-MEINF-
Computació d'Altes Prestacions

Programació amb Pthreads

Mariano Trebino
u1910541

Index

1. Introducció

Per realitzar aquesta pràctica he decidit fer el programa que normalitza un vector. Sincerament els dos programes son molt semblants i he escollit aquest com podria haver escollit l'altre.

En quant a la implementació del programa, he de dir que m'ha costat un mica ja que fa un parells d'anys que no treballo amb C/C++ i m'ha sigut difícil tornar a treballar amb punters, posicions de memòria, etc... No obstant, pel que fa als *PThreads*, és força fàcil d'entendre i de fer servir. A més a més, la documentació està bé i en Internet està ple d'exemples.

2. El programa

El programa donat dos paràmetres d'entrada: mida i nombre de *PThreads*, genera un vector aleatori de N posicions de nombres en coma flotant. El rang dels nombres aleatoris va des de 0 fins N i específicament només n'hi ha un nombre aleatori per cada interval de mida 1 des de 0 fins N, és a dir, només n'hi ha un nombre entre 0 i 1, entre 1 i 2, etc.
Un cop generat aquest vector aleatori executem la funció especifica: seqüencial o amb *PThreads* passant-li el vector i el nombre d'elements.

2.1 Seqüencial

El programa que s'executa de manera seqüencial no te cap secret, ja que és pràcticament idèntic al codi que vam plantejar en classe (veure Fig. 1).

```
int norm_vec(int *x, int n)
{
    int i;
    float modul;

    modul
    for(i=0;i<n;i++)
    {    modul =++ x[i]*x[i];
    }
    modul = sqrt(modul)
    for(i=0;i<n;i++)
    {    x[i] = x[i]/modul
    }
}
```

Fig. 1 : Codi seqüencial de l'algoritme de normalització d'un vector

2.2 PThreads

Pel contrari, el programa implementat amb *PThreads* és menys intuïtiu i per aquesta raó crec que val la pena analitzar-lo pas a pas..

A alt nivell podem veure diferents maneres de dividir l'algoritme en tasques més petites i independents que es puguin paral·lelitzar. Jo, personalment he dividit el programa en aquestes sis parts (veure Fig. 2):

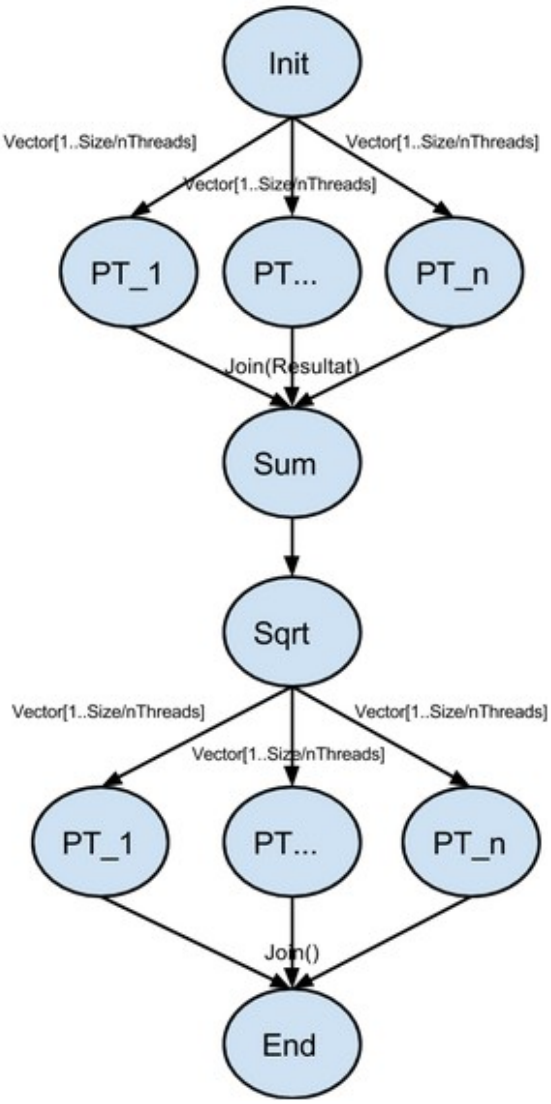

Fig. 2: Etapes de paral·lelització

1. Primer de tot, dividim el vector en diferents parts (si pot ser de la mateixa mida)
2. Per a cada *PThread* generem una sub-divisió del vector
    1. Calculem la suma dels quadrats de la sub-divisió
    2. Retornem el resultat
3. Obtenim tots els resultats i els sumem
4. Obtenim el modul del vector aplicant una arrel quadrada al resultat anterior
5. Tornem a dividir el vector en diferents parts
6. Per a cada *PThread* generem una sub-divisió del vector
    1. Calculem la divisió entre cada component i el modul
    2. Guardem la nova component normalitzada en el mateix vector

No considero rellevants de cares al informe els detalls d'implementació de cadascun d'aquests passos ja que això és pot veure al codi font on és molt fàcil d'entendre ja que cada funció és auto-explicativa.

3. Els resultats

Per tal de fer comparacions de rendiment i analitzar els resultats he tingut que implementar uns temporitzadors al meu codi per mesurar els temps d'execució de les funcions.

Les proves que he fet amb el meu portàtil han estat una mica limitades pel hardware, però crec que son prou representatives per veure el *speedup* entre l'execució seqüencial i paral·lela i entendre com varien el temps quan augmentem el nombre de *Pthreads* i la mida del vector.

Llavors per realitzar aquestes proves he fet un script que executes el programa variant la mida del vector i el nombre de *PThreads* de manera exponencial i mesures els temps que trigava a fer-ho.
Per veure les dades completes obrir el fitxer: *annex_taules_i_grafics*.

3.1 Execució seqüencial

Si representem a l'eix de les 'X' la mida del vector i a les 'Y' el temps en mili-segons que ha trigat en executar-se, ambdós en escala logarítmica, podem veure (mirar Fig. 3), que pràcticament podem representar la execució en seqüencial amb una línia recta amb una inclinació de 45º.

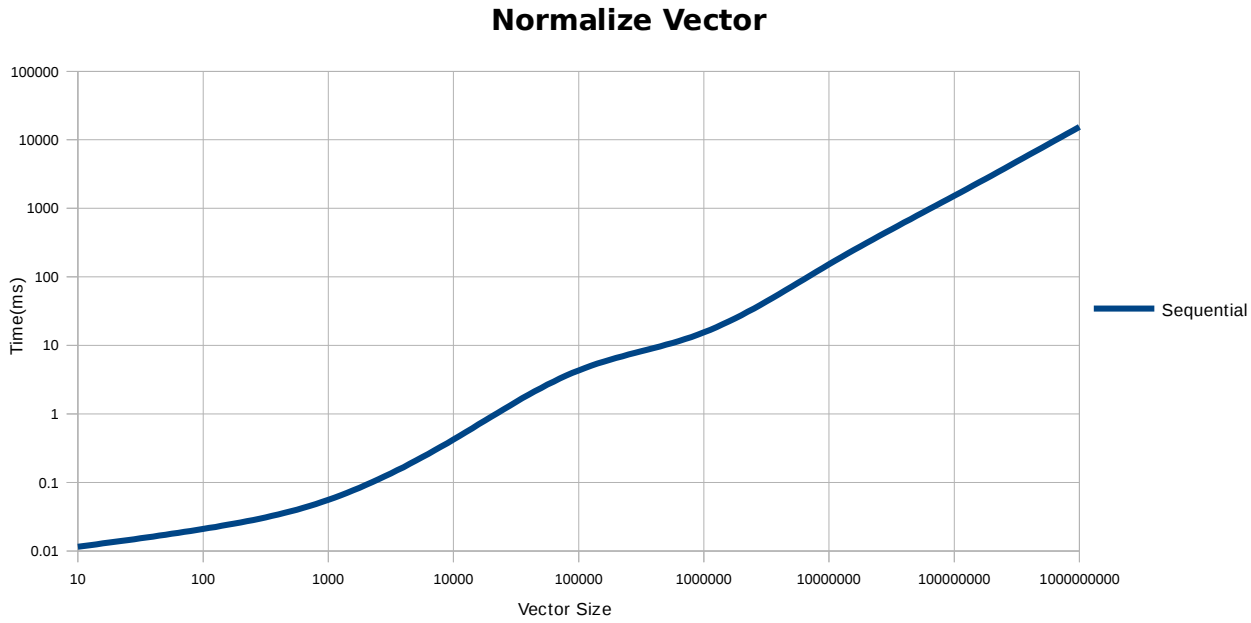

Fig. 3: Execució seqüencial

D'altra banda, també podem observar que quan la mida del vector és molt petita, el programa és molt ràpid, però a mesura que augmentem

exponencialment el nombre d'elements del vector el temps també és veu incrementat de la mateixa manera, per tant l'algorisme és O(n) respecte el nombre d'elements del vector.

3.2 Execució amb 1 PThread

Encara que sembla que no té molt de sentit aquesta execució crec que és força important analitzar-la. En aquest cas, estem utilitzant l'algoritme pensat per executar-se amb diferents *PThreads*, amb només un.

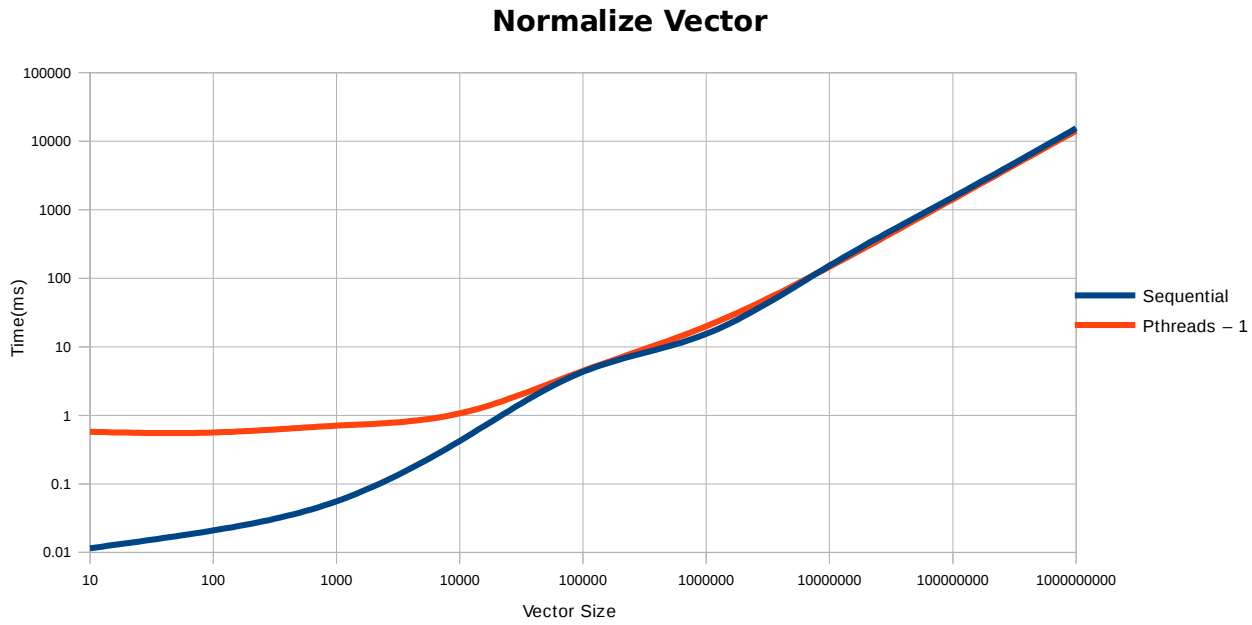

Fig. 4: Execució Seqüencial i amb 1 PThread

Veiem (veure Fig. 4) que quan el vector és molt petit triga molt més que en el cas seqüencial, i a mesura que augmentem la mida del vector escala pràcticament de manera constant fins que el vector és de mida mitjana i llavors comença a escalar linealment de la mateixa manera que en el cas seqüencial.

Aquest curiós comportament és degut al *overhead* provocat per la creació dels *threads*, els *join*, la divisió de les tasques, etc... Que encara que en aquest cas siguin inútils perquè només utilitzem un *PThread*, s'han de fer igual.
Així doncs, quan la mida del vector és molt petita, el temps de *overhead* es pot ignorar ja que el programa triga poc a executar-se. Però quan la mida del vector augmenta molt, aquest *overhead* és totalment negligible ja que el temps que triga el programa en total és molt gran en relació amb el temps de *overhead*.

3.3 Execució amb 2 PThread

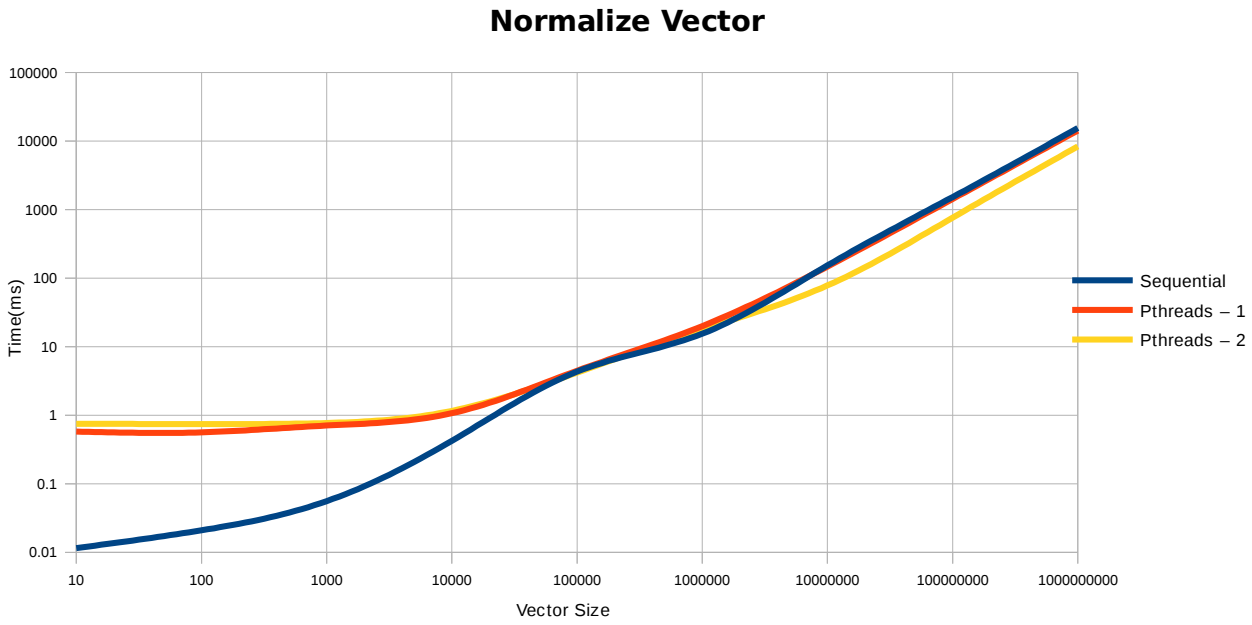

Fig. 5 : Execució Seqüencial, amb 1 i 2 PThreads

Com veiem un altre cop al gràfic (veure Fig. 5) al augmentar el nombre de *PThreads*, el *overhead* també augmenta una mica, ja que ha de fer més operacions de coordinació.
No obstant, quan el nombre d'elements augmenta molt podem veure com el *overhead és compensat amb una reducció del temps d'execució* graciés al augment de *PThreads* reduint d'aquesta manera el temps total d'execució.

3.4 Execució amb 4 PThreads

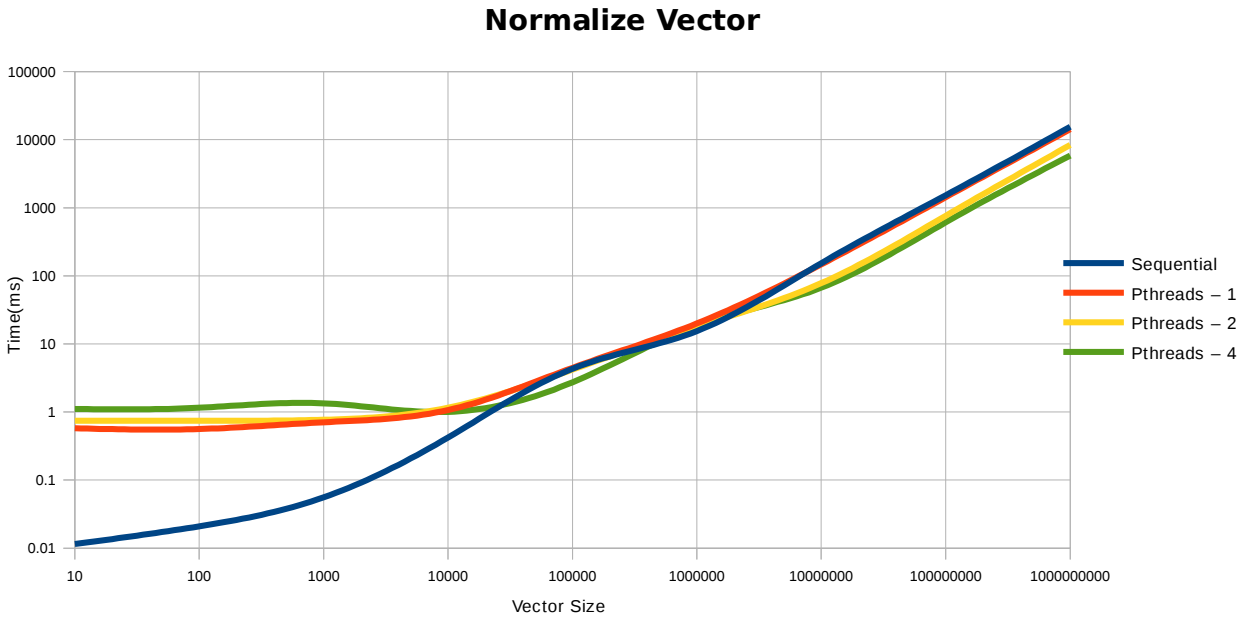

Fig. 6: Execució seqüencial amb 1, 2 i 4 PThreads

Llavors, al augmentar encara més el nombre de *PThreads* podem apreciar (veure Fig. 6) com el mateix fenomen que abans és repeteix però amb més intensitat de manera que quan hi ha pocs elements el programa tarda més, degut a que estem incrementant les operacions de coordinació i per tant el *overhead* , però quan la mida del vector és considerable, aquest *overhead* ja és negligible i comença a valdre la pena utilitzar diferents *PThreads* per executar el paral·lel i aconseguir un *speedup* més gran.

3.5 Execució amb 8 PThreads

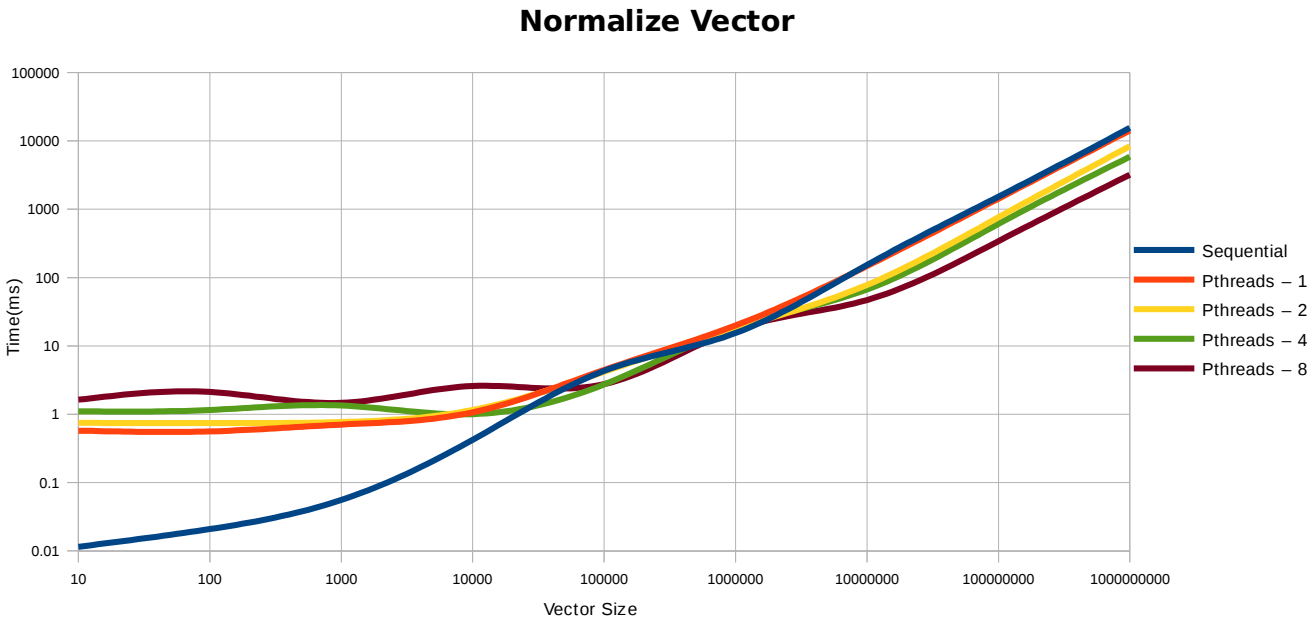

Fig. 7: Execució seqüencial i amb 1, 2, 4 i 8 PThreads

Un altre cop (veure Fig. 7), el mateix fenomen és fa present amb més intensitat, augmentant més encara el temps que triga amb un vector petit però reduint el que triga amb un vector molt gran.

3.6 Execució amb 16 PThreads

Aquest últim gràfic (veure Fig. 8) el considero molt important ja que fa present un fet que fins ara no havíem tingut en compte.

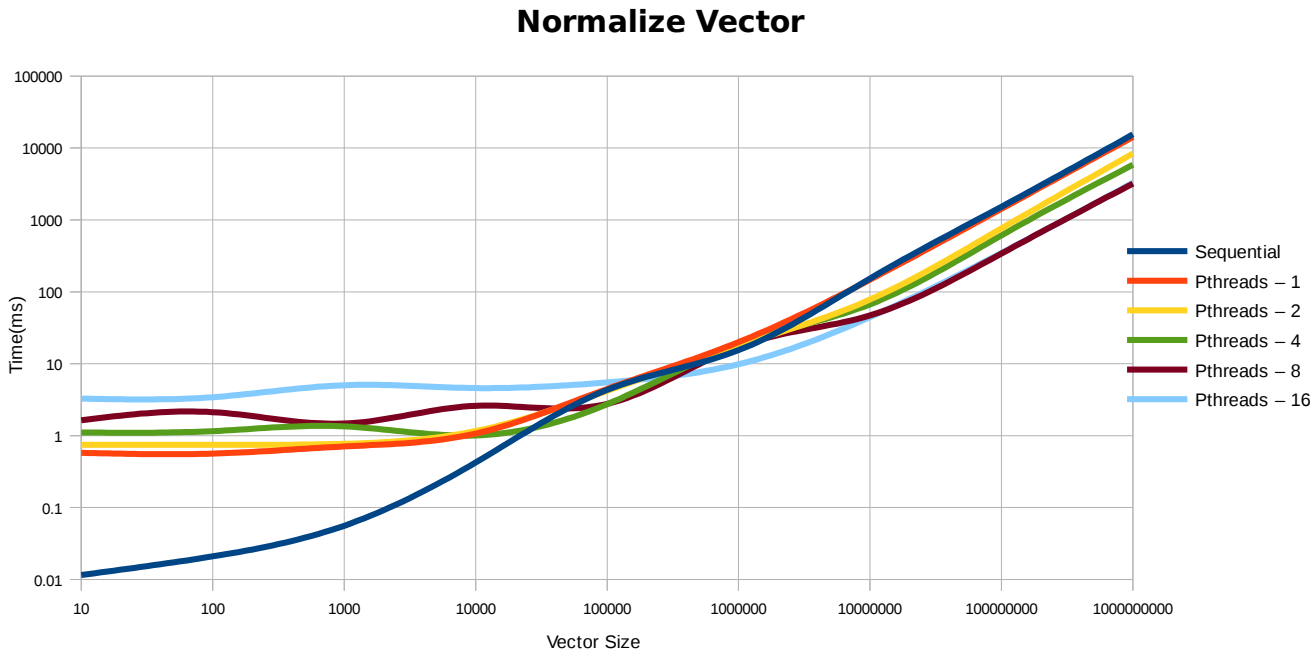

Fig. 8 : Execució seüencial i amb 1, 2, 4, 8 i 16 PThreads

Si ens fixem al gràfic, al igual que en els altres casos, quan el vector es petit hi ha un increment del temps, com sempre degut al *overhead* , però a diferencia que abans, quan el vector és molt gran ja no hi ha un increment en el rendiment. Triga pràcticament el mateix que utilitzant 8 *PThreads*. Això és degut a que no podem explotar més el paral·lelisme amb aquest nombre d'elements i que per més *PThreads* que afegim, el rendiment continuara estancat en aquesta línia, és més, probablement comenci a decaure degut a que *overhead* pot arribar a ser molt gran.

3.7 Molts més PThreads: 32, 64, 128, 256 i 512

Tal i com comentàvem abans, veiem a la Fig. 9 que cada cop més hi ha un *overhead* més gran i no aconseguim reduir el temps d'execució amb vectors molt grans.

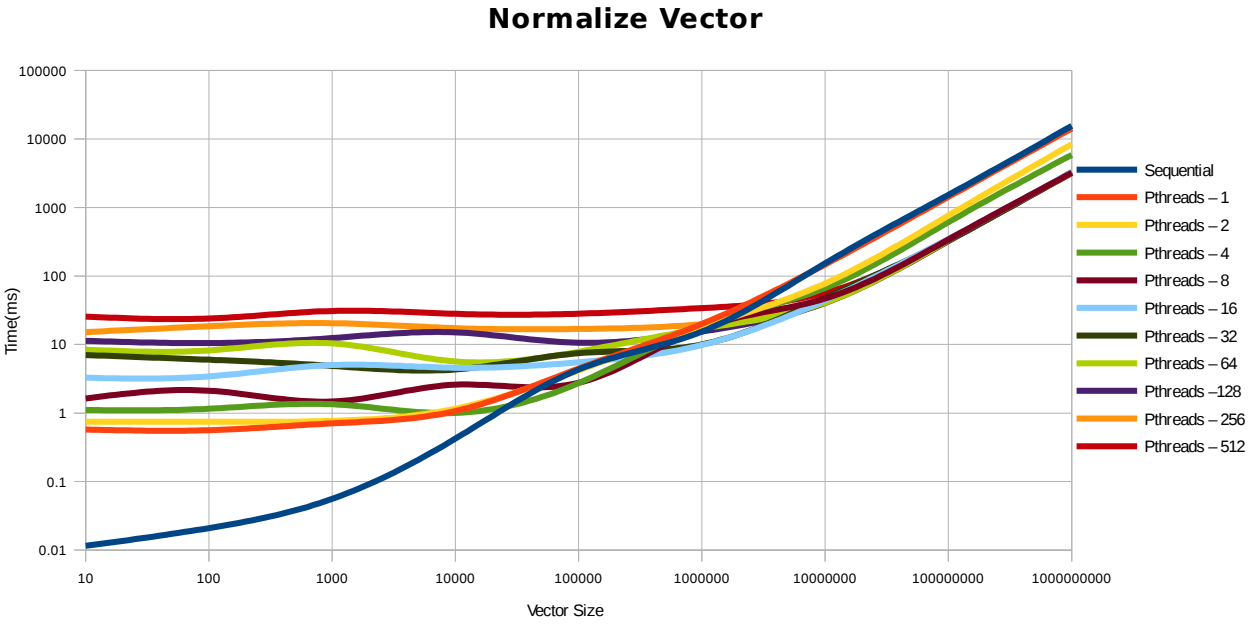

Fig. 9 : Comparativa d'execució en seqüencial i variant el nombre de PThreads

No obstant, el més probable és que per vectors de dimensions encara més grans (al voltant de 100 billons d'elements o més) el temps si que s'aniria reduint a mesura que augmentéssim el nombre de *PThreads*. Llavors arribaríem a un punt que passaria el mateix que aquí i hauríem d'utilitzar vectors encara més grans.

4. Conclusions

En primer lloc crec que és important destacar la dificultat afegida que suposa implementar un algoritme amb *PThreads*, no per la dificultat d'aquesta llibreria si no pel canvi de paradigma que suposa dissenyar un programa que s'executara de manera concurrent on hi hauran segments de memòria compartida, operacions que s'executen a la vegada, etc...

En segon lloc, pel que fa al algoritme, destacar que el sequencial es d'ordre O(n) ja que ha de recorrer tot el vector (de fet, el recorre dues vegades). I per l'altra banda, el algoritme paral·lel, que potser fa més recorreguts per coordinar els *PThreads* (*overhead*), finalment redueix en la majoria de casos el temps d'execució graciés a que a la mateixa vegada estem executant les mateixes operacions sobre diferents parts del vector.

Per últim, m'agradaria tornar a parlar sobre els gràfics: recordar com el primer gràfic que feia referencia a l'execució en seqüencial semblava una línia força equilibrada entre els dos eixos i d'altra banda comentar la execució paral·lela, la qual, al augmentar el nombre de *PThreads*, provocava un *overhead* més gran a cost d'una paral·lelització en les tasques. Dita paralel·lització, és la que ens proporciona un major rendiment, però a partir d'una certa mida, ja que quan el vector és molt petit, tardem més temps fent tasques de coordinació que no pas executa'n codi per normalitzar el vector.

No obstant, aquesta paralel·lització té un limit. En el meu cas ha sigut 16 *PThreads* amb un vector 10^9 elements. Aquest limit ve donat a que hi ha un punt en el qual no podem distribuir més les tasques, ja que estan paral·lelitzades al seu punt màxim i optim, i al afegir més *PThreads* sense incrementar el nombre d'elements, simplement estem incrementant el temps de coordinació sense afegir un increment en la paral·lelització de les tasques i per tant perdent rendiment.

Llavors, crec que puc dir amb certesa que no hi ha una combinació guanyadora. Depenen de la mida del vector que facis servir una combinació serà millor que una altra. No obstant, normalment sempre estem treballant amb vectors d'una mida fixa i el que ens sol interessar és que hi hagi una millora de rendiment important quan el vector és gran i aixo ho podem aconseguir gracies als *PThreads*. Però, hem d'estar alerta i anar amb cura, fer proves i veure quin és el nombre optim de *PThreads* per la nostra mida de vector per tal d'augmentar al màxim el nombre rendiment.