Παράλληλα & Κατανεμημένα Συστήματα Υπολογιστών

Εργασία 1

Κατασκευή οκταδικού δέντρου για την ιεραρχική ομαδοποίηση Ν σωματιδίων στον τρισδιάστατο χώρο. Η παραλληλοποίηση αυτή παρουσιάζεται 3 διαφορετικές υλοποιήσεις με διαφορετικά εργαλεία η κάθε μία :
Cilk, Openmp, Pthreads.

Κασπαρίδης Παναγιώτης, ΑΕΜ:8157

Περιγραφή μεθόδων παραλληλισμού

Υλοποίηση Pthread

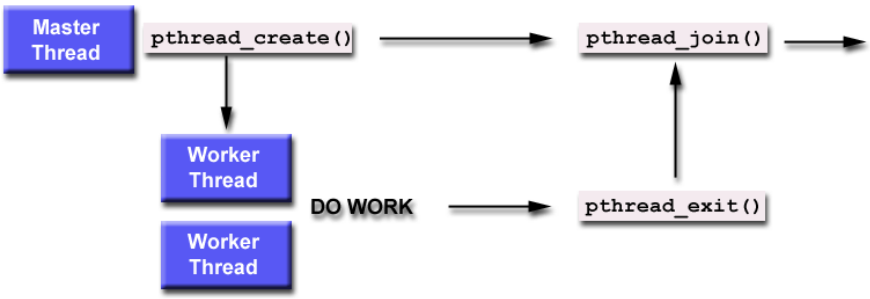
- **test_octree:** Εδώ απλά ζητάω ένα παραπάνω όρισμα για να τρέξει ο κώδικάς μου που είναι ο αριθμός των thread και τον αποθηκεύω σε μια global μεταβλητή μέσα στο header utils.h με όνομα num_threads. Το κάνω αυτό για να μη πειράξω καθόλου τα ορίσματα των συναρτήσεων και απλά με την προσθήκη του header να έχω πρόσβαση στον αριθμό των threads. Μια ακόμη διαφορά είναι ότι καλώ την first_call_truncated_radix_sort αντί για call_truncated_radix_sort η οποία παίρνει ακριβώς τα ίδια ορίσματα και γενικά θα εξηγηθεί παρακάτω γιατί υπάρχει.

- **hash_codes:** Τι περιέχει : Ένα typedef struct:hash_struct , και τις συναρτήσεις που υπήρχαν,+ την προσθήκη της void *thread_quantize_function(void *arg), η οποία παίζει το ρόλο της παλιάς quantize απλά είναι απαραίτητη για την παράλληλη υλοποίηση μου.
Διαφορές: Στην quantize έχω ορίσει τα arguments που θα πάρει η thread_quantize_function καθώς και δημιουργώ και τα threads . Μετά ελέγχω τόσο την pthread_create όσο και την pthread_join να έχουν σωστό return οι συναρτήσεις αυτές.
Εδώ φαίνεται η αρχικοποίηση των arg τα οποία είναι hash_struct και είναι μεγέθους num_threads όμοια τα thread τύπου pthread_t *.
Η thread_quantize_function υλοποιεί την απλή quantize παράλληλα. Το N= N/num_threads ίδιο για όλα τα νήματα και οι πράξεις μέσα ξεκινάνε από κατάλληλες θέσεις ανάλογα το start που έχει πάρει.
Κομμάτια κώδικα του είναι : (δεν συμπεριέλαβα όλο τον κώδικα γι' αυτό και έβαλα « ... »)

```
for (int i=0;i<num_threads;i++){
arg[i].threadID=i;
arg[i].codes =codes;
arg[i].X=X;
arg[i].low=low;
arg[i].step=step;
arg[i].N=N/num_threads;
arg[i].start=i*(N/num_threads)*DIM;
flag=pthread_create(&thread[i],NULL,thread_quantize_function,(void *)&arg[i]);
if(flag){
printf("Error: pthread_create returned code: %d\n", flag);
        return;
        }
}
```

```
hash_struct *data=(hash_struct *)arg;
…
data->codes[data->start+i*DIM + j] = compute_code(data->X[data->start+i*DIM + j], data->low[j], data->step);
```

Στο τέλος περιμένω να τελειώσουν τα thread με join για να πάω στην επόμενη συνάρτηση. Τα thread κάνουν exit στην thread_quantize_function μόλις τελειώσουν την δουλειά που έχουν να κάνουν.
Η λογική μου είναι σαν του διαγράμματος



- **morton_encoding:** Κι εδώ ακολουθώ ακριβώς την ίδια λογική. Τα έξτρα κομμάτια είναι ένα typedef struct morton_struct και η void *thread_morton_function(void *a).
Διαφορές ακριβώς όπως και πριν στην morton_encoding. Ορίζω τους πίνακές μου για τα morton_struct και thread για μέγεθος num_threads(το πλήθος των thread που έχω ορίσει να τρέχει στα ορίσματα της main). Κι εδώ βάζω το κάθε thread να εκτελεί N/num_threads επαναλήψεις και του δίνω ένα start. Όταν τελειώσουν τις επαναλήψεις τους κάνουν exit και τα περιμένω να τελείωσουν με join στην morton_encoding. Ο κώδικας είναι ακριβώς ίδιας λογικής με την hash_codes που έχω εξηγήσει μερικά κομμάτια πιο πάνω, δεν έχει άλλη δυσκολία.
- **Radix_sort:** Η radix_sort ήταν λίγο πιο δύσκολη σαν υλοποίηση σε σχέση με τις προηγούμενες γιατί εδώ δεν μπορούσα να κάνω παράλληλες τις 2 μεγάλες for που είχε για τον καθορισμό του BinSizes και BinCursor. Η συνάρτηση αυτή γινόταν παράλληλη στο recursive κομμάτι της. Στην ουσία άνοιγε σε ένα οκταδικό δέντρο. Αυτό το κομμάτι παραλληλοποίησα.
Τι πρόσθεσα : μια global μεταβλητή την available_threads η οποία χρειάζεται για να ξέρω αν μου μένουν threads όταν είμαι βαθιά στο δέντρο, ένα pthread_mutex_t πάλι global, μια void

first_call_truncated_radix_sort με ακριβώς τα ίδια ορίσματα με την truncated_radix_sort και η δουλειά της στην ουσία ήταν να δίνει τιμή στην available_threads ίση με τον αριθμό των threads(num_threads) και αρχηκοποιεί το mutex. Έπειτα καλούσε την truncated_radix_sort.
Χρειάστηκα πάλι ένα typedef struct radixStruct και μια void *thread_function(void *a) η οποία στην ουσία απλα καλούσε την truncated_radix_sort με τα κατάλληλα ορίσματα που είχε το κάθε thread και στο τέλος της κάνει exit το thread. Η δομή radix_struct έχει σαν μεταβλητές τα ορίσματα της truncated_radix_sort + το offset το οποίο χρησιμοποιείται για την αναδρομή.

Αλλαγές στην truncated_radix_sort : Το κομμάτι που άλλαξα είναι αυτό της αναδρομής (μετά δηλαδή από το σχόλιο που υπήρχε /* Call the function recursively to split the lower levels */). Εδώ έχω την λογική: αρχικά αν έχω available threads ή μου τελείωσαν. Αυτό υπάρχει ώστε σε περίπτωση που έχουν τελειώσει να συνεχίσει σειριακά παρακάμπτοντας το lock mutex που υπάρχει μετά. Στην περίπτωση που έχω threads available διακρίνω 3 υποπεριπτώσεις. Πλέον η κάθε μια είναι critical γιατί θα μεταβάλει τον αριθμό των available threads. Αυτές είναι:

- Να έχω τουλάχιστον 8 threads available όποτε μειώνω την μεταβλητή available_threads κατά 8 επιστρέφω το mutex(unlock) και πλέον αρχικοποιώ τον πίνακα radixStruct και δημιουργώ τα pthread_t. Να αναφέρω εδώ ότι φτιάχνω πίνακες μεγέθους 8 για το καθένα γιατί τόσες θα είναι οι παράλληλες αναδρομές.
  Εδώ φαίνεται πώς βάζω τιμές για available_threads>=8(MAXBINS) όμοια είναι και για την άλλη περίπτωση.

```
for(int i=0; i<MAXBINS; i++){
  int offset = (i>0) ? BinSizes[i-1] : 0;
  int size = BinSizes[i] - offset;
  //available_threads--;
  arg[i].morton_codes=morton_codes;
  arg[i].sorted_morton_codes=sorted_morton_codes;
  arg[i].permutation_vector=permutation_vector;
  arg[i].index=index;
  arg[i].level_record=level_record;
  arg[i].N=size;
  arg[i].population_threshold=population_threshold;
  arg[i].sft=sft-3;
  arg[i].lv=lv+1;
  arg[i].offset=offset;
 flag=pthread_create(&thread[i],NULL,thread_function,(void *)(&arg[i]));
```

- Να έχω λιγότερα από 8 threads αλλά τουλάχιστον 1. Εδώ ακολουθώ την ίδια λογική με πιο πάνω. Αρχικά μηδενίζω τα available threads,επιστρέφω το mutex και δημιουργώ πίνακα με τα arg και thread για όσα available threads είχα και τους αρχικοποιώ. Η μόνη διαφορά με την προηγούμενη περίπτωση είναι ότι εδώ οι επαναλήψεις είναι μέχρι εκείνα τα available threads που είχα. Έπειτα συνεχίζει και κάνει τις υπόλοιπες αναδρομές το ήδη υπάρχον thread και μετά περιμένει τα υπόλοιπα να τελειώσουν.
- Αν τα available threads είναι 0 τότε επιστρέφει το mutex και εκτελεί την αναδρομή σειριακά. Αυτή η περίπτωση υπάρχει γιατί μπορεί κάποιο thread να "δει" available_threads να προχωρήσει στην παράλληλη αναδρομή αλλά όταν πάρει το κλειδί(κάνει lock το mutex) να έχουν τελικά τελειώσει τα threads.

◦ Αρχικά πριν το mutex είπα ότι ελέγχω πάλι τα available threads αυτό συμβαίνει γιατί θέλω όταν θα φτάσω στα 0 threads να μη περνάω πάλι απ' την διαδικασία με τα mutex και να συνεχίζει σειριακά. Οπότε όταν δεν έχω available_threads η συνάρτηση συνεχίζει σειριακά και αποφεύγει τα κλειδιά.

Στο κώδικά μου δεν αυξάνω τα available_threads όταν ένα νήμα τελειώσει την δουλειά του κι αυτό γιατί όταν το έκανα παρατήρησα χειρότερους χρόνους. Μπορεί λογικά να φταίει και η υλοποίηση μου.
Επειδή ο κώδικας είναι μεγάλος και θα πάρει αρκετό χώρο να τον παραθέσω εδώ θα πρότεινα να δείτε στο αρχείο radix_sort.c, το κομμάτι της αναδρομής.

- **data_rearrangement:** Κι εδώ ακολουθώ ακριβώς την ίδια λογική με τις hash_codes & morton_encdoing. Τα έξτρα κομμάτια είναι ένα typedef struct data_rearrangement_struct και η void *thread_rearrangement(void *a).
  Διαφορές ακριβώς όπως και πριν στην data_rearrangement ορίζω τους πίνακές μου για τα data_rearrangement_struct και thread για μέγεθος num_threads. Κι εδώ βάζω το κάθε thread να εκτελεί N/num_threads επαναλήψεις και του δίνω ένα start. Όταν τελειώσουν τις επαναλήψεις τους, κάνουν exit

και τα περιμένω να τελειώσουν με join στην data_rearrangement. Ο κώδικας είναι ακριβώς ίδιας λογικής με την hash_codes,morton_encoding που έχω εξηγήσει μερικά κομμάτια πιο πάνω δεν έχει άλλη δυσκολία.

Υλοποίηση της Cilk

Γενικά η Cilk ήταν πολύ αυτοματοποιημένη και δεν είχε καμία ιδιαιτερότητα στην υλοποίηση της.

- **test_octree:** Το μόνο "δύσκολο" κομμάτι που παρατήρησα ήταν ότι ήθελε πρώτα να κάνω __cilkrts_end_cilk(); Να κλείσω την cilk και μετά να θέσω τον αριθμό των thread με την __cilkrts_set_param("nworkers",num_threads); και μετά να κάνω initialize __cilkrts_init();. Τον αριθμό των thread τον ήθελε σε string οπότε τον διάβαζα κατευθείαν από το όρισμα που έδινα όταν καλούσα την test_octree. Γενικά κι εδώ έβαλα ένα έξτρα όρισμα για τον αριθμό των thread. Μέγιστο αριθμό νημάτων άφηνε η cilk μέχρι 128 στον **διάδη** . Αυτό το έλεγξα με μια printf printf("%d\n",__cilkrts_set_param("nworkers",num_threads)); η οποία επιστρέφει 0 για success και μου επέστρεφε 2 (Parameter value out of range) για 256 και παραπάνω threads, γυρνούσε μετά στους default workers που στο διάδη ήταν 8. Γενικά τα νούμερα 128, 8 δεν είναι συγκεκριμένα εξαρτιόνται από τον επεξεργαστή του κάθε υπολογιστή.
- **hash_codes:** Εδώ δεν δούλευε σωστά η cilk_for,είχε πρόβλημα με τις 2 for που υπήρχαν και οπότε άλλαξα λίγο το τρόπο επανάληψης ως εξής:

```
cilk_for(i=0; i<N; i++){
  codes[i*DIM ] = compute_code(X[i*DIM ], low[0], step);
  codes[i*DIM + 1] = compute_code(X[i*DIM + 1], low[1], step);
  codes[i*DIM + 2] = compute_code(X[i*DIM + 2], low[2], step);
}
```

- **morton_encoding, radix_sort, data_rearrangement:** Απλά μια cilk_for αρκούσε για να παραλληλοποιηθούν. Στην radix_sort κάνω cilk_for στην αναδρομή .

```
cilk_for(i=0;i<MAXBINS;i++){
    int offset = (i>0) ? BinSizes[i-1] : 0;
    int size = BinSizes[i] - offset;
 truncated_radix_sort(&morton_codes[offset],
  &sorted_morton_codes[offset],
  &permutation_vector[offset],
  &index[offset], &level_record[offset],
  size,
  population_threshold,
  sft-3, lv+1);
 }
```

Υλοποίηση OpenMP

- **test_octree:** Κι εδώ έβαλα ένα έξτρα όρισμα που ήταν ο αριθμός των thread(numOfThreads).Επσίσης έχω βάλει έλεγχο σε περίπτωση εισαγωγής λανθασμένης τιμής .

```
if (numOfThreads<= 0) {
  numOfThreads = omp_get_num_procs(); //default number of threads
  }
```

- **hash_codes, morton_encoding, data_rearrangement :** Εδώ χρησιμοποιώ την παραλληλοποίηση της openMP για την for(δεν παραθέτω όλο το κώδικα απλά κάθε ένα τα κομμάτια της παραλληλοποίησης για να φανεί τι έβαλα private και ποιες shared μεταβλητές).

```
Quantize:     #pragma omp parallel num_threads(numOfThreads) shared(X, low, step)
  {
  #pragma omp for private(i, j)
Morton_encoding: #pragma omp parallel num_threads(numOfThreads) shared(mcodes,codes)
        {
        #pragma omp for private(i)
Data_rearrangement:    #pragma omp parallel num_threads(numOfThreads)
              {
#pragma omp for private (i)
```

Οι υπόλοιπες μεταβλητές πάνε by default shared. Η υλοποίησή της είναι εύκολη και δεν παρουσιάζει καμία δυσκολία.

- **Radix_sort:** Η υλοποίηση που έχω κάνει εδώ στην αναδρομική κλίση είναι ίδια με αυτήν στα pthread. Η openMP απλά θέλει άλλες δύο μεταβλητές τις allow_parallelism και active_threads.
  - Με τον ίδιο τρόπο ελέγχω αν έχω available_threads .

➔ Αν έχω, μέσα σε critical section (#pragma omp critical) ελέγχω πάλι αν έχω available_threads και αν ναι, θέτω την allow_parallelism=1(αλλιώς 0). Μετά την χρησιμοποιώ ώστε να θέσω omp_set_nested(allow_parallelism) enable or disable ανάλογα την τιμή του allow_parallelism. Μετά υπάρχουν οι 3 περιπτώσεις που έκανα και στα pthread απλώς εδώ λόγω του nested είναι πολύ πιο εύκολο. Παραθέτω τον κώδικα δίπλα.
Ανάλογα τον αριθμό τον available_threads έχω τις περιπτώσεις που φαίνονται και θέτω αντίστοιχα την active_threads = ή με MAXBINS(8) ή available_threads και μειώνω τα available_threads. Στην δεύτερη περίπτωση μηδενίζω τα available_threads. Όλο αυτό είναι σε critical section γιατί κανένα άλλο thread δεν πρέπει να αλλάξει την τιμή των active_threads όσο και των available_threads. Έπειτα εκτελώ την παράλληλη for με num_threads(active_threads). Εδώ βάζω και schedule static ώστε να μη παραλληλοποιηθεί δυναμικά αλλά να γίνει MAXBINS/active_threads το grainsize. Το θέλω στατικό και να μην αλλάζει. Στην ουσία για παραπάνω από 8 νήματα, κάθε νήμα να παίρνει μόνο μια αναδρομή. Αντίστοιχα και για λιγότερα threads.

```
omp_set_nested(allow_parallelism);
#pragma omp critical
 {
    if(omp_get_nested()!=0 && available_threads>=MAXBINS){
        active_threads=MAXBINS;
available_threads=available_threads-MAXBINS;


}
 else if (omp_get_nested()!=0 && available_threads<MAXBINS &&
available_threads>=0){
 active_threads=available_threads;
 available_threads=0;
 }
 }
```

```
#pragma omp parallel private(i) num_threads(active_threads)
{
   #pragma omp for nowait \
   schedule(static)
```

➢ Αν δεν έχω available_threads(=0) για να αποφύγω την καθυστέρηση των critical section, με την else πάω κατευθείαν σειριακά. Αυτό εξυπηρετεί η πρώτη if else που ελέγχει τα available_threads.

Παρατήρηση: Κι εδώ είδα καθυστέρηση όταν αύξανα τον αριθμό των available_threads, όταν τελείωνε κάποιο νήμα οπότε δεν το έκανα έτσι.
Για καλύτερη κατανόηση του κώδικα καλό θα ήταν να δείτε και την υλοποίηση της radix_sort στο αρχείο της.

Διαγράμματα για την ταχύτητα των υπολογισμών συγκριτικά με το σειριακό κώδικα

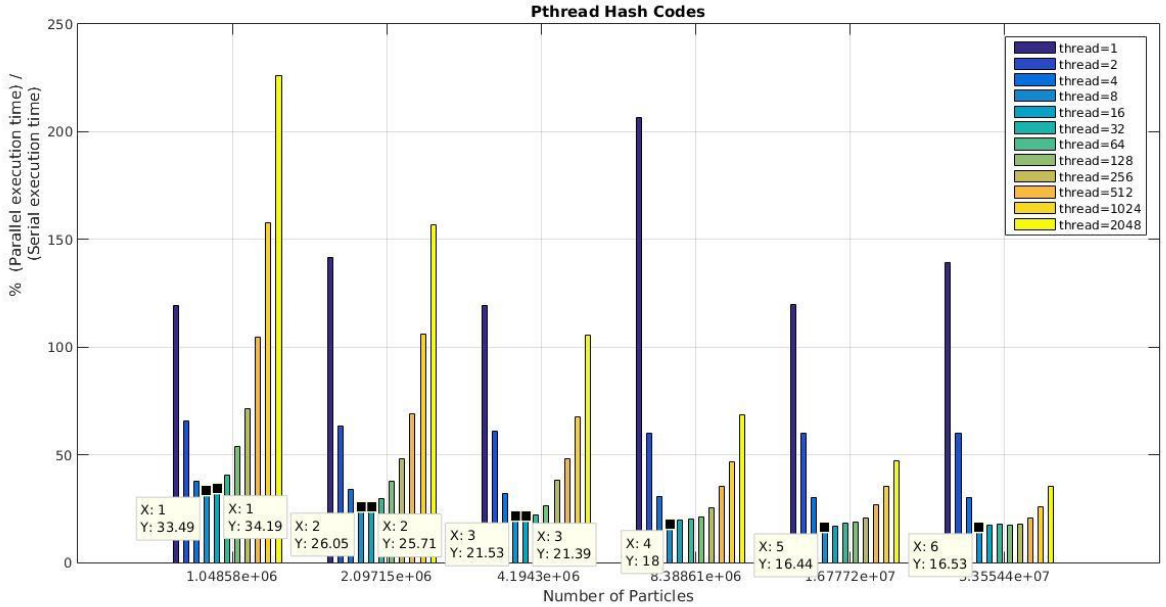
Όπου υπάρχει ποσοστό είναι το κλάσμα : [(Parallel execution time)/(Serial execution time)]*100, που δηλώνει σε ποσοστό το χρόνο που έκανε να εκτελεστεί ο παράλληλος κώδικας σε σχέση με τον σειριακό. Πχ, αν έχω 60% σημαίνει ότι ο παράλληλος χρειάστηκε το 60% του σειριακού χρόνου για να εκτελεστεί. (δηλαδή αν ο σειριακός έκανε 1s, ο παράλληλος έκανε 0.6s)

Πρέπει να λάβουμε υπόψιν ότι ο αριθμός των επαναλήψεων ήταν 10, στα διαγράμματα παρουσιάζονται οι μέσες τιμές της μετρούμενης ποσότητας και ότι όταν έκανα τα τεστ, στο διάδη υπήρχαν κι άλλοι χρήστες.
Τα Ν δυστηχώς δεν τα παρουσίασε όμορφα η matlab οπότε παραθέτω εδώ τις τιμές τους

N=1048576 2097152 4194304 8388608 16777216 33554432

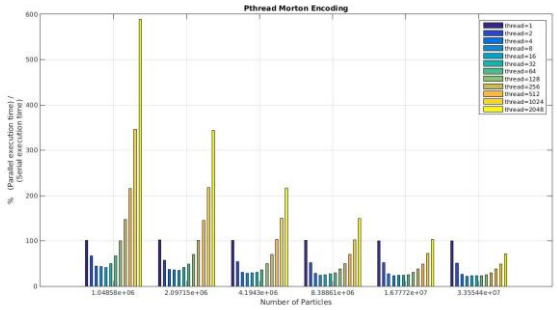
Επίσης, ο διάδης έχει 8 threads , οπότε θα περιμένω τους καλύτερους χρόνους μου να τους βρώ στα 8 threads. Όπου δεν συμβαίνει αυτό θα το σχολιάζω.

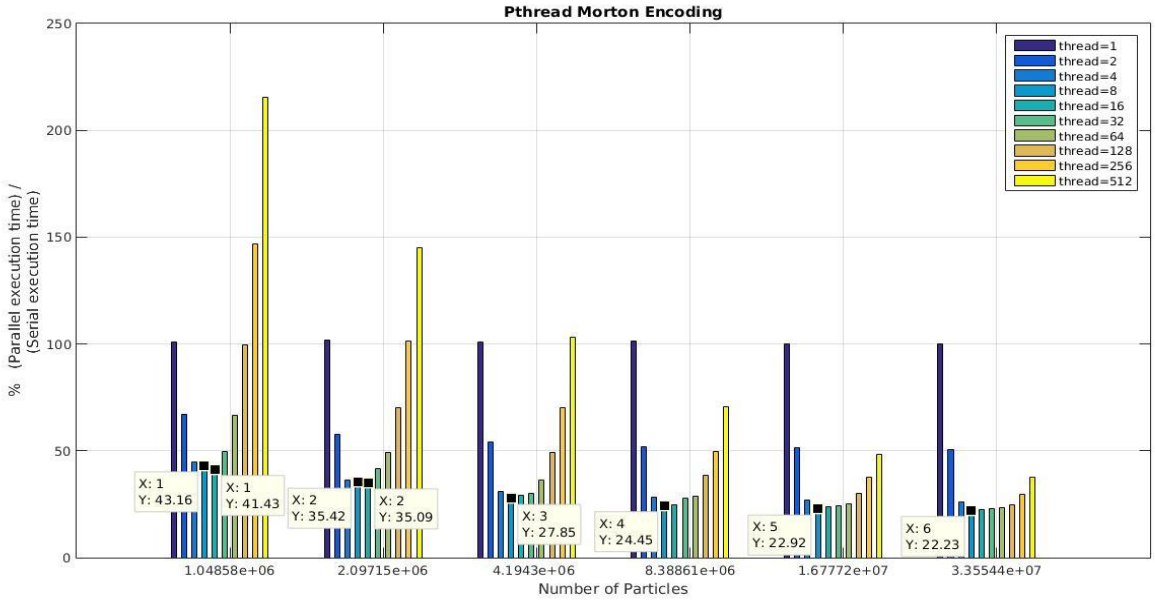
Cube Pthread



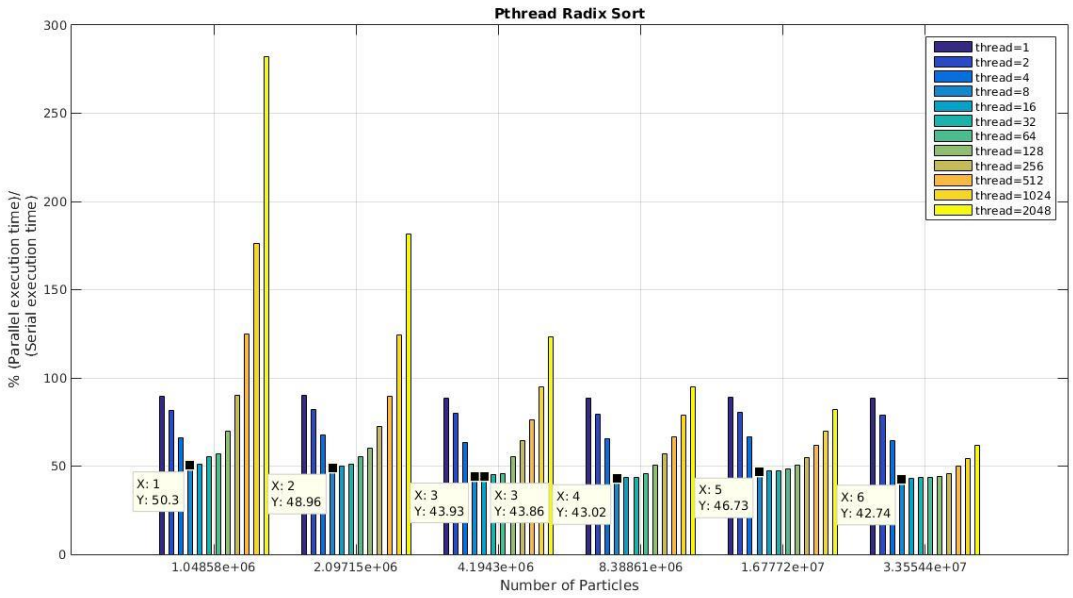
Η hash code εκτελείται σε ποσοστό 33.5%-16.53% σε σχέση με το χρόνο εκτέλεσης της σειριακής hash code. Λόγω των 8 thread που υποστηρίζει ο διάδης είναι αναμενόμενο να πετυχαίνεται ο καλύτερος χρόνος στα 8 threads. Σε κάποιες περιπτώσεις, όπως φαίνεται στο διάγραμμα, τα 16 thread είναι ταχύτερα. Εφόσον όμως δεν έχουν μεγάλες διαφορές δεν το θεωρώ σημαντικό. Μπορεί να φταίει κάποια τιμή στην επανάληψη επειδή πχ έτρεχε και κάποιος άλλος μαζί μου ή οτιδήποτε άλλο. Σε γενικές γραμμές τους καλύτερους χρόνους τους πετυχαίνω στα 8 thread.

Επειδή σε μεγάλο αριθμό thread ο χρόνος εκτέλεσης του παράλληλου κώδικα υπερβαίνει έως και σχεδόν 600% ,στο επόμενο διάγραμμα αφαιρώ τα thread 1024 2048 εφόσον φαίνεται ξεκάθαρα ότι οι χρόνοι χειροτερεύουν. Την ίδια λογική για καλύτερη παρουσίαση των διαφορών θα εφαρμόζω και στα υπόλοιπα διαγράμματα.

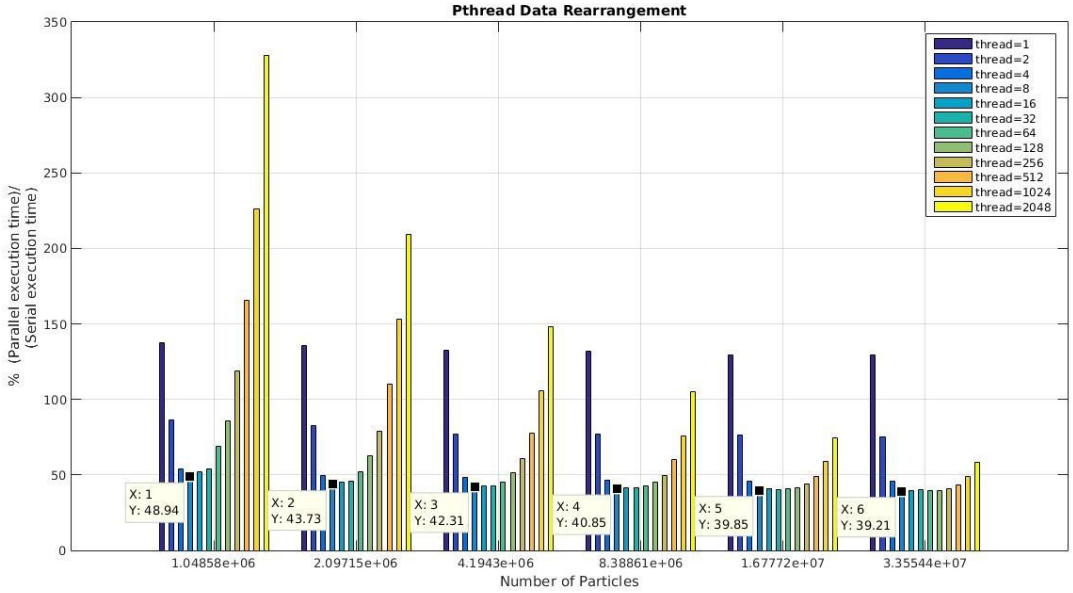




Όμοια με τα προηγούμενα βλέπουμε ελάχιστος χρόνος πάλι για 8 thread με 2 εξεραίσεις που δεν έχουν μεγάλες αποκλίσεις μεταξύ τους.

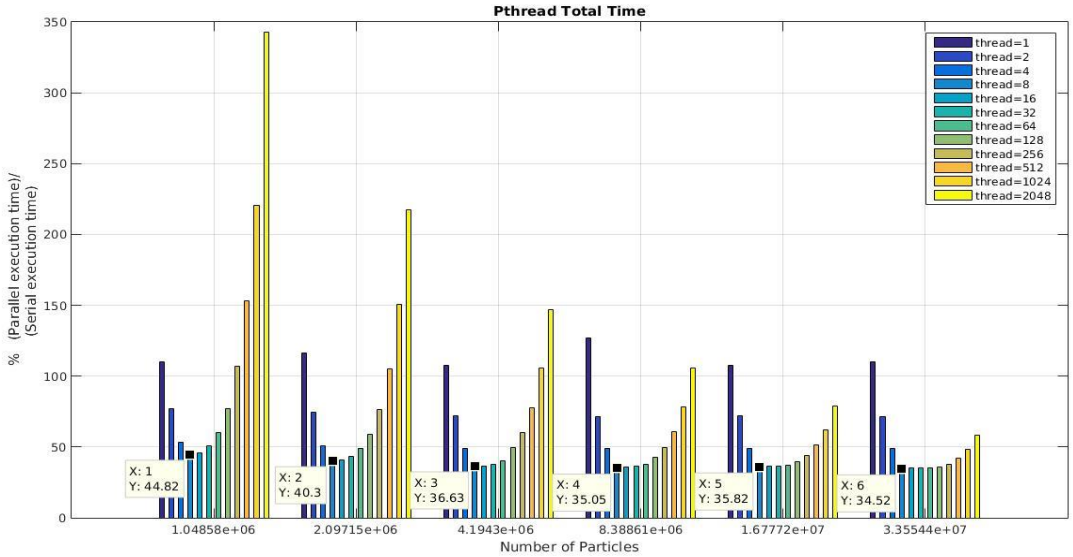


Το ίδιο και στην radix sort καλύτερο χρόνο πετυχαίνω στα 8 thread .



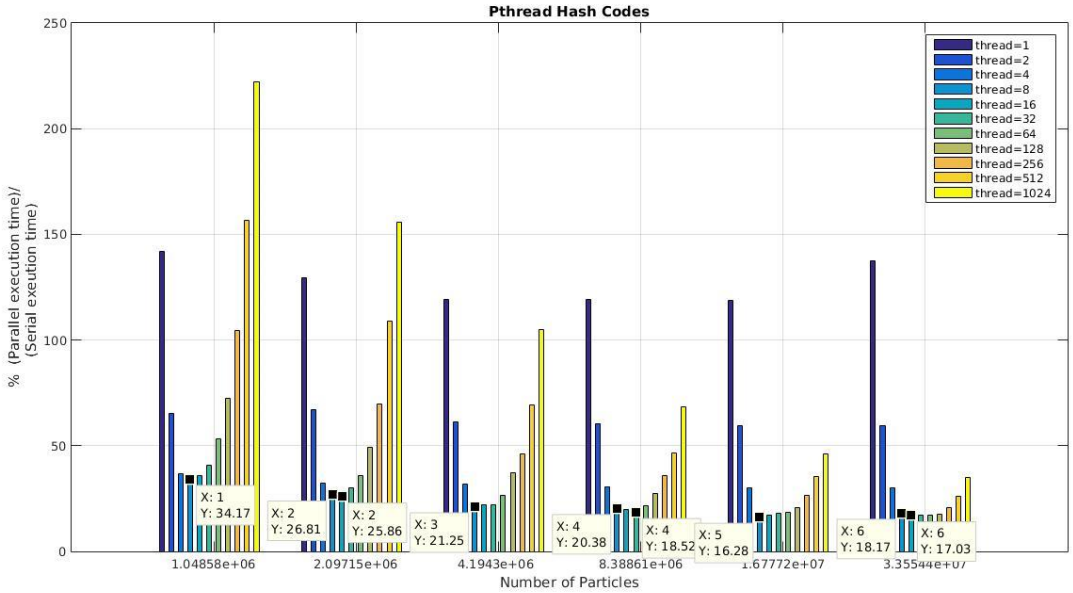
Όμοια στα 8 thread έχω τον καλύτερο χρόνο.

Παρατήρηση: Γενικά σε μικρά N και μεγάλο αριθμό thread οι συναρτήσεις ξεπερνάνε σε χρόνο την σειριακή εκτέλεση ενώ σε μεγάλα N υπάρχει άνοδος αλλά δεν την ξεπερνά για μέχρι 2048 thread που έτρεξα τα τεστ μου.

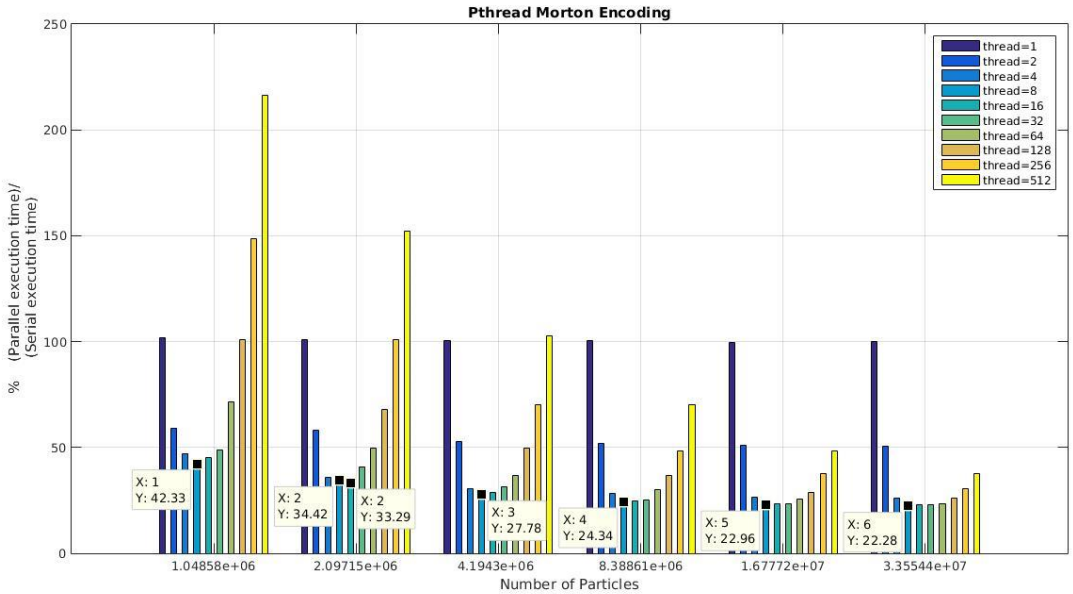


Στο συνολικό χρόνο φαίνεται τελικά ότι έχω τους καλύτερους χρόνους εκτέλεσης για τα 8 thread. Όλοι οι χρόνοι είναι μικρότεροι έως 45% και όσο μεγαλώνει το N βλέπω καλύτερη βελτίωση που είναι λογικό για τι το πρόγραμμα τρέχει παράλληλα.

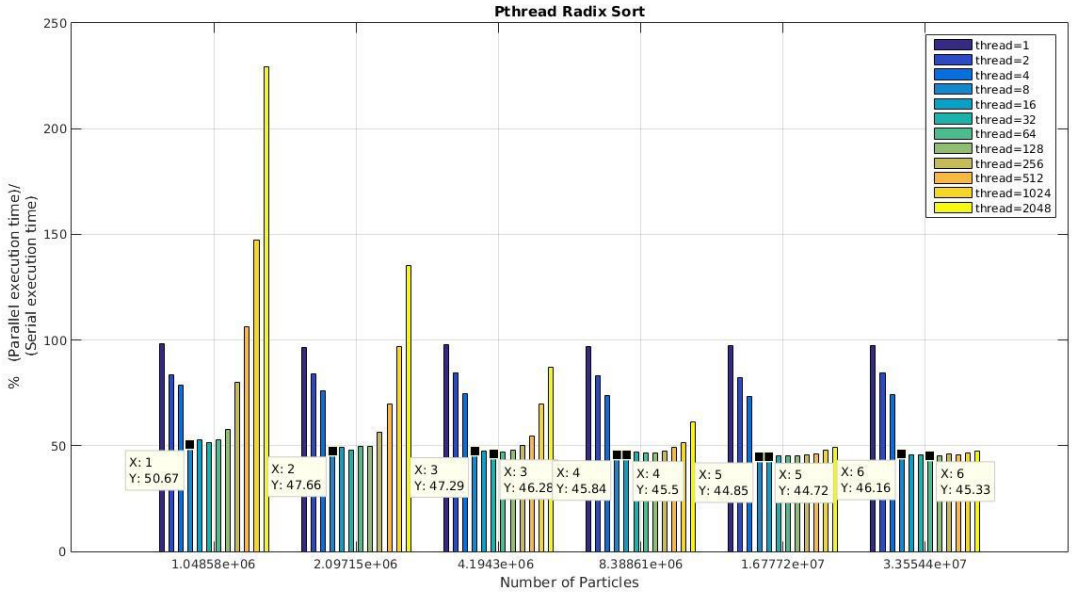
Pthread Sphere



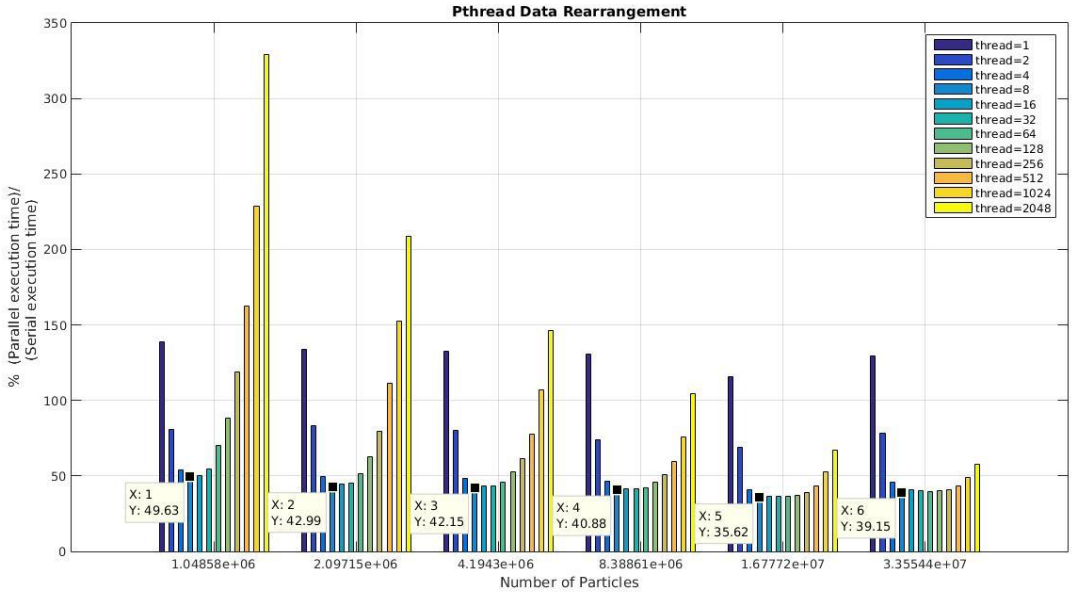
Για τη σφαίρα πάλι έχω τους καλύτερους χρόνους στα λίγα thread κυρίως στα 8. Έχει όμως και αλλού ελάχιστα τα οποία είναι πολύ κοντά σε ποσοστό σε σχέση με τα 8 thread. Οπότε για μεγαλύτερο πλήθος επαναλήψεων πιστεύω θα πήγαινε γύρω από τα 8 thread το ελάχιστο.



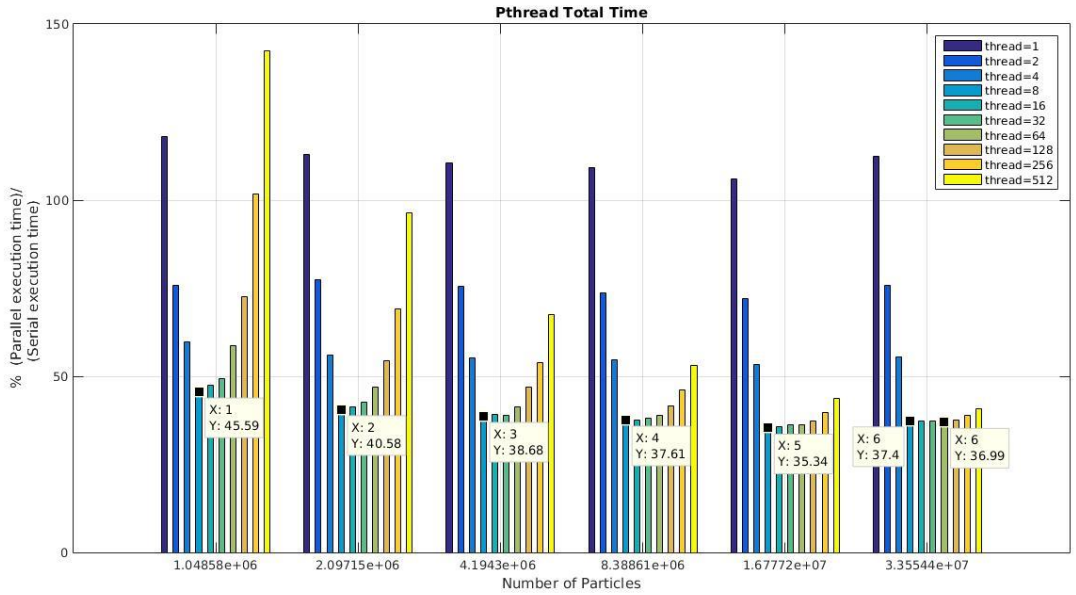
Εδώ φαίνεται πιο ξεκάθαρα ότι ο καλύτερος χρόνος είναι τα 8 thread .



Κανονικά θεωρώ ότι ο χρόνος θα έπρεπε να ήταν αλλού ελάχιστος στις περισσότερες τον περιπτώσεων είναι όμως σε μεγαλύτερο αριθμό thread. Με παραξενεύει ότι βρήκα ελάχιστο στα 64 thread στα μέγιστο N. Ίσως έτρεχε κάτι ακόμη μαζί μου, ίσως φταίνε οι 10 μόνο επαναλήψεις που έκανα τις οποίες θεωρώ λίγες.



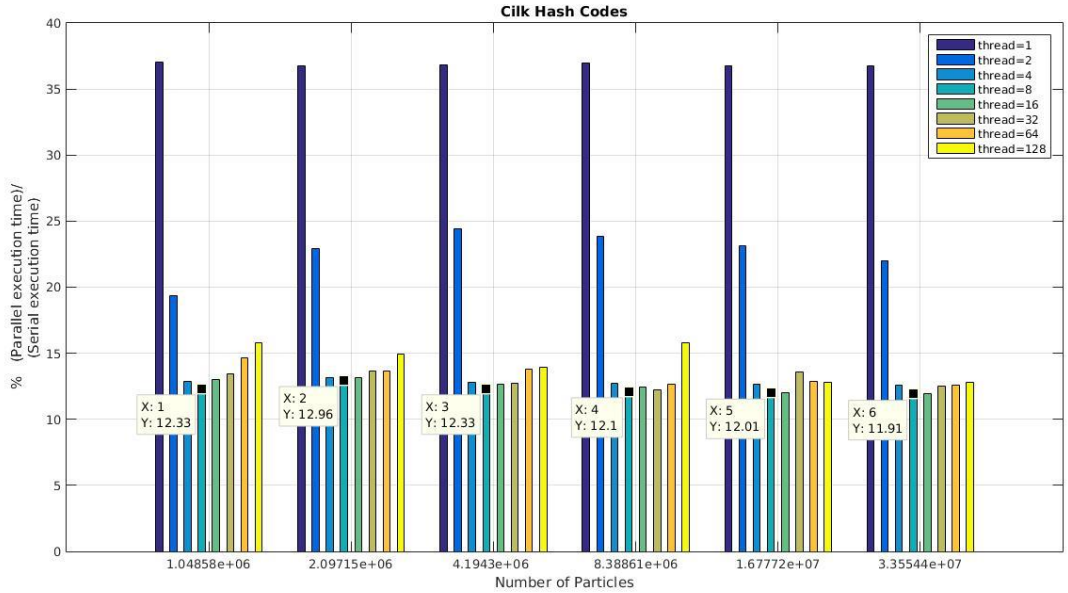
Εδώ φαίνεται πιο ξεκάθαρα ότι ο καλύτερος χρόνος είναι τα 8 thread .



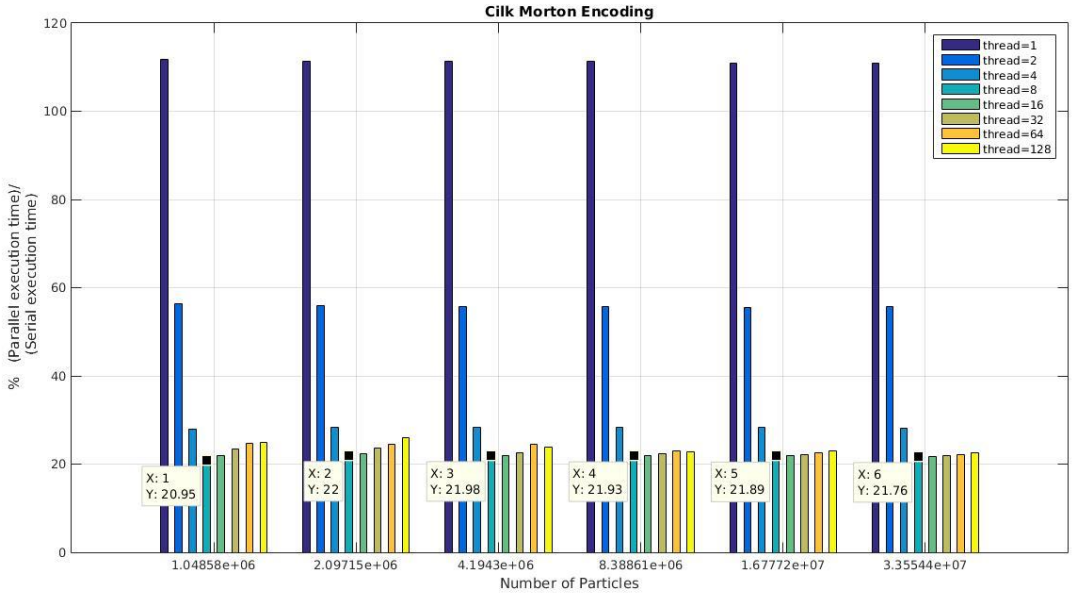
Όμοια φαίνεται καλύτερος χρόνος εκτέλεσης τα 8 thread εκτός του τελευταίου N που είναι πάλι τα 64 αλλά η διαφορά τους είναι μικρή και ίσως για παραπάνω επαναλήψεις να βελτιωνόταν. Ωστόσο με παραξενεύει.

CILK CUBE

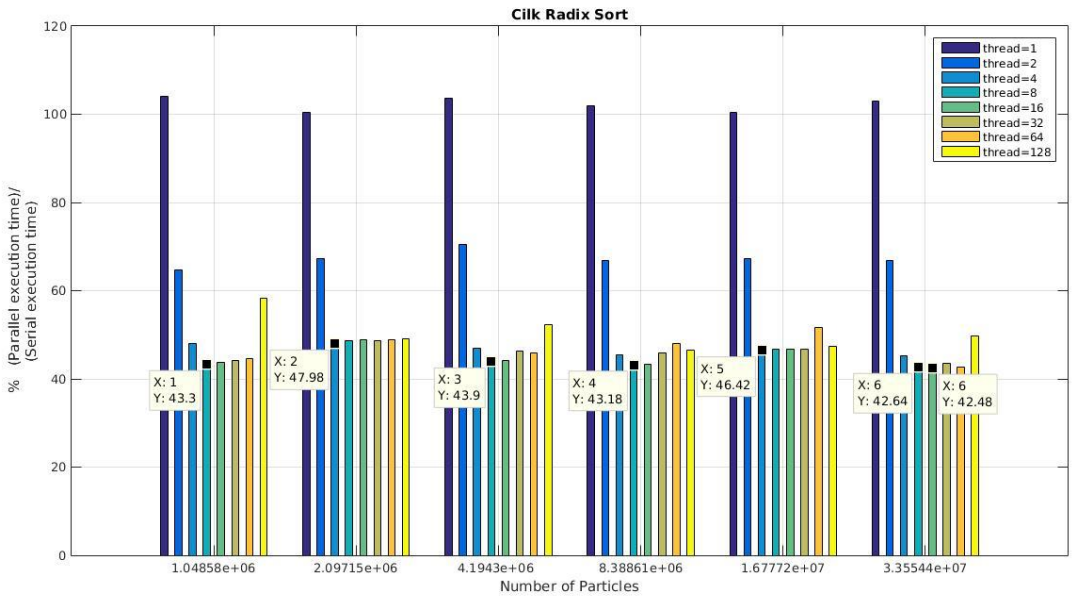
Η Cilk επιτρέπει μέχρι 16*core_workes thread οπότε τρέχει μέχρι 128 thread. Για μεγαλύτερη τιμή thread επιστρέφει στα default του που εδώ είναι τα 8. Για το λόγο αυτό στα διαγράμματα παρουσιάζονται οι τιμές μέχρι 128.



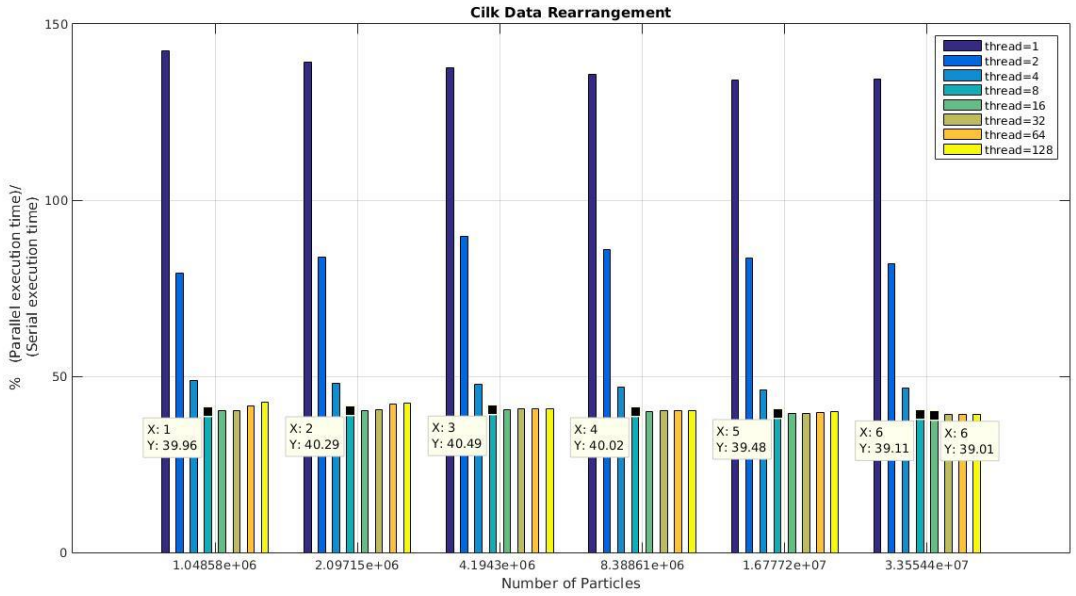
Όμοια με τα προηγούμενα ο ελάχιστος χρόνος είναι για 8 νήματα. Περίεργο είναι το γεγονός ότι σε 1 νήμα ο χρόνος εκτέλεσης του παράλληλου είναι 40% του συνολικού. Αν και έχω αλλάξει την cilk hash code αυτό δεν δικαιολογεί τέτοια μεταβολή. Περαιτέρω ανάλυση της υλοποίησης γίνεται στις αρχικές σελίδες της αναφοράς.



Όμοια με τα προηγούμενα ο ελάχιστος χρόνος είναι για 8 νήματα.

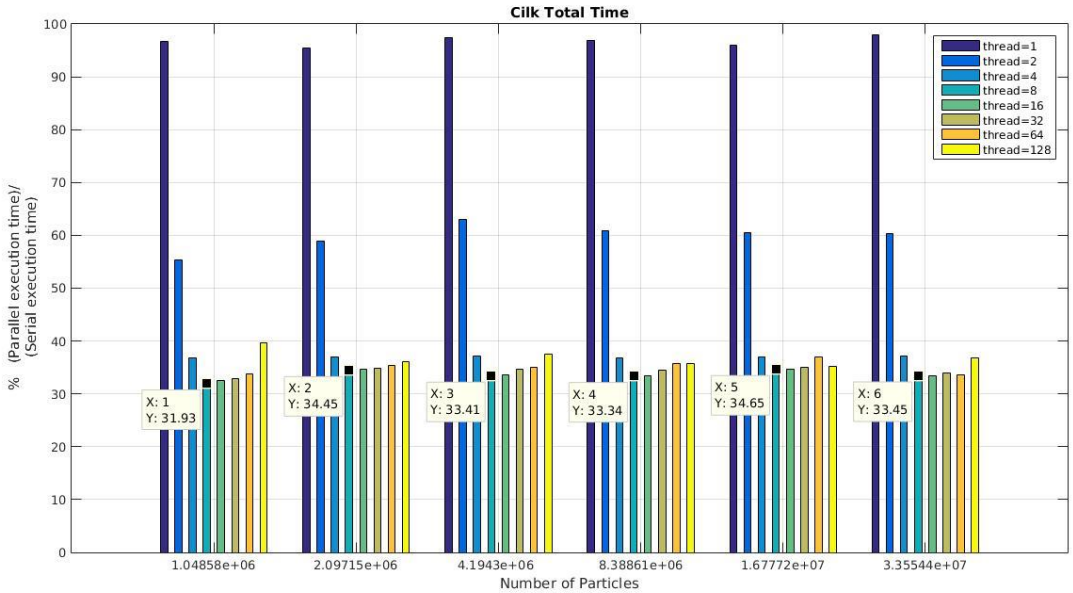


Ελάχιστος χρόνος πάλι φαίνεται να είναι στα 8 thread με εξαίρεση για N=33554432, που πάλι δεν φαίνεται ουσιαστική διαφορά και θα μπορούσε για μεγαλύτερο αριθμό επαναλήψεων να μην υπήρχε.



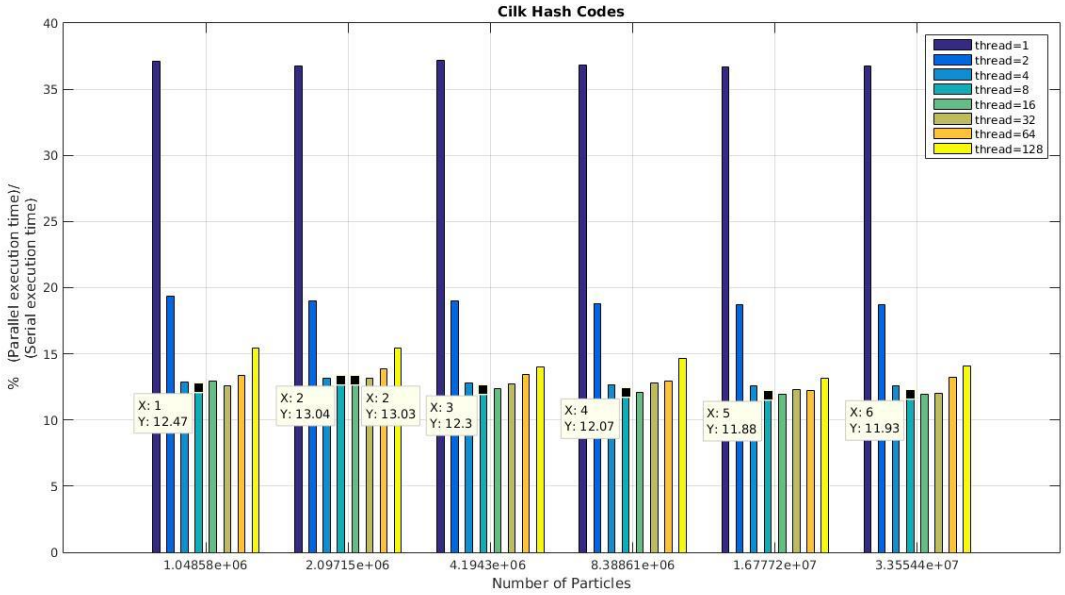
Όμοια με τα προηγούμενα παρατηρώ βέλτιστο χρόνο υλοποίησης στα 8 νήματα με εξαίρεση στο τελευταίο N για 16 thread που και πάλι η διαφορά τους είναι αμελητέα.

Παρατήρηση: Γενικά στη Cilk παρατήρησα ότι από 8 thread έως 128 (που ήταν το μέγιστο που μπορούσα να βάλω) δεν έχει μεγάλες αυξήσεις στο χρόνο εκτέλεσης ιδικά στην data rearrangement που είναι σχεδόν ίδιοι. Θεωρώ,χωρίς να ξέρω πολλά από τον τρόπο διαχείρισης των thread από την cilk, ότι αυτό οφείλεται στο ότι χειρίζεται με τον βέλτιστο τρόπο τα thread και καθορίζει το grainsize των for δυναμικά.

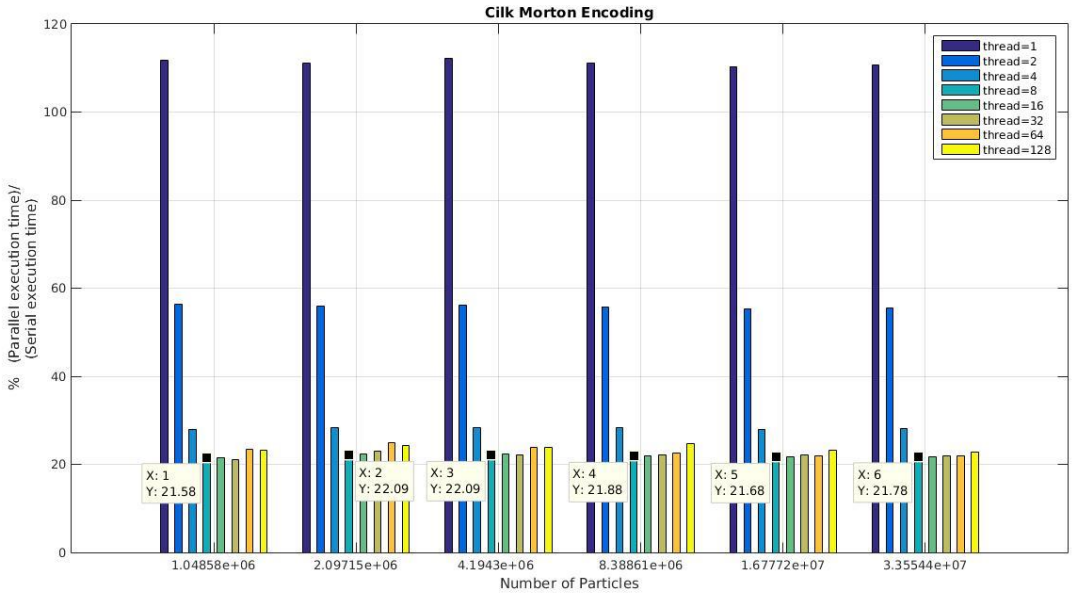


Εδώ φαίνεται ότι ο συνολικός χρόνος εκτέλεσης είναι ο καλύτερος δυνατός στα 8 threads και παρατηρούνται μικρές αυξήσεις του χρόνου για threads>8.

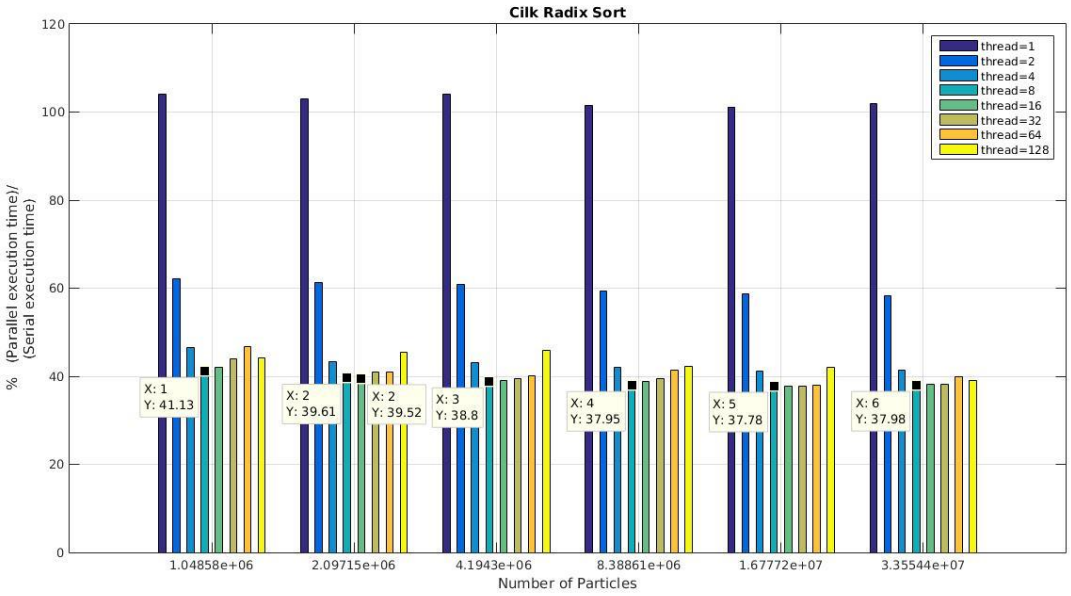
Cilk Sphere



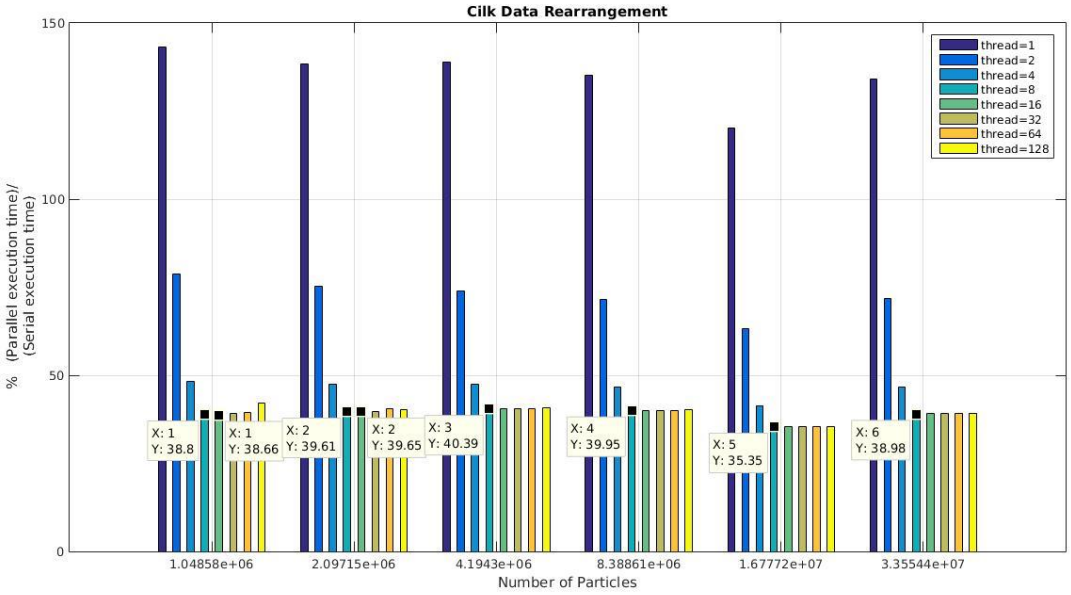
Όμοια με τα προηγούμενα ο ελάχιστος χρόνος είναι για 8 νήματα. Πάλι συμβαίνει να είναι στο 40% ο χρόνος για 1 thread που μου φαίνεται περίεργος.



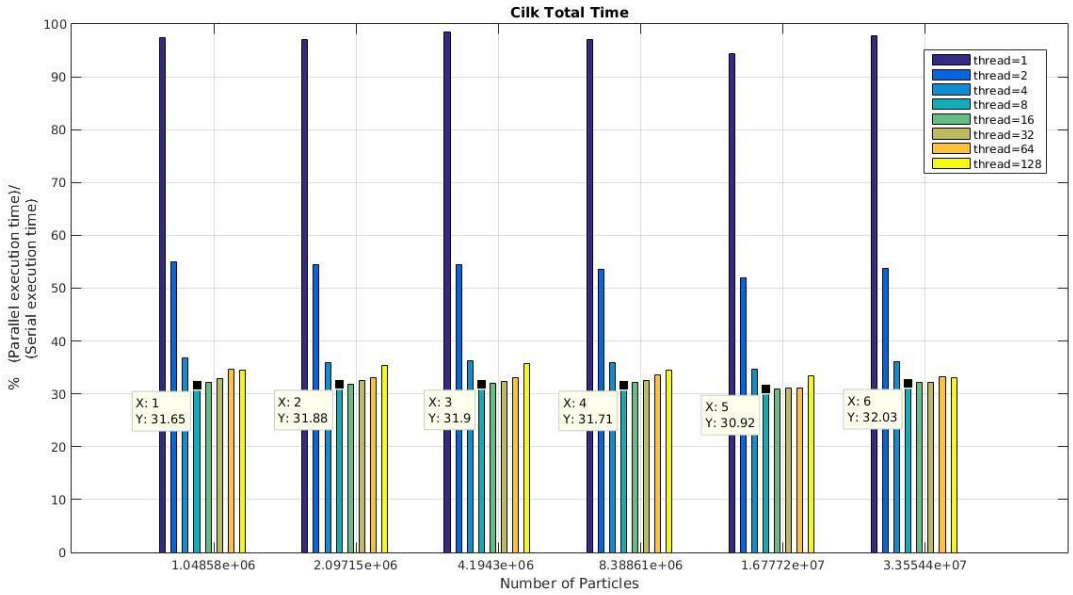
Όμοια με τα προηγούμενα ο ελάχιστος χρόνος είναι για 8 νήματα.



Όμοια ο ελάχιστος χρόνος είναι για 8 thread. Με εξαίρεση στο N=2097152 που ο ελάχιστος χρόνος ειναι 16 νήματα αλλά οι χρόνοι υλοποίησης των 8 και 16 σχεδόν δεν απέχουν.

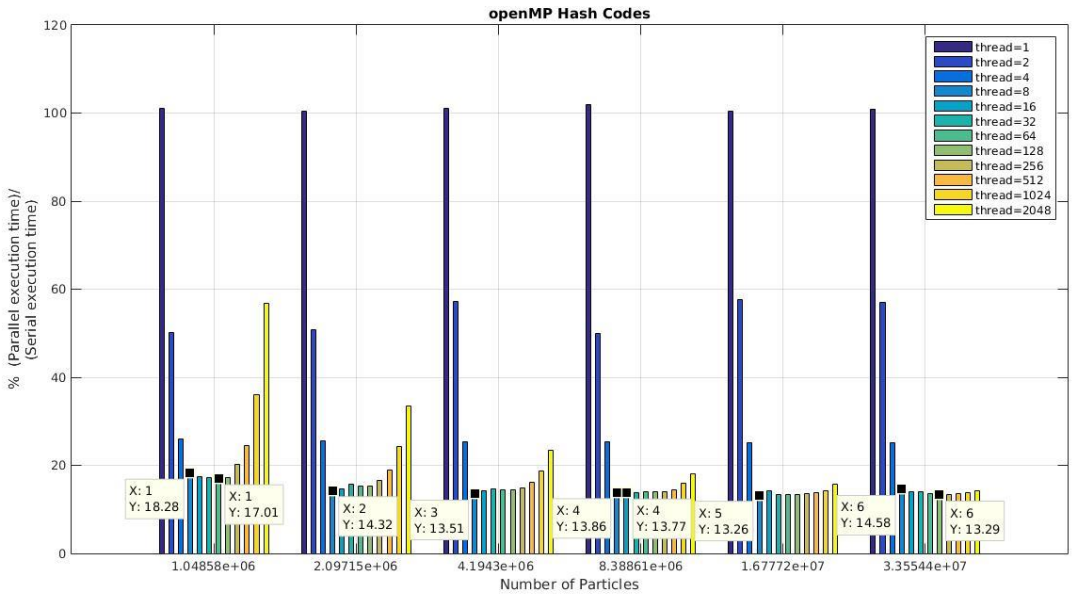


Και εδώ οι χρόνοι φαίνεται να συγκλίνουν ώστε η καλύτερη υλοποίηση να είναι στα 8 thread πάλι με διαφοροποίηση στο N= 2097152 που η διαφορά των χρόνων είναι αμελητέα.

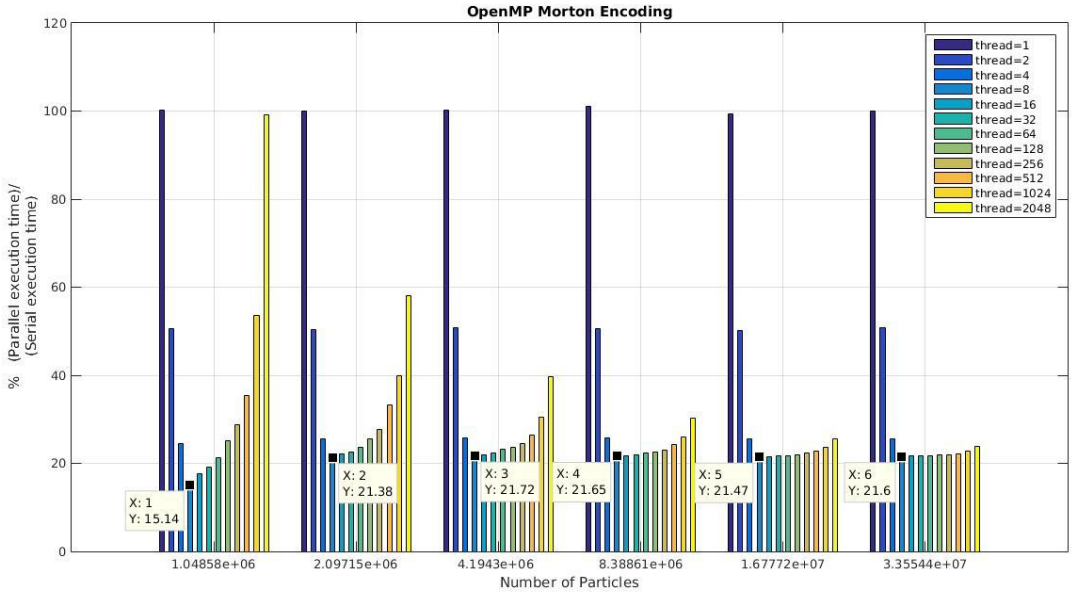


Τελικά ο συνολικός χρόνος φαίνεται είναι ο καλύτερος για 8 threads όπως ήταν αναμενόμενο.

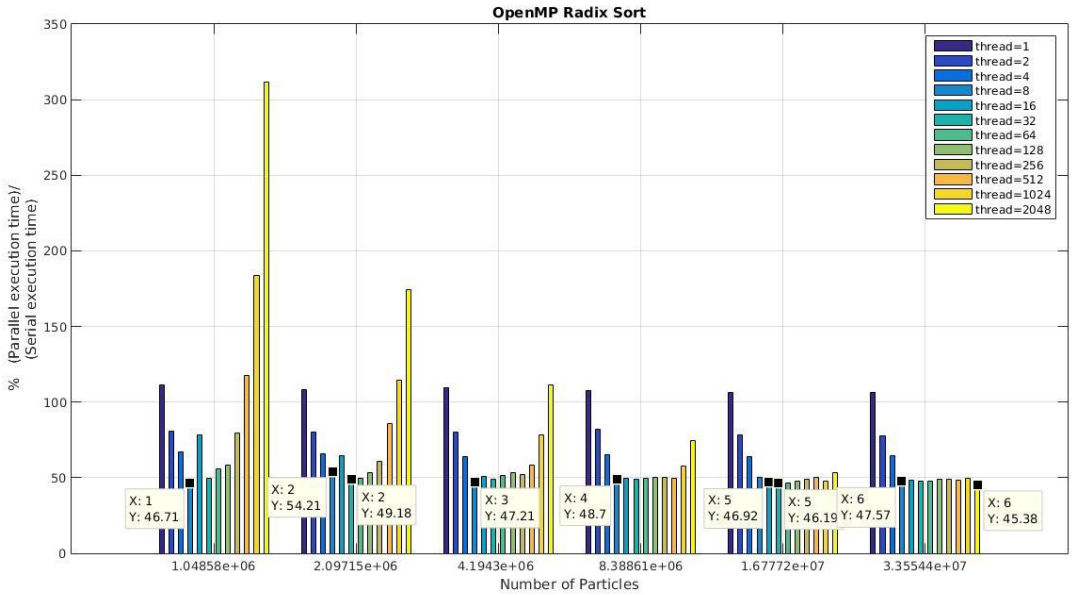
OpenMP Sphere



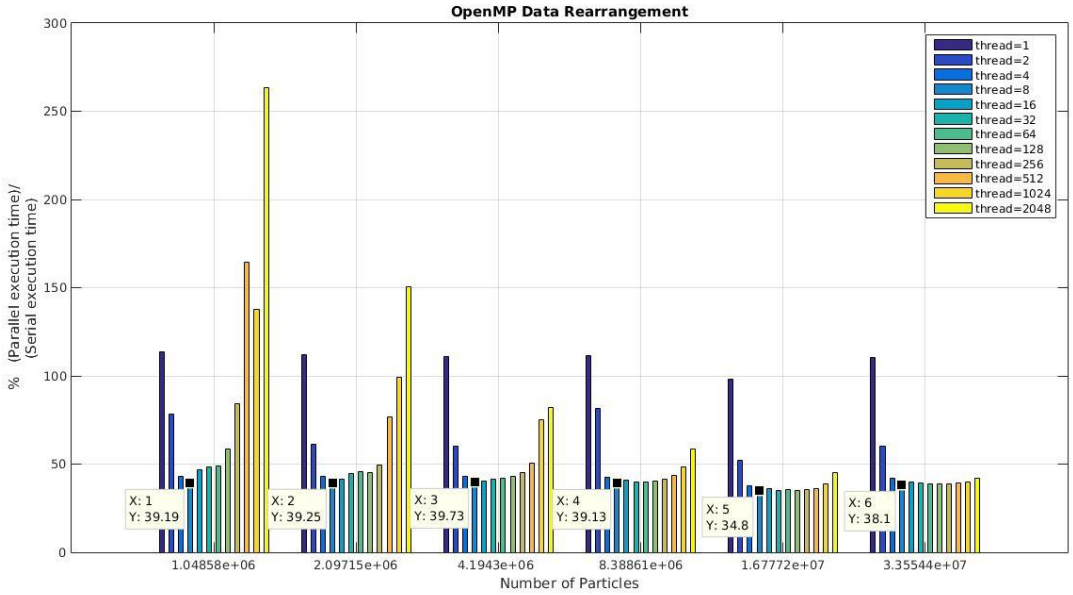
Αν και περίμενα συγκεντρωμένους τους ελάχιστους χρόνους στα 8 νήματα, παίζουν λίγο με αρκετά μη λογικές αποκλίσεις. Πχ στο 1ο N και τελευταίο, το ελάχιστο βρίσκεται στα 64 και 128 αντίστοιχα. Δεν μπορώ να ξέρω γιατί συνέβη αυτό, πιστεύω ότι θα έτρεξε κάποιος μαζί μου εκείνη την στιγμή, πάντως δεν το θεωρώ πολύ φυσιολογικό.



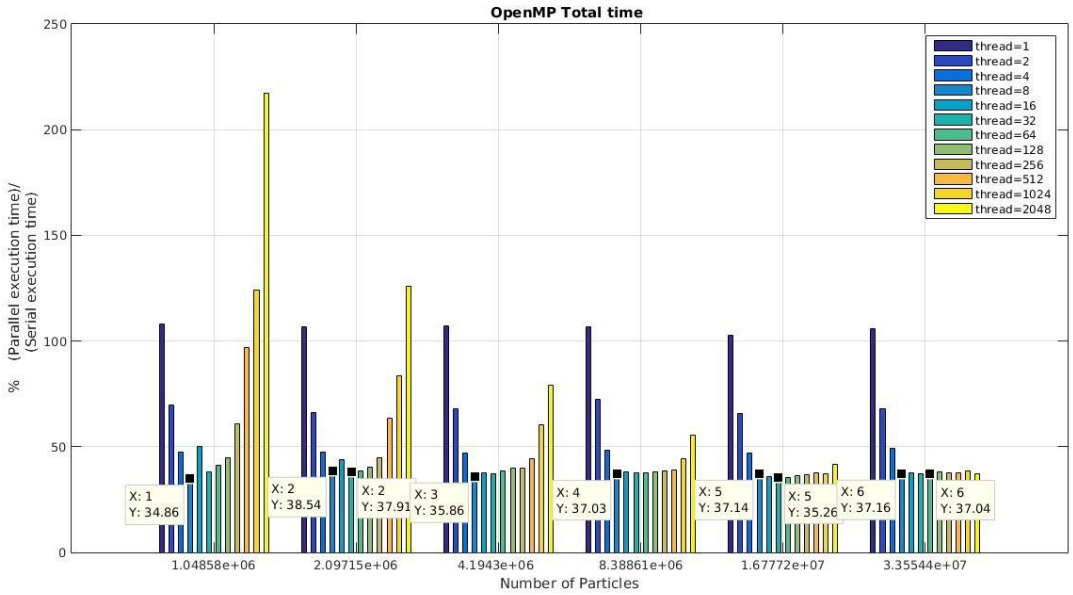
Εδώ οι ελάχιστοι χρόνοι είναι στα 8 threads.



Κι εδώ υπάρχουν διαφορές στον αριθμό των thread που εμφανίζονται τα ελάχιστα . Στο N=2097152 φαίνεται ότι για 8 thread έχω μικρό χρόνο για 16 μεγαλώνει απότομα και 32 έχει μεγάλη πτώση. Επίσης ιδιαίτερα περίεργο είναι το γεγονός ότι ελάχιστο χρόνο έχω στα **2048 thread** για N=33554432 το οποίο δεν θα έπρεπε να συμβαίνει. Δεν νομίζω να έχω θέμα στον κώδικα μου όσο ότι έπρεπε να πάρω καλύτερα στατιστικά κάποια τιμή ήταν λάθος.



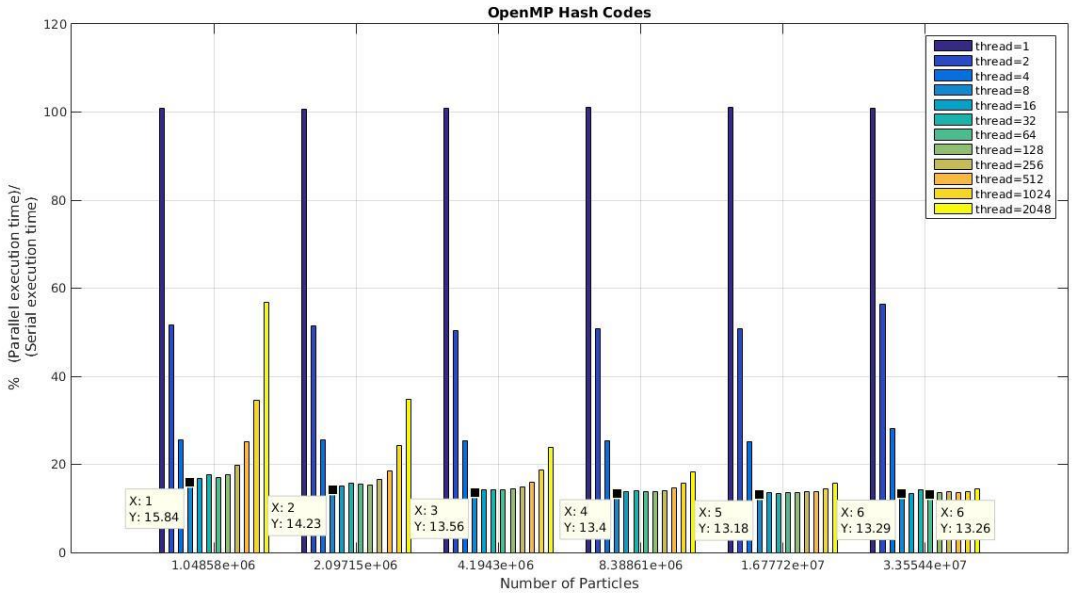
Εδώ είναι ξεκάθαρα οι ελάχιστες τιμές των χρόνων στα 8 threads



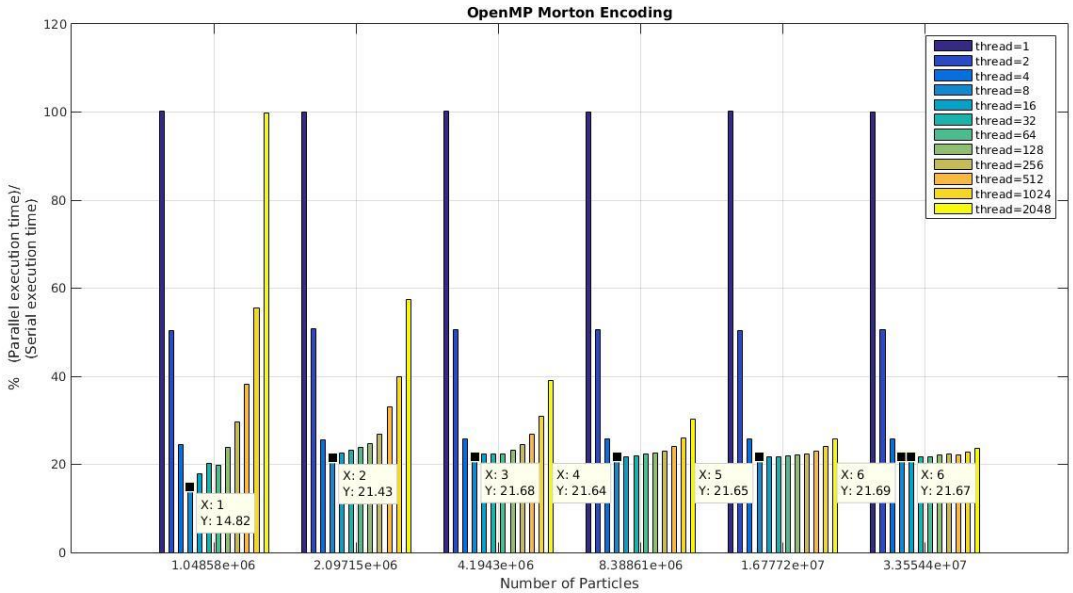
Γενικά φαίνεται να παίζει πολύ ο ελάχιστος χρόνος, ωστόσο με προβληματίζει το γεγονός ότι στο N=209715e+06 είναι τα 16 threads πιο αργά από τα 32 και 64 κλπ. Το ίδιο πράγμα συμβαίνει και στο πρώτο N για 16 νήματα. Γενικά οι μετρήσεις που πήρα δεν μου άρεσαν.

Παρατήρηση: Θα περίμενα να βγάλω ως βέλτιστη υλοποίηση αυτή στα 8 thread , όμως οι τιμές παίζουν. Σε αυτό ίσως να φταίει ο μικρός αριθμός των επαναλήψεων ή ότι έτρεχε κάτι/κάποιος παράλληλα στα δικά μου τεστ.

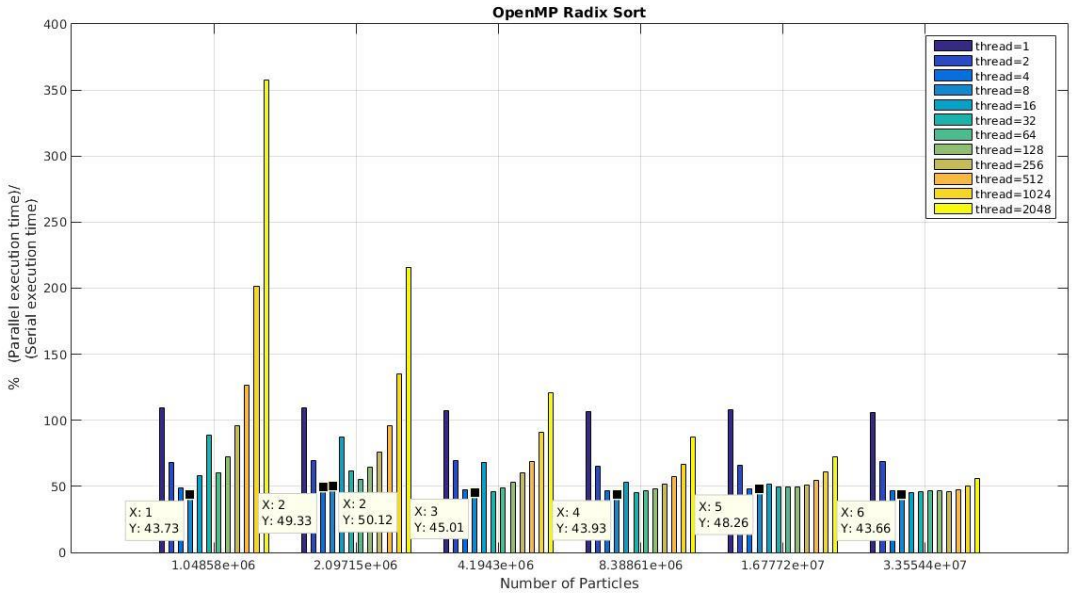
OpenMP Cube



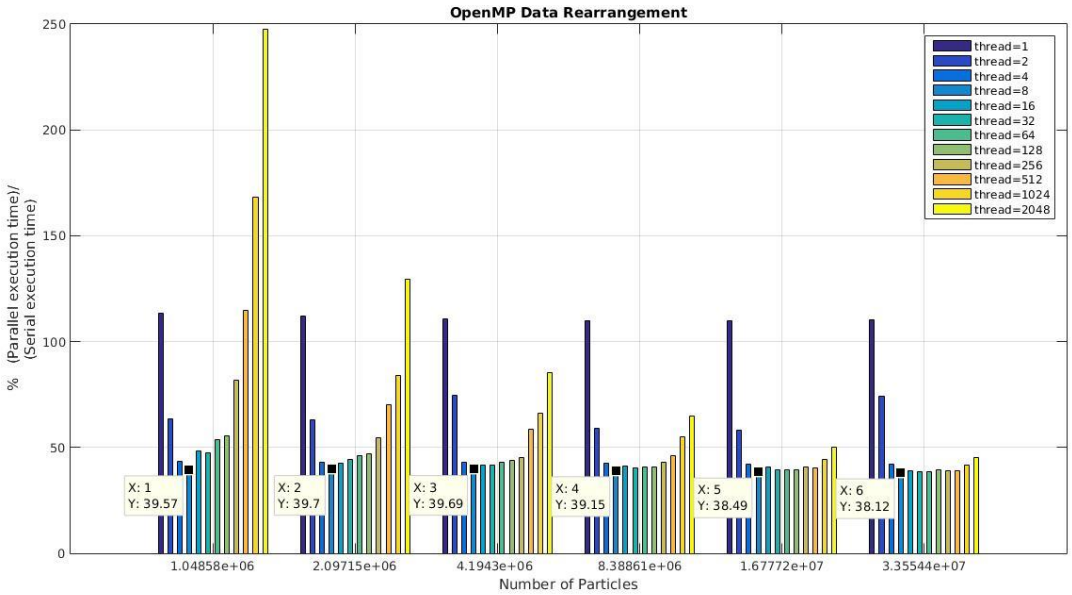
Ο βέλτιστος χρόνος φαίνεται να είναι στα 8 thread εκτός του τελευταίου N που είναι στα 64 αν και τα ποσοστά δεν διαφέρουν πολύ και μπορεί να είναι λόγο κάποιας “λάθους” τιμής. Γενικά στις 10 επαναλήψεις είναι δύσκολο με μόνο το μέσο όρο να βγάλεις καλά στατιστικά.



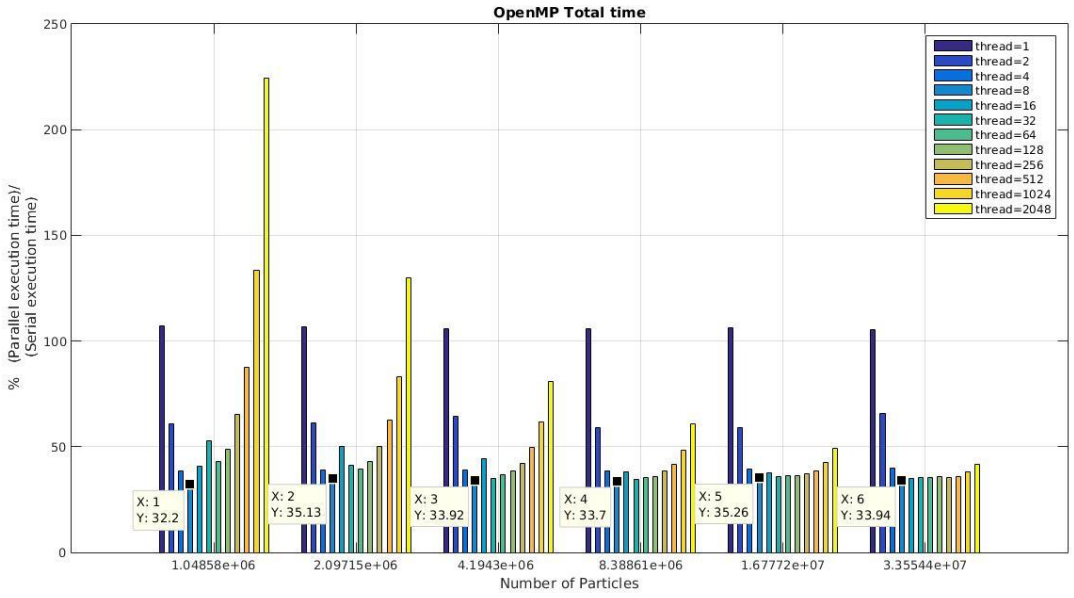
Όμοια στα 8 νήματα είναι ο καλύτερος χρόνος και πάλι στο τελευταίο N εμφανίζεται για 16 threads με ελάχιστη διαφορά χρόνου από τα 8.



Όμοια στα 8 νήματα είναι ο καλύτερος χρόνος για τα περισσότερα N. Στο δεύτερο N εμφανίζεται για 4 threads με ελάχιστη διαφορά χρόνου από τα 8.



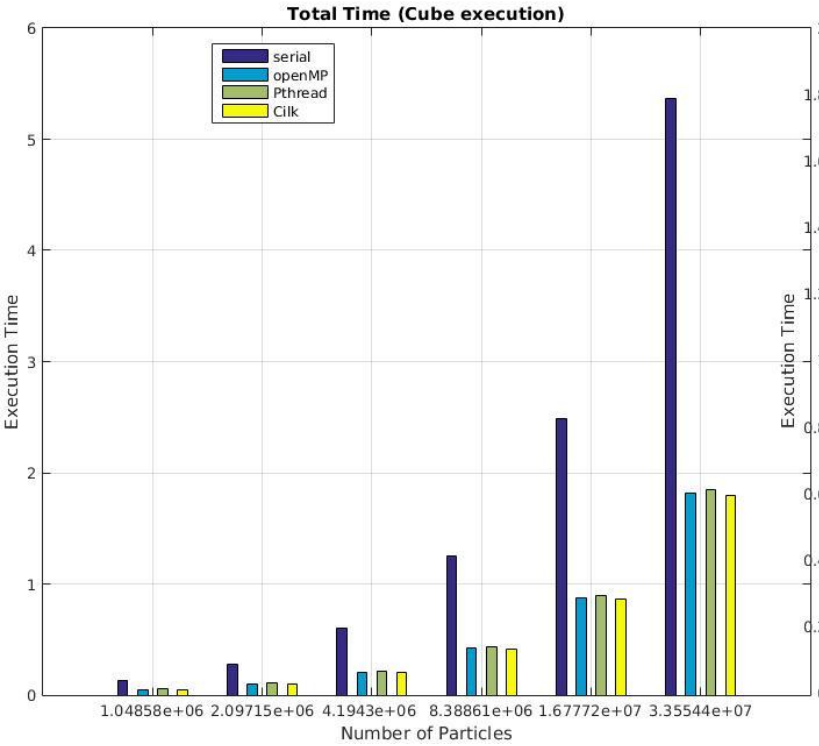
Εδώ είναι ξεκάθαρα ο καλύτερος χρόνος στα 8 νήματα.

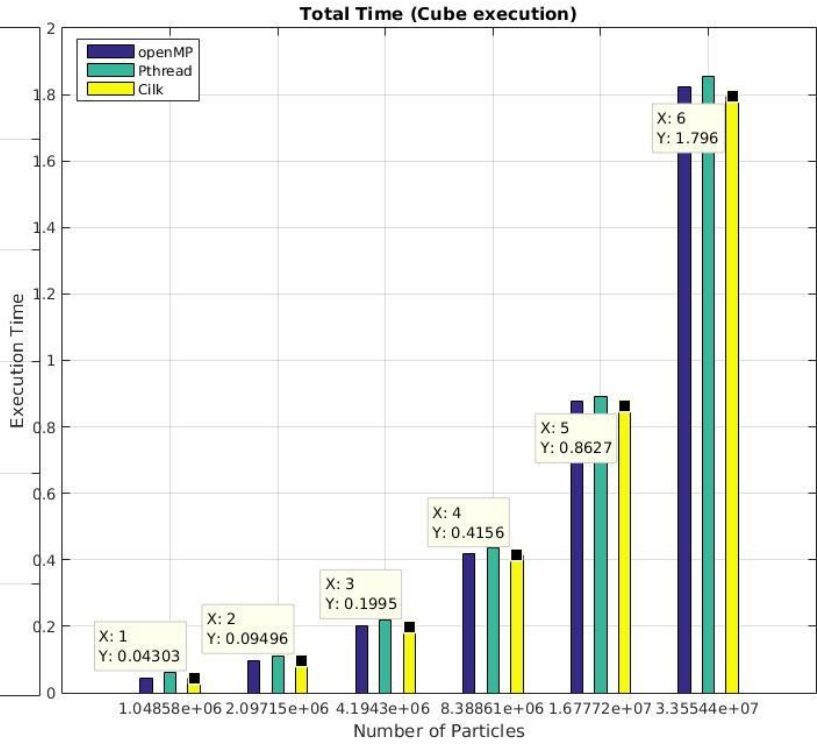


Όμοια κι εδώ που είναι πλέον ο συνολικός χρόνος της υλοποίησης και δεν παρατηρώ κάτι ασυνήθιστο. Οι ελάχιστοι χρόνοι είναι στα 8 νήματα.

Συγκρίσεις μεταξύ των παράλληλων υλοποιήσεων

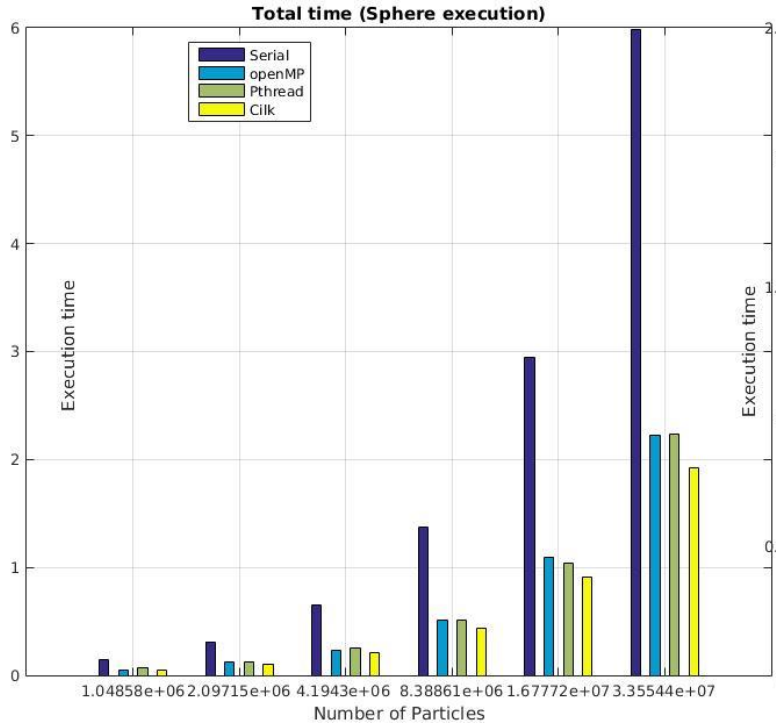
Cube

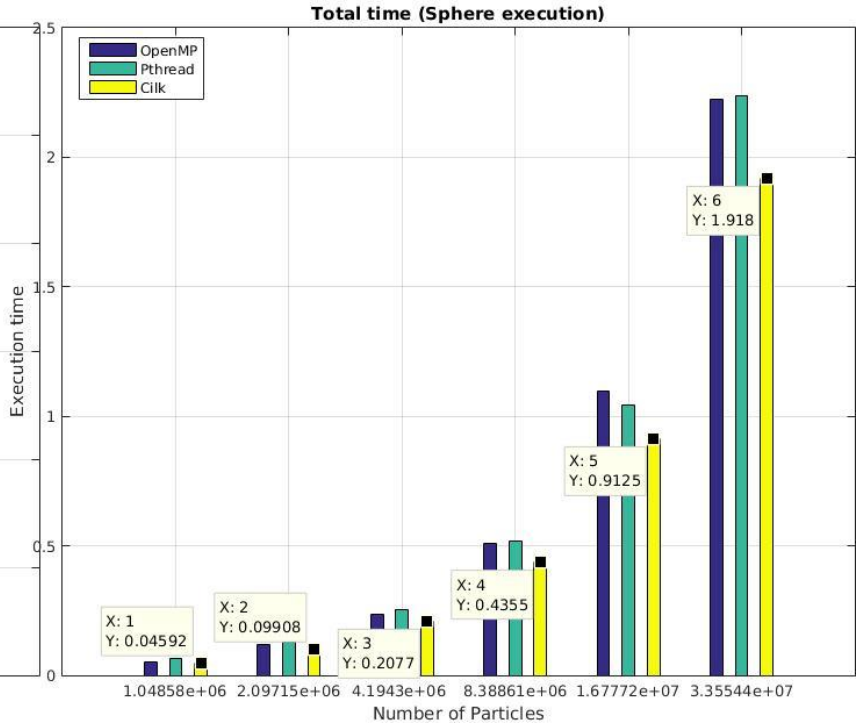




Στην υλοποίηση με τα σωματίδια σε ομοιόμορφη κατανομή σε κύβο παρατηρώ ότι καλύτερη υλοποίηση είναι αυτή της cilk. Με ελάχιστη διαφορά από την OpenMP και δυστυχώς τελευταία η υλοποίηση με pthread.

Sphere





Όμοια φαίνεται πως ο καλύτερος χρόνος πετυχαίνεται με την Cilk και ακολουθούν η openMP και Pthread.

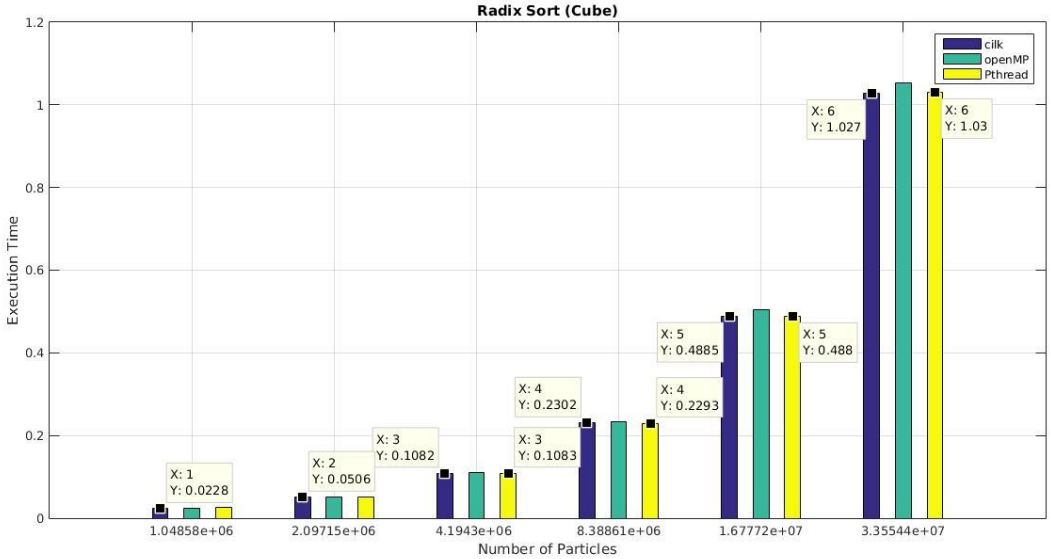
Σχολιασμός: Θα ήθελα να ήταν καλύτερη η pthread, στην οποία αφιέρωσα και τον περισσότερο χρόνο, αλλά από ότι φαίνεται η Cilk κάνει την βέλτιστη παραλληλοποίηση, τουλάχιστον σε σύγκριση με την δική μου υλοποίηση των pthread. Στα pthread φαίνεται να υστερώ αρκετά στην παράλληλη υλοποίηση των for στις hash_codes morton_encoding και data_rearrangement. Δεν συντρέχει λόγος να δείξω με διαγράμματα την διαφορά φαίνεται στα πιο πάνω διαγράμματα. Θα αναφέρω απλά ποσοστά (στο περίπου) για τα 8 threads για όλα τα N.

	Pthread	Cilk	OpenMP
Hash_codes	33-16.5%	~12%	16-13%
Morton_encoding	43-22%	20-22%	~21%
Data_rearrangement	49-39%	39-40%	38-40%

Τα ποσοστά δείχνουν τον % χρόνο εκτέλεσης του παράλληλου προς τον σειριακό και είναι από τις υλοποιήσεις για Cube όμοια είναι για Sphere. Συνήθως μικραίνουν όσο αυξάνεται το N.

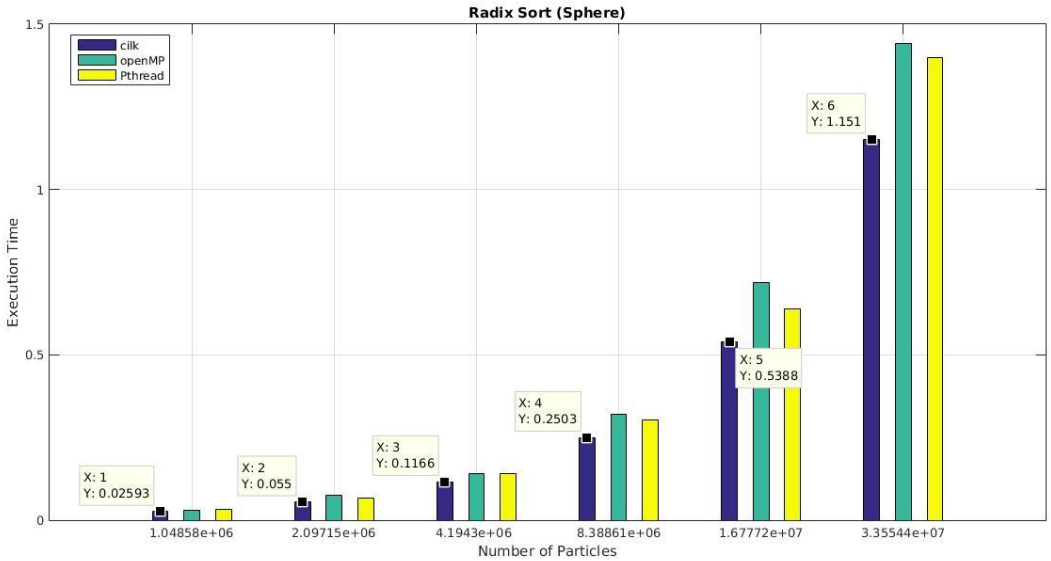
ΣΥΓΚΡΙΣΗ ΤΗΣ RADIX SORT

Cube



Για την Radix Sort φαίνεται ότι για την ομοιόμορφη κατανομή του κύβου οι υλοποιήσεις της cilk και pthread είναι πολύ κοντά με μερικές φορές καλύτερη την pthread.

Sphere



Στην ομοιόμορφη κατανομή σε 1/8 της επιφάνειας της σφαίρας είναι καλύτερη η cilk. Η OpenMP υλοποιήθηκε με το ίδιο τρόπο με τα pthread οπότε είναι λογικό να υστερεί κι αυτή.