dei thread che hanno terminato. Le valutazioni sul tempo rimangono invariate.

Implementazione

La versione lineare è triviale e non verrà discussa.

Plain threads

La versione con plain threads è realizzata con i pthreads, e si basa sulla costruzione di un array di task (con annessi due locks e due condition variables) che indica che lavoro deve esser eseguito. Ogni task è assegnata ad esattamente un thread (che a sua volta ha assegnata una sola task). Un worker si mette in attesa su un contatore (tipo il contatore delle iterazioni del metodo di jacobi), quando il valore cambia, allora il worker esegue la sua task, incrementa un contatore delle task ultimate (esegue una signal se è l’ultimo) e si rimette in wait. Il thread principale si occupa di aspettare che il contatore delle task ultimate sia w. Quando lo è, verifica le condizioni di termine, e se non sono verificate, allora incrementa il contatore delle iterazioni (ed invia una signal in broadcast), dopo di che si rimette in wait sul contatore delle task ultimate. In sostanza il thread principale, esegue solo un controllo di costo N ad ogni iterazione. I thread eseguono attesa passiva.

Fastflow farm

La versione con fast flow è realizzata tramite una semplice farm in cui l’emitter ed il collector sono stati rimpiazzati da due stage custom. La farm è “wrapped_around” così il collector, ogni *w* messaggi ricevuti, invia un messaggio all’emitter (o un EOS se la condizione di fine metodo è stata raggiunta). L’emitter invece invia referenze di celle di un array

Mattia Villani
502347
mattiavillani94@gmail.com

(precedentemente inizializzato) che rappresenta il lavoro da svolgere (in sostanza, la coppia di indici (inizio e fine) che indicano le righe da moltiplicare), più altri dati necessari alla computazione. Di conseguenza, ad ogni ciclo del metodo di Jacobi, non sono effettuate allocazioni. L'emitter si limita a passare tutte le referenze dell'array delle task ai worker (costo w), mentre il collector, si limita a contare le referenze ricevute. Quando sono w, verifica le condizioni di termine (costo n).

Guida d'uso

Come indicato dall'help del programma, le opzioni sono le seguenti :

```
Usage : ./jacobiff [
                  | -w val        num of workers
                  | -t val        internal paralelism for farm
                  | -n val        size of the problem to generate
                  | -l            (run the linear version)
                  | -p            (run the plain thread version)
                  | -f            (run the farmed version FASTFLOW)
                  | -m val        max iteration ...
                  | -e valf       error tolareted ...
                  | -s            silent the output
                  | -r            do srand on time(NULL)
                  | -h            print this message and exit
        ]
        [input_file]    (In this case -n can't be used)
        Where input_file has the following format:
                N
                a11  a12  ... a1N
                ...  ...  ... ...
                aN1  aN2  ... aNN
                b1   b2   ... bN
```

Ovvero, sono previste due modalità di "input" della matrice :

1) da file
2) generarne uno in modo aleatorio (con A a predominanza diagonale per garantire la convergenza)

Durante il testing, essendo la soluzione in se secondaria rispetto al tempo di completamento, il metodo utilizzato è il 2.
Sempre nella filosofia del non dare importanza al input, è possibile silenziare la stampa di alcune info con *-s*.
K è determinato dai parametri *-m*, ed *-e* che sono rispettivamente il numero massimo di iterazioni e la tolleranza tra due *x* consecutive per essere considerate uguali.
I parametri *-w* ed *-t* sono rispettivamente il numero di workers per la farm (o di threads per la versione in plain threads) ed il grado di parallelismo intero di ogni worker (solo per la versione farm).
Mentre i parametri *-l*, *-p*, *-f* indicano che implementazione usare, rispettivamente *linear, parallel,* ed *farmed.* Nel caso siano dati come argomento, più di un metodo, ciascuno di essi verrà usato (nell'ordine lpf).

Mattia Villani
502347
mattiavillani94@gmail.com

E’ possibile modificare gli output con dei flag a compile time:

-DTRACE_FASTFLOW stampa le stat di fastflow (solo per farm)
-DNO_RESULT silenzia la stampa del risultato
-DNO_TIMES silenzia la stampa dei tempi

Ricapitolando quindi, un esempio di comando è il seguente:
./jacobiff -f -m1000 -n5000 -w2 -t4
Che produce il seguente output (con la opzione -DNO_RESULT)

```
Problem_creation(s):0.59375
RUNNING:
        n 5000
        w 2
        t 4
        m 1000
        e -1
        r 0
        linear 0
        plain thread 0
        farm 1
        ff's MAX_NUMTHREADS 512

farmed
        Initialization(s):0.125
        Iterations:1000
        Loop(s):81.5469
        Cleanning(s):0.015625
        Enlapsed time(s):81.6875
```

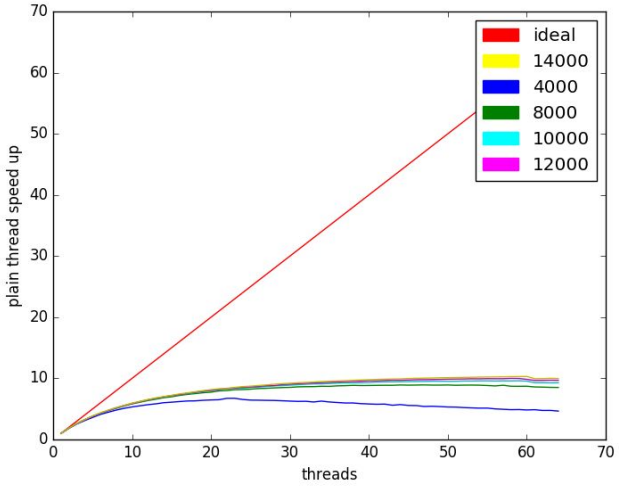
Risultati

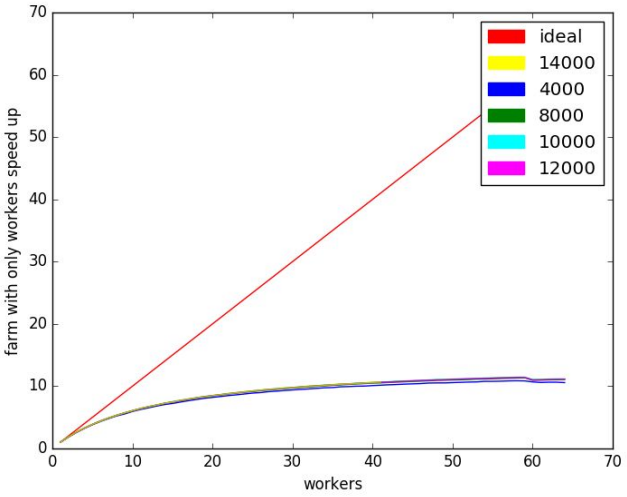
I risultati verranno presentati per la versione realizzata in plain thread, e per la farm (con i casi specifici in cui il grado di parallelismo di ogni worker sia 1, 2, 4, 8).

Speed up

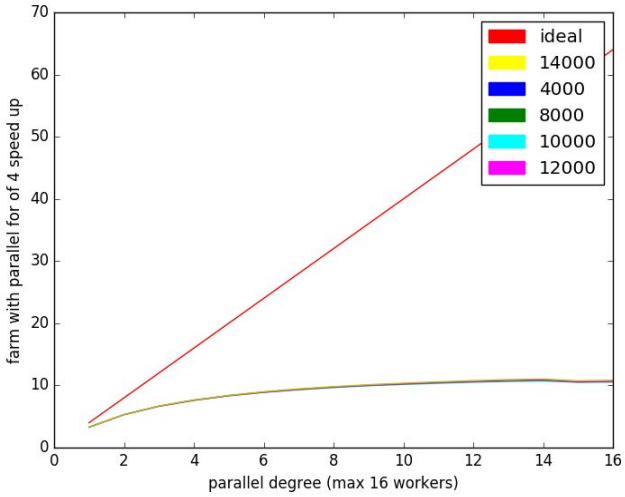
Speed up	
speed up per la versione plain thread	speed up per la versione farm con solo workers

Mattia Villani
502347
mattiavillani94@gmail.com

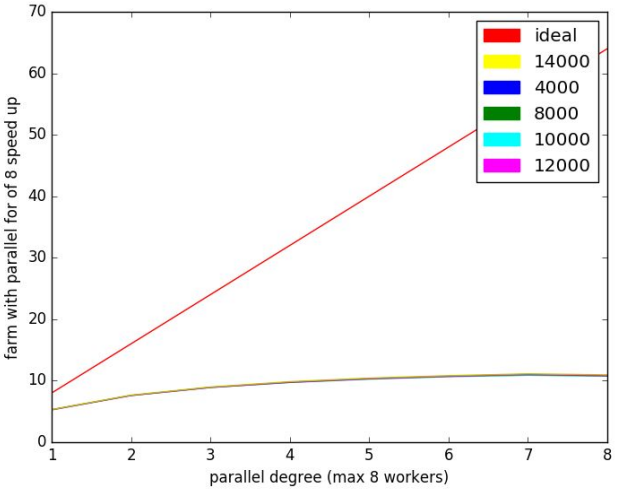




speed up per farm con grado di parallelismo 4 per ogni fam



speed up per farm con grado di parallelismo 8 per ogni worker

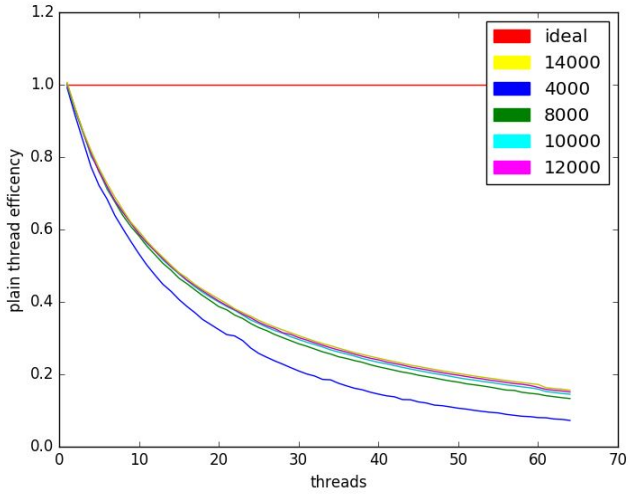


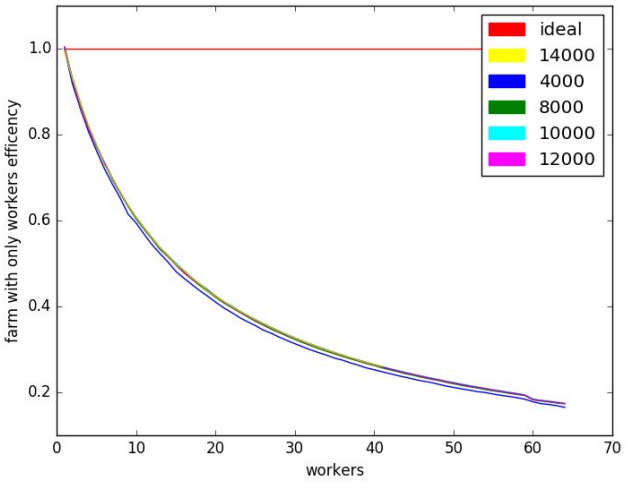
Efficency

Efficiency per verione plain thread

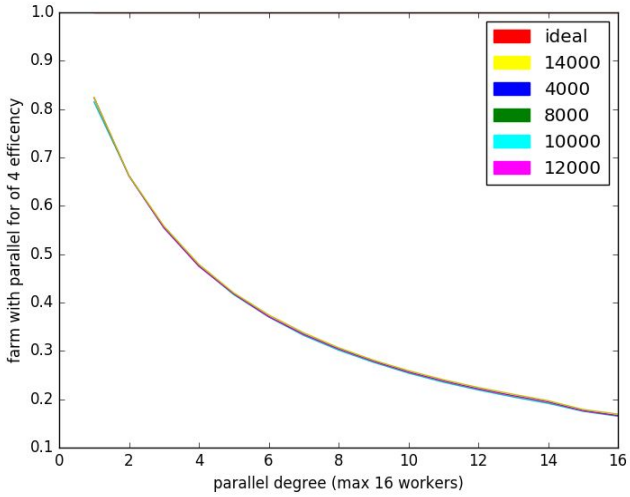
Efficiency per versione farm con solo workers

Mattia Villani
502347
mattiavillani94@gmail.com

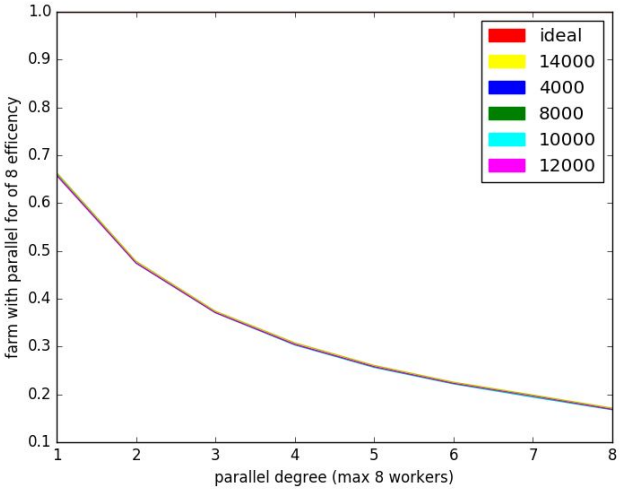




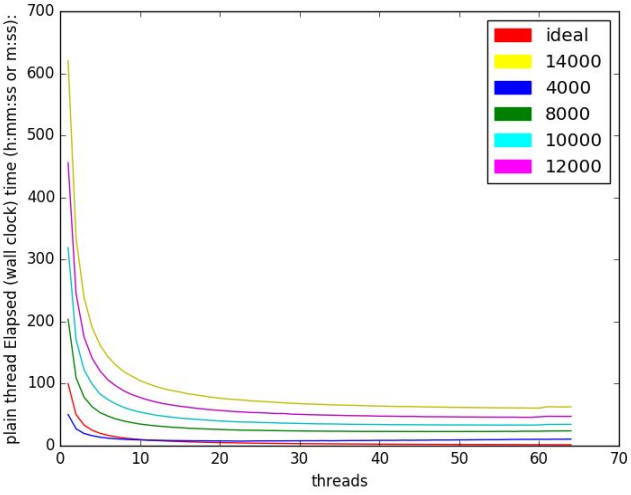
Efficiency per versione farm con g.p. 4 per worker

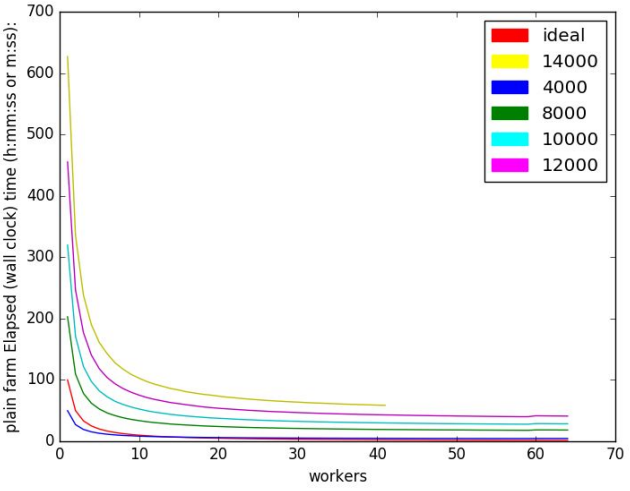


Efficiency per versione farm con g.p. 8 per worker

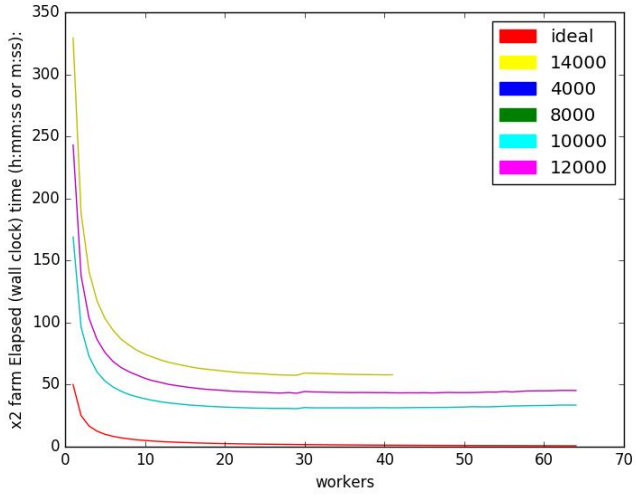


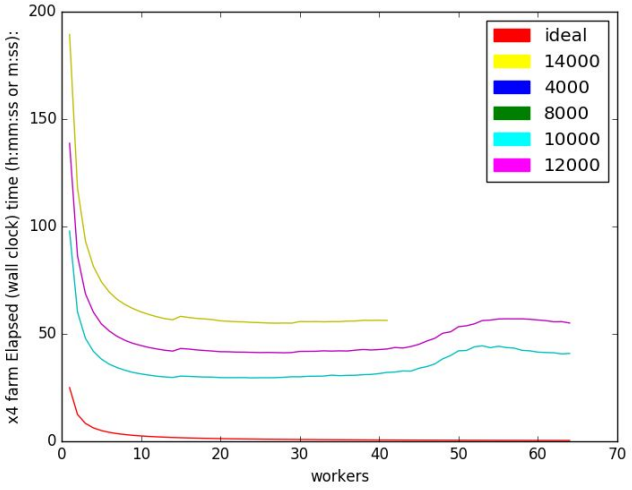
Tempi di completamento (con riferimento ideale per problema con n = 10000 (ed un fattore di 1/1000000))

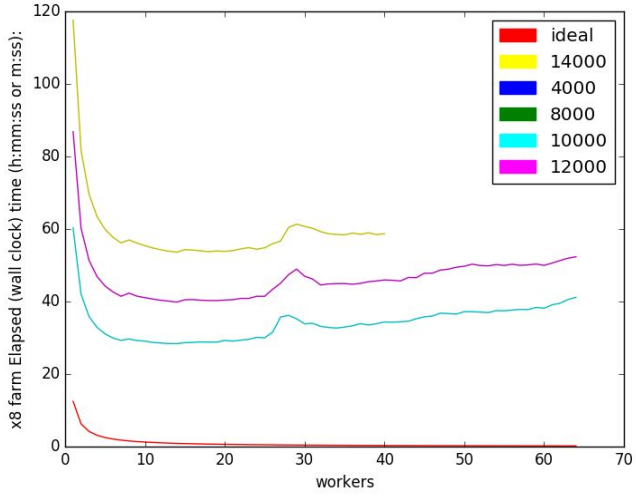




Mattia Villani
502347
mattiavillani94@gmail.com







Come vediamo dai risultati, quando il grado di parallelismo cresce troppo, come nel caso degli ultimi due grafici (il cui grado di parallelismo arriva ad essere 64 x 8) le prestazioni deteriorano. Vediamo che, come ci si potrebbe aspettare, le varie implementazioni scalano bene fino a che il grado di parallelismo richiesto è minore di circa 60-64.