ΠΑΡΑΛΛΗΛΑ ΚΑΙ ΔΙΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

ΕΡΓΑΣΙΑ Ι

Παραλληλοποίηση προγράμματος Octree

ΜΑΣΤΟΡΑΣ ΡΑΦΑΗΛ ΕΥΑΓΓΕΛΟΣ

ΑΕΜ 7918

ΝΟΕΜΒΡΙΟΣ 2016

Περιγραφή του προγράμματος

Ο κώδικας που μας δόθηκε υλοποιεί μια ιεραρχική ομαδοποίηση N σωματιδίων σε οκταδικό δέντρο, για αριθμό σωματιδίων,μέγιστο ύψος δέντρου ,όριο πληθυσμού και χώρο (μοναδιαίο κύβο ή 1/8 μοναδιαίας σφαίρας) που επιλέγονται από τον χρήστη.
Το πρόγραμμα αρχικά υπολογίζει τυχαίες τιμές για τις συντεταγμένες των N σωματιδίων, μέσα στο εύρος τιμών της εκάστοτε κατανομής με την συνάρτηση create_dataset().
Μετά κωδικοποιεί τις συντεταγμένες των σωματιδίων ανάλογα με την θέση τους μεσώ μιας διαδικασίας κβάντισης με την συνάρτηση compute_hash_codes() .
Έπειτα μέσω της morton_encoding() για κάθε σωματίδιο, με διαδοχικά shift των bits για κάθε μία από τις 3 συντεταγμένες του, παράγεται ένας κώδικας Morton , ο οποίος στην ουσία αντιπροσωπεύει και τις τρεις συντεταγμένες του σωματιδίου σε έναν κώδικα.
Ύστερα ταξινομούνται μερικώς τα σωματίδια ανάλογα με των κώδικα Morton τους, μέσω μίας recursive Radix sort MSD(most significant digit).
Επιπλέον ανακατατάσσει τα σωματίδια στην μνήμη ώστε κοντινά σωματίδια να βρίσκονται σε κοντινές θέσεις μνήμης μέσω της data_rearrangement().
Τέλος ελέγχετε αν οι πίνακες index[] και sorted_morton_codes[] έχουν ταξινομηθεί σωστά με τις συνάρτησεις check_index() και check_codes(), δηλαδή αν κάθε σωματίδιο x έχει index[x]=x και σωματίδια που βρίσκονται στο ίδιο επίπεδο έχουν ίδια κωδικοποίηση morton_codes, και εμφανίζεται το ανάλογο μήνυμα.

Περιγραφή της εργασίας

Ζητούμενο της εργασίας αυτής είναι να παραλληλοποιήσουμε το πρόγραμμα αυτό με τρεις τεχνικές: **Pthreads**, **OpenMp** καθώς και **Cilk**. Έπειτα να πάρουμε χρόνους για κάθε μία από αυτές τις τεχνικές και να συγκρίνουμε τα αποτελέσματα με τους χρόνους της αρχικής έκδοσης αλλά και μεταξύ τους.

Σχόλια: Όλα τα αποτελέσματα πάρθηκαν στον server diades. Επειδή στον server αυτόν συνδέονται πολλοί φοιτητές , ενδέχεται λόγο αυξημένης κινητικότητας μερικά αποτελέσματα να εμφανίζουν απρόσμενες μεταβολές, παρότι έγινε προσπάθεια να παρθούν τα αποτελέσματα σε ώρες χαμηλής κινητικότητας ώστε να είναι πιο αντιπροσωπευτικά.
Επίσης για κάθε υλοποίηση που θα παρουσιάσω έχει γίνει έλεγχος ορθής εκτέλεσης με τις συναρτήσεις check_index και check_codes , και για οποιαδήποτε επιλογή εισόδου που ζητήθηκε τα αποτελέσματα έχουν ελεγχθεί και επιτύχει. Επιπλέον επισυνάπτω μαζί με τα υπόλοιπα αρχεία που ζητήθηκαν και τα αρχεία των αποτελεσμάτων.

Επιλογή συναρτήσεων προς παραλληλοποίηση

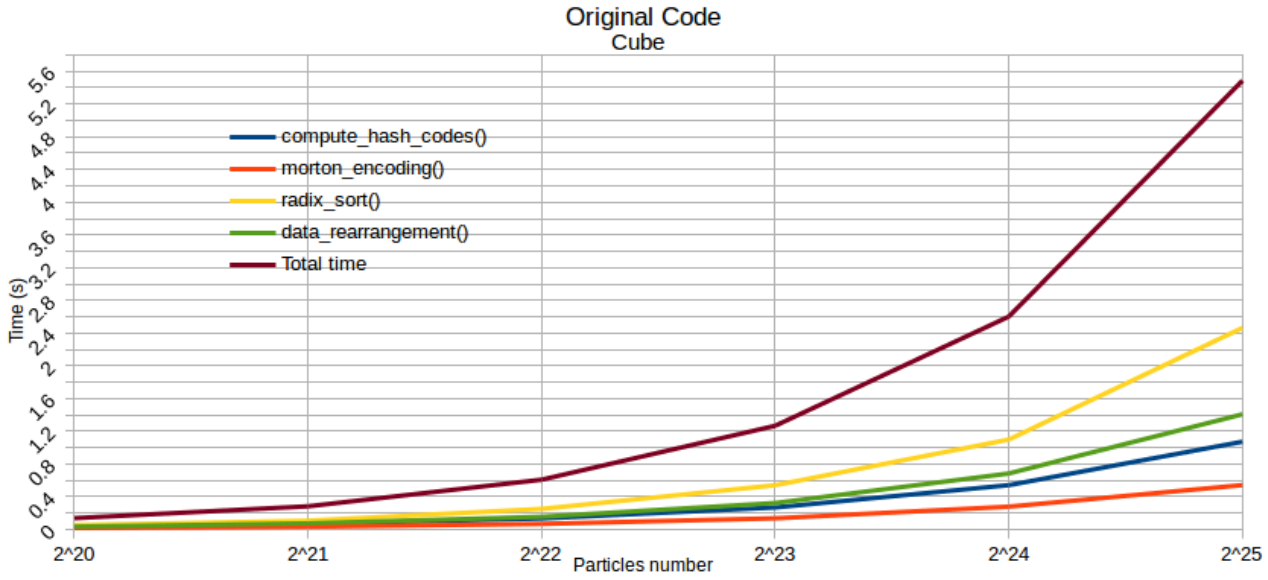
Αρχικά για να επιλέξω ποιες από τις συναρτήσεις του προγράμματος θα παραλληλοποιήσω, χρησιμοποίησα το εργαλείο GPROF.
Έκανα δηλαδή compile τον αρχικό κώδικα που δόθηκε με χρήση του flag **-pg**. Έτσι κατά την εκτέλεση του εκτελέσιμου αρχείου test_octree δημιουργείτε ένα αρχείο gmon.out το οποίο με την εντολή
gprof test_octree gmon.out > analysis.txt παράγει το αρχείο analysis.txt που περιέχει στατιστικά στοιχεία για τις συναρτήσεις του προγράμματος. Ενδεικτικά για
./test_octree 33554432 0 128 1 18 παίρνουμε τα εξής αποτελέσματα:

time %	
26.92	truncated_radix_sort
25.06	data_rearrangement
18.60	cmpfunc
6.56	morton_encoding
6.17	compute_hash_codes

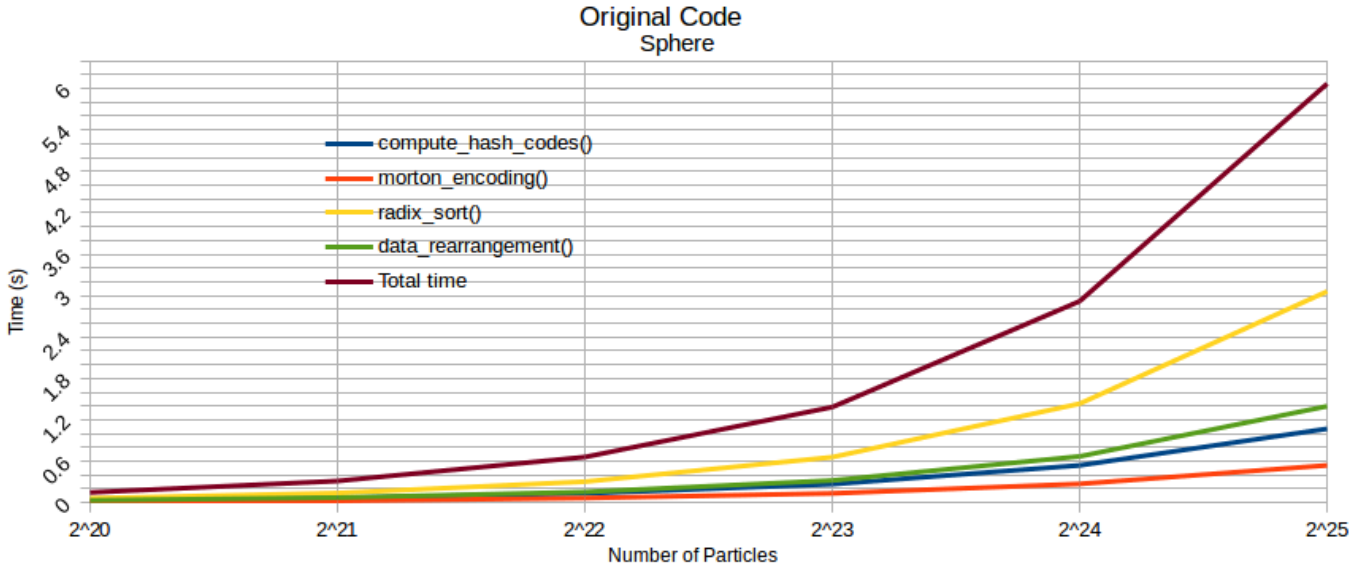
Καταλαβαίνουμε λοιπόν οτι είναι σημαντικό να παραλληλοποιηθούν οι συναρτήσεις truncated_radix_sort , data_rearrangement, morton_encoding και compute_hash_codes. Παρέλειψα την cmpfunc καθώς εμπεριέχεται στην συνάρτηση check_codes(), συνάρτηση που ελέγχει τα αποτελέσματα.

Επίσης βλέποντας(βλ. Επόμενη σελίδα) τους χρόνους που απαιτούνται για την εκτέλεση των συναρτήσεων αυτών, αυξανομένων των σωματιδίων , ενισχύεται η παραπάνω διαπίστωση.

Αποτελέσματα του αρχικού προγράμματος



Cube	2^20	2^21	2^22	2^23	2^24	2^25
compute_hash_codes()	0.033876	0.067402	0.134224	0.26886	0.541438	1.071983
morton_encoding()	0.017132	0.034259	0.067726	0.13571	0.278235	0.54205
radix_sort()	0.053847	0.106915	0.251457	0.540204	1.09834	2.464141
data_rearrangement()	0.033479	0.074086	0.154636	0.322839	0.684005	1.407852
Total time	0.138333	0.282662	0.608042	1.267613	2.602019	5.486026



Sphere	2^20	2^21	2^22	2^23	2^24	2^25
compute_hash_codes()	0.03382	0.067107	0.134125	0.268008	0.540787	1.070972
morton_encoding()	0.017079	0.033915	0.067648	0.1353	0.272447	0.540168
radix_sort()	0.063547	0.139381	0.304678	0.659739	1.434575	3.060055
data_rearrangement()	0.033406	0.073399	0.154708	0.321892	0.671037	1.397718
Total time	0.147852	0.313801	0.661158	1.384938	2.918846	6.068913

Βλέπουμε λοιπόν πως ,για αυξανόμενα σωματίδια, διαρκώς αυξάνονται οι χρόνοι που απαιτούνται για την εκτέλεση των συναρτήσεων αυτών, με την radix sort να επιβαρύνει περισσότερο απ’ όλες τον συνολικό χρόνο του προγράμματος.

Παραλληλοποίηση με Pthreads

Αρχικά και απαραίτητο βήμα ήταν η προσθήκη στα ζητούμενα απο τον χρήστη ορίσματα μίας ακόμα global int μεταβλητής NUM_THREADS η οποία αντιπροσωπεύει τον αριθμό των νημάτων που ζήτησε ο χρήστης . Η μεταβλητή αυτή ορίζεται μέσα στο αρχείο utils.h και της δίνουμε τιμή στην main του προγράμματος. Είναι προσπελάσιμη απ' όλες τις συναρτήσεις του προγράμματος χρησιμοποιώντας την εντολή
extern int NUM_THREADS .Επίσης σε κάθε παράλληλη συνάρτηση έκανα #include την βιβλιοθήκη pthread.h ώστε να έχω πρόσβαση στις συναρτήσεις της.

Η γενική διαδικασία που ακολούθησα για την παραλληλοποίηση των **compute_hash_codes, morton_encoding** και **data_rearrangement** είναι η εξής :

- Δημιουργία μιας δομής thread_args που περιλαμβάνει τα απαραίτητα ορίσματα για την κλήση της παράλληλης συνάρτησης.
- Δημιουργία της παράλληλης συνάρτησης όπου και ορίζονται τοπικές μεταβλητές στις οποίες μεταβιβάζω τις τιμές του ορίσματος *arg έπειτα από κατάλληλο cast type σε thread_args. Επίσης στην συνάρτηση αυτή εμπεριέχονται οι εντολές της original συνάρτησης.
- Αλλαγή της αρχικής συνάρτησης ώστε να ορίζει όσα threads έχουν ζητηθεί με την εντολή pthread_t threads[NUM_THREADS] , τα οποία ορίζονται JOINABLE,δηλαδή ότι πρέπει να περιμένει το πρόγραμμα να τερματίσουν όλα, μέσω των εντολών:
  ```
  pthread_attr_t myattr;
  pthread_attr_init(&myattr);
  pthread_attr_setdetachstate(&myattr, PTHREAD_CREATE_JOINABLE);
  ```
  Δημιουργία δομών thread_args για όσα threads έχουν ζητηθεί, καθώς και δύο πίνακες int size[NUM_THREADS] και int offset[NUM_THREADS] οι οποίοι αποσκοπούν στον διαμοιρασμό των εκάστοτε σωματιδίων σε τόσα κομμάτια όσα και τα ζητούμενα νήματα. Ο πίνακας size εμπεριέχει τον αριθμό των σωματιδίων που θα χειριστεί το κάθε νήμα, ενώ ο πίνακας offset εμπεριέχει το σημείο εκκίνησης δηλαδή το πρώτο σωματίδιο για το κάθε νήμα.
  Έπειτα μέσα σε ένα loop γίνεται κλήση της παράλληλης συνάρτησης NUM_THREAD φορές, με την εντολή
  `rc=pthread_create(&threads[i],&myattr,παράλληλη συνάρτηση,(void *)&args[i]);`
  αφού πρώτα έχει πάρει τιμές το όρισμα args[i].
  Ελέγχεται κάθε pthread_create αν εκτελέστηκε σωστά, αλλιώς εμφανίζει error και κάνει return .Τέλος με το πέρας του loop περιμένει το πρόγραμμα την επιστροφή όλων των δημιουργημένων νημάτων μέσω της εντολής
  `rc = pthread_join(threads[i], &status);` για την οποία πάλι γίνεται έλεγχος ορθής εκτέλεσης.

Η παραπάνω διαδικασία :

- Για την **compute_hash_codes** έγινε στην συνάρτηση quantize() και στην παράλληλη της void *thread_quantize(void *arg) με τιμές για την δομή args

```
args[i].X=&X[DIM*offset[i]];
args[i].codes=&codes[DIM*offset[i]];
args[i].N=size[i];
args[i].low=low;
args[i].step=step;
```

- Για την **morton_encoding** έγινε στην συνάρτηση morton_encoding και στην παράλληλη της void *thread_morton(void *arg) με τιμές για την δομή args

```
args[i].mcodes=&mcodes[offset[i]];
args[i].codes=&codes[DIM*offset[i]];
args[i].N=size[i];
args[i].max_level=max_level;
```

- Για την **data_rearrangement** έγινε στην συνάρτηση data_rearrangement() και την παράλληλη της void *thread_rearrangement(void *args) με τιμές για την δομή args

```
args[i].Y=&Y[offset[i]*DIM];
args[i].X=X;
args[i].permutation_vector=&permutation_vector[offset[i]];
args[i].N=size[i];
```

Στην συνέχεια περιγράφεται η υλοποίηση της παράλληλης radix_sort με χρήση Pthreads.

Αρχικά για τον παραλληλισμό της συνάρτησης **truncated_radix_sort** χρειάστηκε μια επιπλέον global μεταβλητή active_threads η οποία ορίστηκε στην utils.h και μηδενίζεται στην main του προγράμματος μας πριν την κλίση της συνάρτησης truncated_radix_sort. Η μεταβλητή αυτή σκοπό έχει να μας ενημερώνει για τα πόσα νήματα έχουν δημιουργηθεί κάθε στιγμή ώστε να αποφύγουμε την δημιουργία περισσότερων από αυτών που έχουν ζητηθεί. Επίσης όπως και στις προηγούμενες συναρτήσεις έχω δημιουργήσει μια δομή thread_arguments που περιέχει τα απαραίτητα ορίσματα για την κλίση της παράλληλης συνάρτησης.
Η γενική ιδέα της παραλληλοποιημένης έκδοσης της radix_sort είναι μία **εμφωλευμένη αναδρομική παραλληλοποίηση**, δηλαδή κάθε νήμα μπορεί να δημιουργήσει με την σειρά του καινούρια νήματα, μέχρις ότου η μεταβλητή active_threads φτάσει το όριο των νημάτων που εισάχθηκαν από τον χρήστη ,καλώντας την συνάρτηση αναδρομικά.

Πιο συγκεκριμένα την πρώτη φορά που θα κληθεί η truncated_radix_sort εκτελεί σειριακά το κομμάτι που υπολογίζει του πίνακες των ψηφίων του κώδικα morton, κανει τα αντίστοιχα swap και υπολογίζει τις μεταβλητές της δομής args[i], για args[MAXBINS] μέσα σε ένα loop , όπως κάνει και το αρχικό μας πρόγραμμα. Ωστόσο σε αυτό το σημείο

δεν καλεί ξανά την truncated_radix_sort αλλά ελέγχει πρώτα αν τα νήματα που επέλεξε ο χρήστης είναι λιγότερα από MAXBINS δηλαδή 8 και αν είναι τότε η μεταβλητή active_threads παίρνει την τιμή NUM_THREADS, αλλιώς αν τα νήματα που ζήτησε ο χρήστης είναι περισσότερα από 8 παίρνει την τιμή MAXBINS. Αυτό το κάνω ώστε να δημιουργήσω έως και 8 αρχικά νήματα πριν αρχίσουν τα υπόλοιπα νήματα να δημιουργούν εμφωλευμένα καινούρια νήματα. Δηλαδή αν ζητήσει ο χρήστης π.χ. 2 threads θα δημιουργηθούν 2 αρχικά threads και κανένα εμφωλευμένο. Ενώ αν ζητήσει π.χ. 16 thread θα δημιουργηθούν 8 αρχικά threads και 8 εμφολευμένα.

Έπειτα μέσα σε μία for(i=0;i<MAXBINS;i++) ελέγχεται κάθε φορά αν τα αρχικά νήματα που έχουν δημιουργηθεί είναι λιγότερα από active_threads και τότε καλεί την παράλληλη συνάρτηση δημιουργώντας ένα καινούριο νήμα με την εντολή
rc=pthread_create(&thread[i],&myattr,parallel_radix_sort,(void*)&args[i]);
ενώ όταν φτάσουμε τον περιορισμό των active_threads ή 8 αρχικών νημάτων, η συνάρτηση αυτή καλείται σειριακά με την εντολή parallel_radix_sort((void *)&args[i]) .
Έπειτα μέσα στην truncated_radix_sort περιμένουμε να επιστρέψουν τα αρχικά νήματα που δημιουργήθηκαν με την εντολή rc=pthread_join(thread[i],&status) .

Στο κομμάτι της παράλληλης **parallel_radix_sort** κάθε νήμα εκτελεί εκ νέου τις εντολές που υπολογίζουν τους πίνακες των ψηφίων του εκάστοτε morton_code,για τα δικά του ορίσματα το καθένα, έως το κομμάτι της αναδρομικής κλίσης. Όταν φτάσει στο σημείο αυτό, μέσα σε μία for(i=0,i<MAXBINS;i++) ελέγχει αν τα active_threads είναι μικρότερα από τα νήματα που ζήτησε ο χρήστης και τότε δημιουργεί ένα καινούριο νήμα με την εντολή rc=pthread_create(&thread[thread_id],&myattr,parallel_radix_sort,(void *)&new_args[i])
Αλλιώς την καλεί σειριακά με την εντολή parallel_radix_sort((void *)&new_args[i]) .

Για την **ακεραιότητα και την αποφυγή λαθών** χρησιμοποίησα τις εντολές
```
pthread_mutex_lock(&mymutex);
thread_id=active_threads;
active_threads++;
pthread_mutex_unlock(&mymutex);
```

οι οποίες επιτρέπουν σε ένα νήμα την φορά να εκτελέσει της εντολές ανάμεσα στο lock και unlock. Ο ορισμός του mutex έγινε με global μεταβλητή, ενώ το initiallization του έγινε μέσα στην truncated_radix_sort.

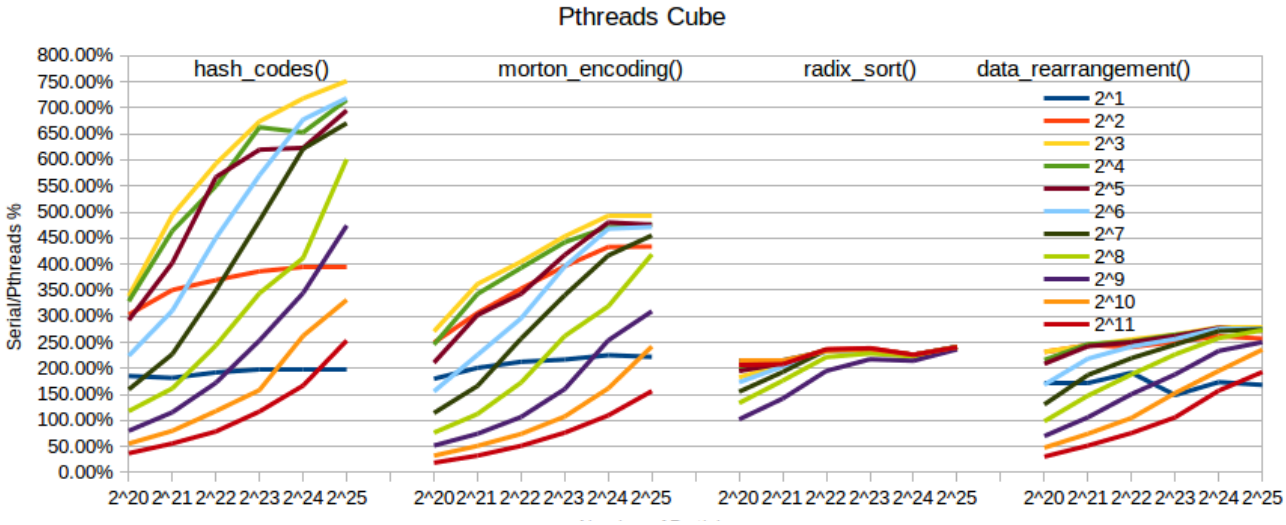
Τα εμφωλευμένα νήματα που δημιουργούνται ενώνονται μέσα στο loop όπου δημιουργήθηκαν με την εντολή rc=pthread_join(thread[thread_id],NULL). Το pthread_t *thread ορίσθηκε ως global μεταβλητή για να έχουν πρόσβαση και τα εμφωλευμένα νήματα και για αυτό το λόγο χρησιμοποιήθηκε malloc. Με το τερματισμό της truncated_radix_sort γίνεται free(thread) για την απελευθέρωση της δεσμευμένης μνήμης.

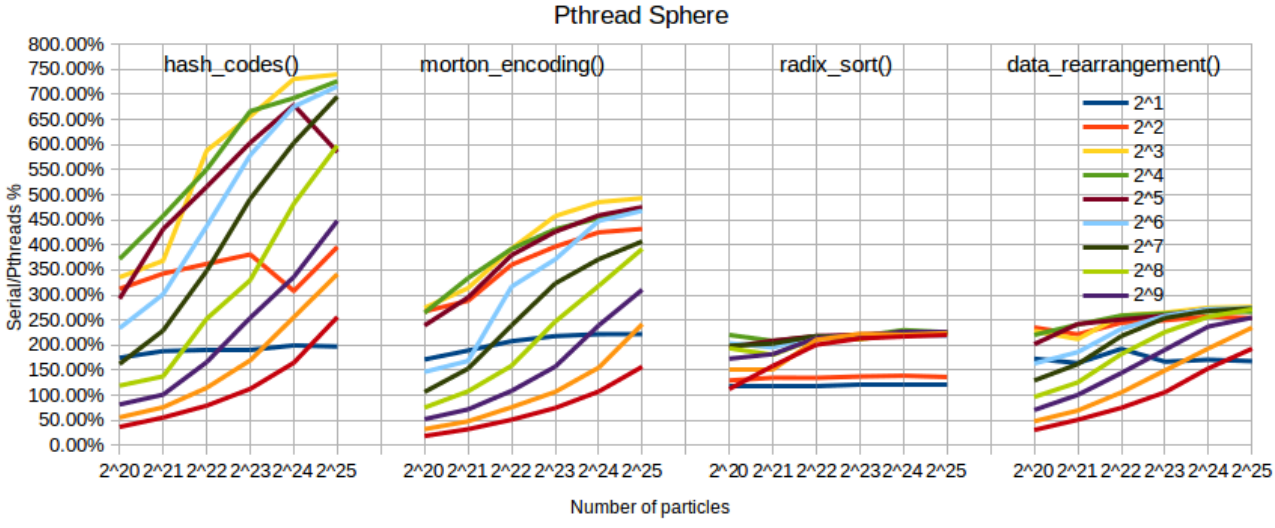
Τα ορίσματα της **args[i]** στην περίπτωση της radix είναι τα
args[i].morton_codes=&morton_codes[offset];
args[i].sorted_morton_codes=&sorted_morton_codes[offset];
args[i].permutation_vector=&permutation_vector[offset];
args[i].index=&index[offset];
args[i].level_record=&level_record[offset];
args[i].N=size;
args[i].population_threshold=population_threshold;
args[i].sft=sft-3;
args[i].lv=lv+1;

ενώ τα offset και size είναι τα ίδια με αυτά που ορίζονται και στο αρχικό πρόγραμμα που δόθηκε.

Το πρόγραμμα έγινε compile με την εντολή :
gcc -O3 -pthread -std=gnu99

Αποτελέσματα υλοποίησης Pthreads



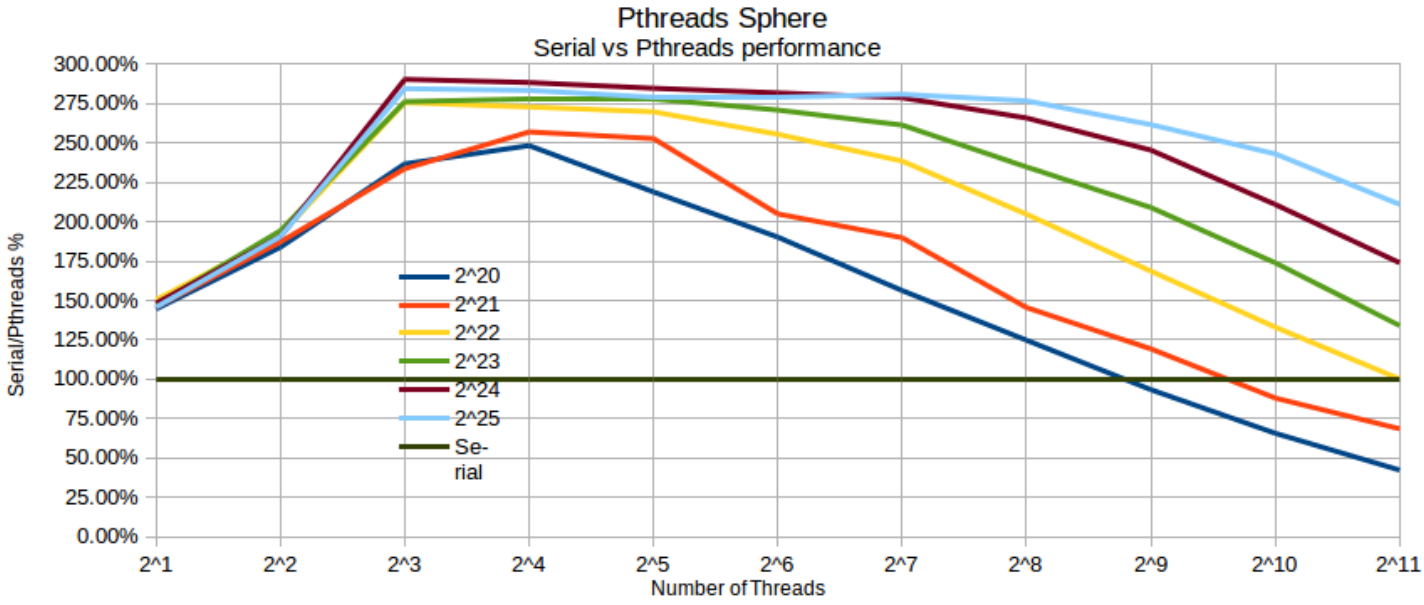


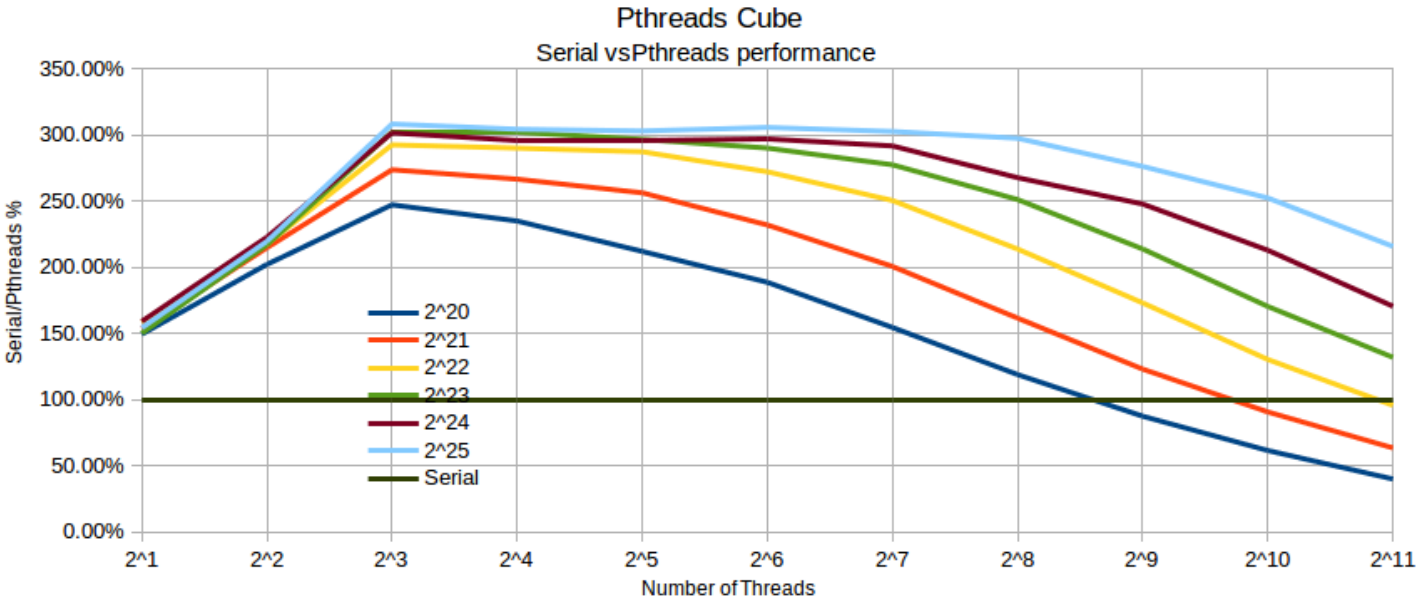
Παρατηρούμε πως την δραματικότερη βελτίωση στους χρόνους εκτέλεσης, σε σχέση με το σειριακό πρόγραμμα, έχει η compute_hash_codes με έως και 750% καλύτερη απόδοση. Ακολουθεί η morton_encoding με μέγιστη βελτίωση της τάξης 500%, έπειτα η data_rearrangement με μέγιστη τα 275% ενώ μικρότερη βελτίωση βλέπουμε για την truncated_radix sort με μέγιστη 240% αλλά ιδιαίτερα σταθερή συμπεριφορά. Η χαμηλότερη βελτίωση της radix από τις υπόλοιπες, δεν πρέπει να μας ξεγελάει καθώς είναι η συνάρτηση που επιβαρύνει περισσότερο το πρόγραμμα (έχει τους μεγαλύτερους χρόνους εκτέλεσης) και αυτή η μικρή της βελτίωση έχει τα σημαντικότερα αποτελέσματα στον συνολικό χρόνο εκτέλεσης. Για να το καταλάβουμε καλύτερα αυτό ας δούμε τους χρόνους για 2^3 νήματα :

Number of Particles	2^20	2^21	2^22	2^23	2^24	2^25
original radix	0.053847	0.106915	0.251457	0.540204	1.09834	2.464141
pthread radix	0.025147	0.049785	0.107914	0.227987	0.486079	1.021831
time difference (s)	**0.0287**	**0.05713**	**0.143543**	**0.312217**	**0.612261**	**1.44231**

Number of Particles	2^20	2^21	2^22	2^23	2^24	2^25
original hash_codes	0.033876	0.067402	0.134224	0.26886	0.541438	1.071983
Pthread hash_codes	0.010005	0.013649	0.022647	0.03989	0.075409	0.142673
time difference (s)	**0.023871**	**0.053753**	**0.111577**	**0.22897**	**0.466029**	**0.92931**

Γίνεται κατανοητό οπότε ότι σε απόλυτα νούμερα η παραλληλοποίηση της radix_sort εξοικονομεί περισσότερο χρόνο απο την hash_codes.

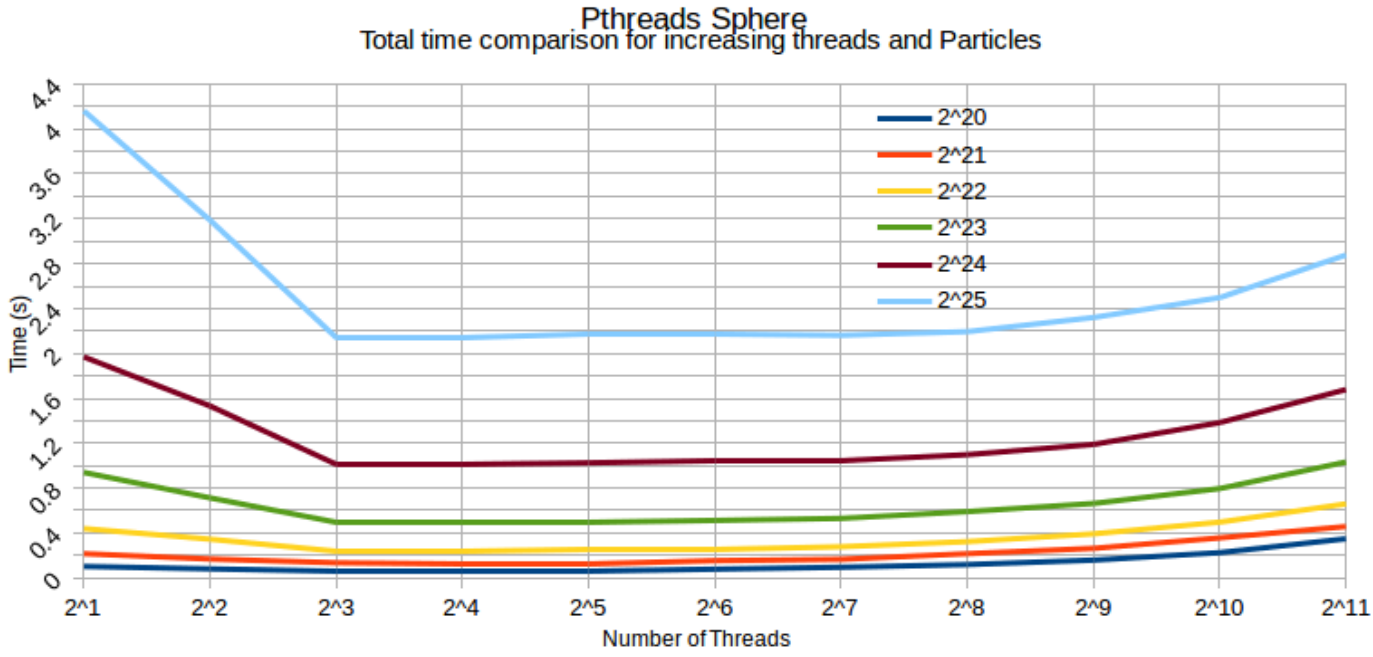




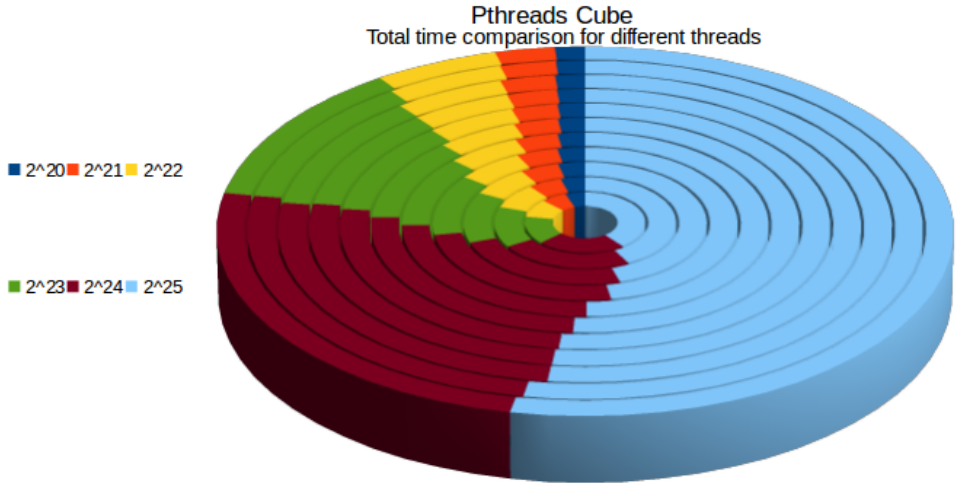
Στα τελευταία δύο διαγράμματα βλέπουμε την επίδοση της υλοποίησης με Pthreads σε σχέση με τον σειριακό κώδικα χρησιμοποιώντας τους συνολικούς χρόνους των συναρτήσεων που παραλληλοποιήσαμε.

Παρατηρούμε μία καταπληκτική αύξηση των επιδόσεων του προγράμματός μας σε σχέση με το σειριακό πρόγραμμα , η οποία φτάνει να είναι έως και **308,8% γρηγορότερη** από αυτή του σειριακού!

Γενικά παρατηρούμε πως μόνο για μεγάλο αριθμό νημάτων(2^9 και πάνω) και μικρό αριθμό σωματιδίων(2^21 και κάτω) ο χρόνος μας είναι χειρότερος απο του σειριακού.



Καταλαβαίνουμε επίσης ότι για 2^3 νήματα παίρνουμε το καλύτερο αποτέλεσμα, από εκεί και πέρα όμως όσο αυξάνονται τα νήματα χειροτερεύουν οι χρόνοι εκτέλεσης ενώ τα χειρότερα αποτελέσματα τα παίρνουμε 2^25 σωματίδια και 2^1 νήματα. Αυτό εξηγείται από το γεγονός ότι κάθε pthread_create επιβαρύνει τον τελικό χρόνο, όπως και οι επιπλέον εντολές που χρειαστήκαμε για την υλοποίηση αυτή. Άλλωστε είναι φυσικό η παράλληλη υλοποίηση να συμφέρει για μεγάλο αριθμό δεδομένων, καθώς και ο αριθμός των νημάτων να μη ξεπερνάει κατα πολύ τους πυρήνες του συστήματος . Δηλαδή πρέπει κανείς να βρει την χρυσή τομή. Για το server diades φαίνεται ότι αυτή είναι τα 8 νήματα, καθώς έχει και 8 πυρήνες!



Σχόλια: Δοκίμασα στην radix_sort κάθε φορά που ένα νήμα επιστρέφει να μειώνω τα active_threads, ώστε να μπορούν να δημιουργηθούν νέα νήματα. Ωστόσο η υλοποίηση αυτή θα είχε χειρότερους χρόνους καθώς δημιουργούνται συνολικά πάρα πολλά νήματα και η εντολή pthread_create επιβαρύνει σημαντικά τον χρόνο εκτέλεσης. Ενδεικτικά είδα πως μειώνοντας τα active_threads κάθε φορά που τελείωνε ένα νήμα, για 2^25 σωματίδια και 2^11 threads δημιουργήθηκαν συνολικά 1.521.880 νήματα εκτοξέυοντας τον συνολικό χρόνο εκτέλεσης στα 7 δευτερόλεπτα! Μία τέτοια υλοποίηση με Pthreads θα ήταν ασύμφορη, ειδικά για μεγάλο αριθμό νημάτων.

Παραλληλοποίηση με Cilk

Για την παραλληλοποίηση του προγράμματος με Cilk διάλεξα να χρησιμοποιήσω την εντολή **cilk_spawn** καθώς παρατήρησα καλύτερα αποτελέσματα απ' ότι με την cilk_for. Επειδή η τεχνική παραλληλοποίησης που χρησιμοποίησα για την cilk είναι παρόμοια με την τεχνική που ανέλυσα προηγουμένως για τα Pthreads θα εστιάσω στις ουσιαστικές διαφορές.

Οι απαραίτητες βιβλιοθήκες για την υλοποίηση αυτή είναι οι cilk.h και cilk_api.h. Προκειμένου να ορίσουμε τον αριθμό των workers (αντίστοιχα threads) στην cilk, αφότου οριστούν από τον χρήστη, χρειάζονται οι παρακάτω εντολές.

```
char str[3];
sprintf(str,"%d",NUM_THREADS);
__cilkrts_end_cilk();
__cilkrts_set_param("nworkers",str);
```

Μάλιστα οι δύο τελευταίες χρειάζεται να μπουν πριν απο κάθε παράλληλη συνάρτηση, όπως αναφέρεται στο site της Intel.

Για την παραλληλοποίηση των **compute_hash_codes**, **morton_encoding** και **data_rearrangement** η λογική είναι ίδια με των pthreads, δηλαδή δημιουργώ μία δομή που περιλαμβάνει τις απαραίτητες μεταβλητές για την κλίση της παράλληλης συνάρτησης. Δημιουργώ επίσης μια παράλληλη συνάρτηση που παίρνει σαν όρισμα την δομή αυτή και περιέχει τις εντολές που χρησιμοποιούνται και στην αρχική συνάρτηση. Στην αρχική συνάρτηση χωρίζω με τους πινάκες size και offset τα σωματίδια με τα οποία θα ασχοληθεί κάθε νήμα . Υπολογίζω τα ορίσματα args[NUM_THREADS] και καλώ την παράλληλη συνάρτηση μέσα σε ένα loop με την εντολή
cilk_spawn ονομα_παράλληλης((void*)&args[i])
Τέλος χρησιμοποίησα την εντολή **cilk_sync** μετά το loop αυτό για να περιμένει το πρόγραμμα να τελειώσουν όλα τα δημιουργημένα νήματα.

Για την συνάρτηση **truncated_radix_sort** χρειάστηκα μια global μεταβλητή active_threads και την χρήση pthread_mutex_lock και unlock. Όταν κληθεί η συνάρτηση αυτή , τρέχει την πρώτη φορά μέχρι το σημείο που χωρίζει την αναδρομή σε MAXBINS κομμάτια. Τότε εξετάζει αν active_threads<NUM_threads και δημιουργεί αν είναι δυνατό ένα καινούριο νήμα την φορά με την εντολή

```
cilk_spawn truncated_radix_sort(&morton_codes[offsets[i]],&sorted_morton_codes[offsets[i]],
                                &permutation_vector[offsets[i]],&index[offsets[i]],
                                &level_record[offsets[i]],sizes[i],population_threshold,sft-3, lv+1);
```

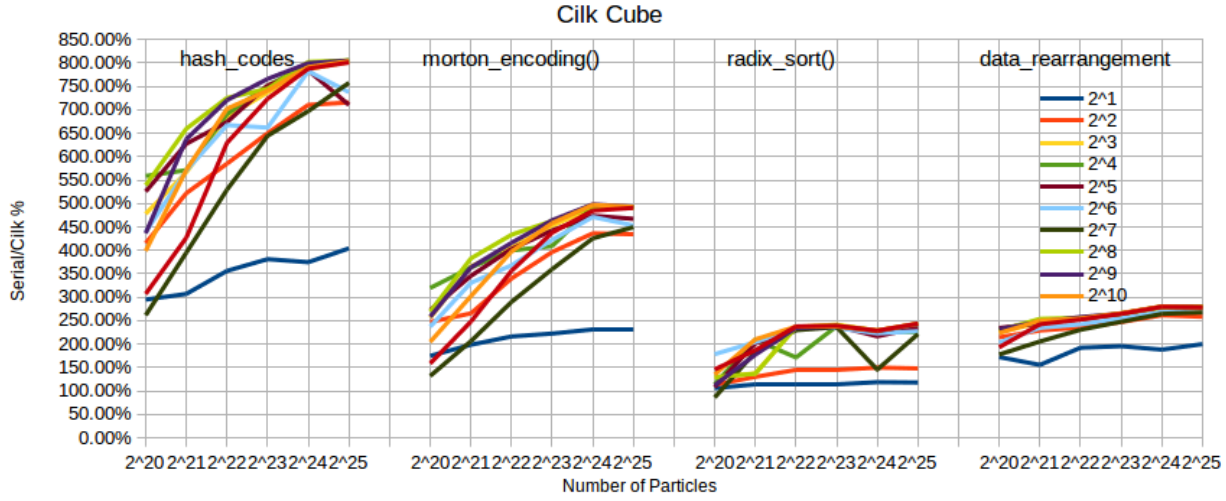
ενώ προηγουμένως έχω αυξήσει τον αριθμό των active_threads κατά ένα την φορά χρησιμοποιώντας mutex lock και unlock . Διαφορετικά καλεί σειριακά την συνάρτηση truncated_radix_sort με τα κατάλληλα πάντα ορίσματα. Τέλος εξωτερικά της for με την **cilk_sync** περιμένει το πρόγραμμα να επιστρέψουν όλα τα δημιουργημένα νήματα.

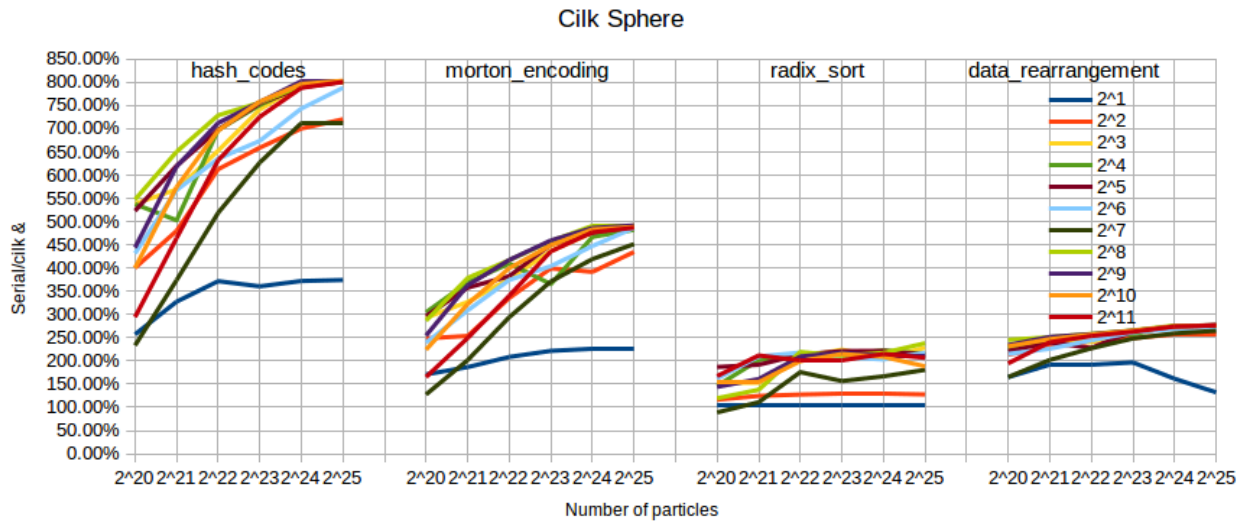
Για να γίνει compile το πρόγραμμα με cilk χρησιμοποίησα την εντολή

icc -O3 -lcilkrts -pthread -ldl -std=gnu99

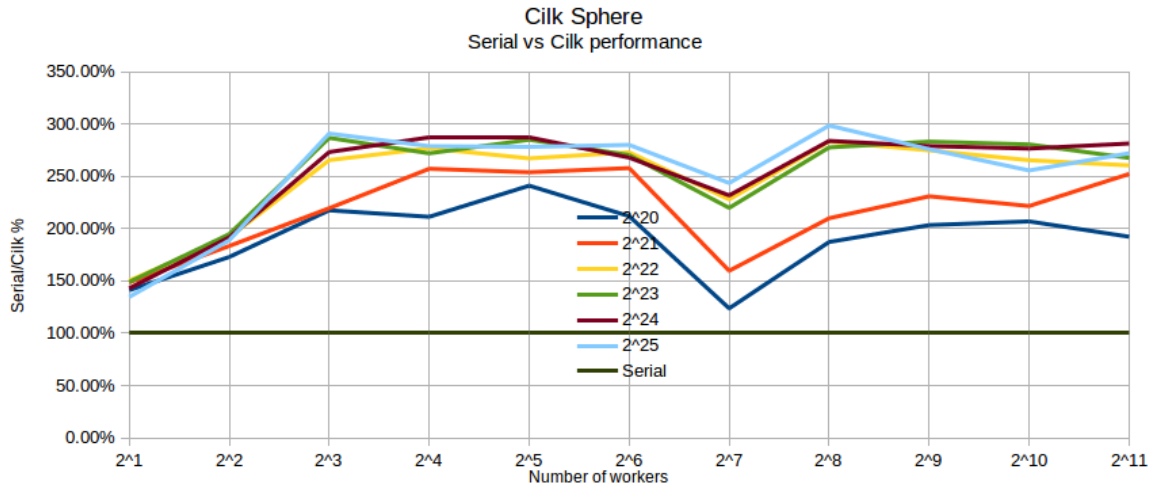
Σχόλια: Δοκίμασα την ίδια υλοποίηση με reducers αντί για mutexes αλλα τα αποτελέσματα ήταν ελαφρός επιβαρυμένα. Επίσης όπως θα δούμε και στα αποτελέσματα της υλοποίησης, η cilk υστερεί σε ένα κομμάτι. Δεν επιτρέπει στον χρήστη να υπερβεί πάνω το όριο των 16*(πυρήνες του συστήματος) workers. Το όριο αυτό για το server diades είναι τα 128 workers καθώς έχει 8 πυρήνες. Όταν ζητήσουμε περισσότερα νήματα απο αυτόν τον αριθμό, αυτόματα το πρόγραμμα μας τρέχει για την default τιμή των 8 workers. Το γεγονός αυτό μπορεί να το παρατηρήσει κανείς στα διαγράμματα αφού για τιμές πάνω απο 2^7 workers παρατηρούμε μια σταθερή συμπεριφορά ανάλογη αυτής για 2^3 workers. Πέρα απο τα διαγράμματα μπορεί κάποιος να ελέγξει αυτόν τον περιορισμό με την εντολή __cilkrts_get_nworkers() που επιστρέφει τον αριθμό των workers που έχουν τεθεί στο api της cilk. Αυτός ο περιορισμός ισχύει και για την υλοποίηση με cilk_for .

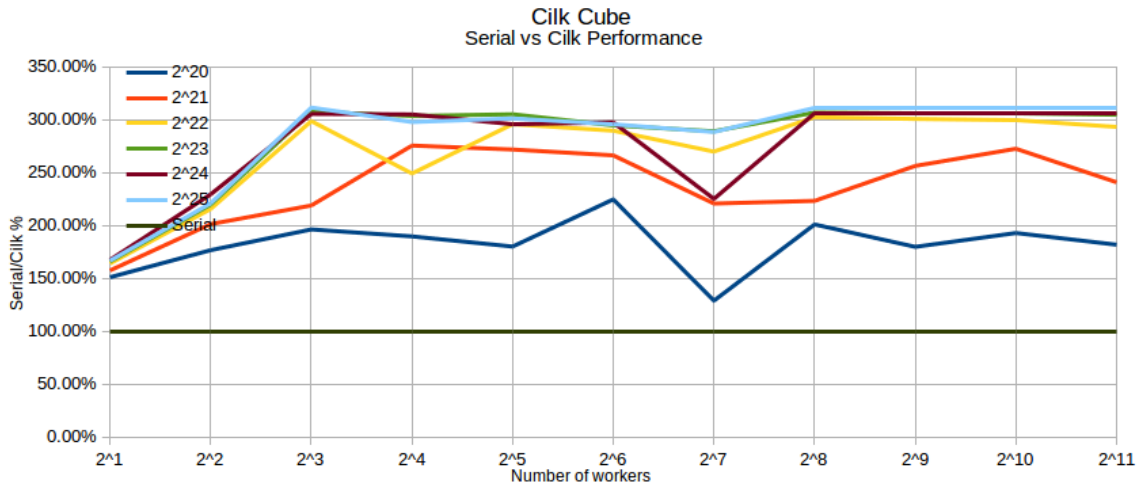
Αποτελέσματα υλοποίησης με Cilk



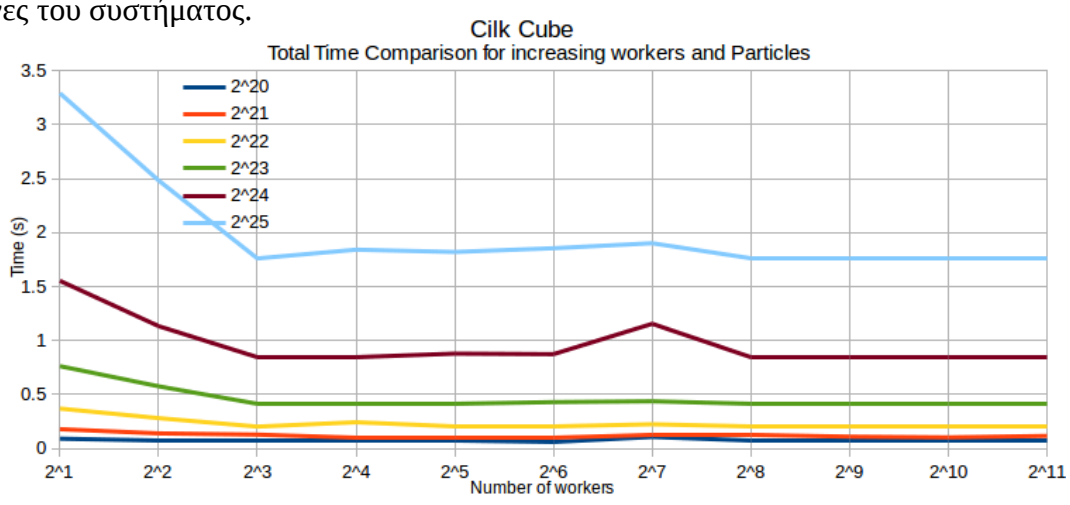


Παρόμοια συμπεριφορά με την υλοποίηση των Pthreads παρατηρούμε και στην Cilk. Αξίζει να σημειωθεί πως για καμία τιμή δεν παίρνουμε χειρότερα αποτελέσματα από το αρχικό πρόγραμμα ενώ και εδώ η βελτίωση που παρατηρούμε ποσοστιαία φτάνει για την hash_codes το 800%, για την morton_encoding το 500%, για την data_rearrangement το 280% ενώ για την radix_sort το 250% που όπως εξήγησα είναι σημαντική βελτίωση.





Στα δύο προηγούμενα διαγράμματα βλέπουμε την συνολική βελτίωση της υλοποίησης μας με cilk σε σχέση με τον σειριακό κώδικα. Η βελτίωση των χρόνων φτάνει μέχρι και 311% ενώ ποτέ δεν παίρνουμε χειρότερα αποτελέσματα απο το αρχικό πρόγραμμα. Επίσης όπως σχολίασα προηγουμένως φαίνεται και από τα διαγράμματα πως έπειτα από 2^7 workers η συμπεριφορά της cilk είναι ίδια με 2^3 workers, καθώς υπάρχει ο περιορισμός από την cilk των max workers 16*πυρήνες του συστήματος.



Παραλληλοποίηση με OpenMp

Για την παράλληλη υλοποίηση με OpenMp χρειάζεται να κάνουμε #include την βιβλιοθήκη omp.h .
Και εδώ η παραλληλοποίηση των συναρτήσεων **compute_hash_codes , morton_encoding** και **data_rearrangement** είναι όμοια και αρκετά εύκολη οπότε τις αναφέρω μαζί.
Η τεχνική που χρησιμοποιήθηκε για την παραλληλοποίηση των συναρτήσεων αυτών είναι η εξής :

Αρχικά έχω δηλώσει μια global μεταβλητη NUM_THREADS την οποία ορίζει ο χρήστης. Για όποιες for έχει νόημα να τις παραλληλοποιήσουμε χρησιμοποιούμε την ακόλουθη εντολή αφότου δηλώσουμε το size=N/NUM_THREADS

```
#pragma omp parallel for private(i) num_threads(NUM_THREADS) schedule(dynamic,size)
```

Στην ουσία με αυτή την εντολή παραλληλοποιείται η for για NUM_THREADS κομμάτια , που το καθένα έχει αναλάβει N/NUM_THREADS σωματίδια. Με την εντολή private() δηλώνουμε ποιες μεταβλητές είναι ατομικές , ενώ αν χρειαστεί μπορούμε να βάλουμε και shared() μεταβλητές. Η εντολή schedule(dynamic) ορίζει ότι εφόσον ένα νήμα έχει τελειώσει μπορεί να εργαστεί πάνω σε κάποιο ελεύθερο κομμάτι. Βέβαια η τελευταία εντολή δεν έχει και πολύ νόημα καθώς έτσι και αλλιώς έχουμε ισομοιράσει την δουλειά για κάθε νήμα. Αν δεν χρησιμοποιήσουμε το schedule, αυτόματα το OpenMp ισομοιράζει την for για όσα num_threads() έχουμε επιλέξει.

Για την data_rearrangement χρησιμοποιήθηκε αυτή κάθε αυτή η εντολή για την for που περιέχει την εντολή memcpy.
Για την compute_hash_codes χρησιμοποιήθηκε η εντολή
```
#pragma omp parallel for  private(i, j) shared(X, low, step) num_threads(NUM_THREADS)
```
πάνω από την for που περιέχεται στην συνάρτηση quantize. Εδώ χρησιμοποιήθηκε και η εντολή shared() καθώς έτσι παίρνουμε καλύτερα αποτελέσματα.
Ενώ για την morton_encoding χρησιμοποιήθηκε η εντολή
```
#pragma omp parallel for shared(mcodes, codes) private(i)
num_threads(NUM_THREADS)schedule(dynamic, size)
```
για την for που βρίσκεται μέσα στην morton_encoding.

Για την παραλληλοποίηση της **truncated_radix_sort** χρειάστηκε να ορίσω μια global μεταβλητή active_threads όπως και στις υπόλοιπες υλοποιήσεις της radix sort. Ωστόσο η υλοποίηση διαφέρει σαν ιδέα καθώς για κάθε recursive διαδικασία και όσο δεν ξεπερνάμε το όριο των active_threads σπάμε την for(i=0;i<MAXBINS;i++) που κάνει την αναδρομική κλήση της συνάρτησης σε δύο κομμάτια δημιουργώντας δύο καινούρια νήματα, αλλιώς την καλούμε σειριακά. Επίσης κάθε φορά που κάποιο νήμα επιστρέφει μειώνουμε τον αριθμό

των ενεργών νημάτων κατά ένα ώστε να δημιουργήσουμε καινούρια.
Πιο συγκεκριμένα αφού κληθεί η truncated_radix_sort και εκτελέσει όλες τις αρχικές εντολές μέχρι και τα swap() , υπολογίζω τους πίνακες offsets[MAXBINS] και sizes[MAXBINS] που είναι οι αντίστοιχες τιμές των offset και size που υπάρχουν στο αρχικό πρόγραμμα, μόνο που πλέον είναι σε μορφή πινάκων. Έπειτα ελέγχεται αν active_threads<NUM_THREADS και τότε μια μεταβλητή NEW_THREAD παίρνει την τιμή 1, αλλιώς την τιμή 0.
Ελέγχουμε την τιμή της μεταβλητής NEW_THREAD και αν είναι 1 ορίζουμε μία παράλληλη περιοχή 2 νημάτων με την εντολή
#pragma omp parallel num_threads(2)
{

μέσα στην οποία ο αριθμός των active_threads αυξάνεται κατα ενα απο κάθε νήμα χρησιμοποιώντας πρώτα την εντολή #pragma omp atomic ώστε να την εκτελεί ένα νήμα την φορά.

Υπολογίζεται το int size=MAXBINS/2 και με την εντολή
#pragma omp for private(i) schedule(dynamic,size) χωρίζουμε την for(i=0;i<MAXBINS;i++) σε δύο κομμάτια, ένα για κάθε νήμα που δημιουργήσαμε. Μέσα στην for αυτή γίνεται η αναδρομική κλήση της

```
truncated_radix_sort(&morton_codes[offsets[i]],
    &sorted_morton_codes[offse ts[i]],&permutation_vector[offsets[i]],
    &index[offsets[i]],&level_record[offsets[i]],sizes[i],
    population_threshold, sft-3, lv+1);
```

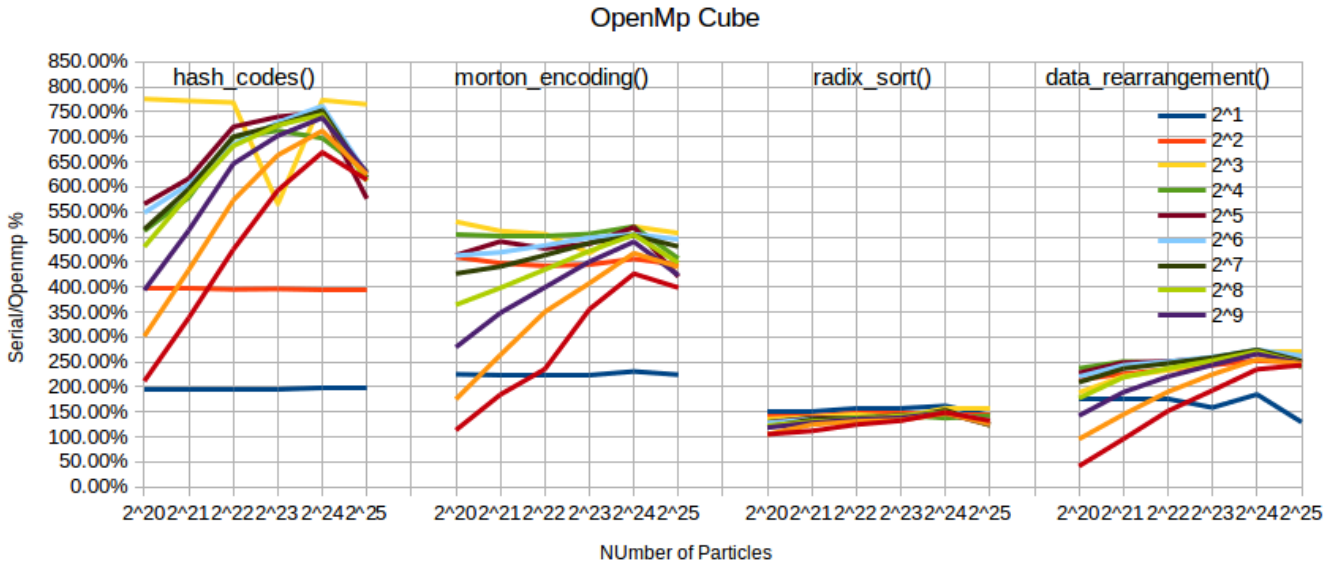
Ενώ έξω από την for κάθε νήμα που επιστρέφει μειώνει τα active_threads κατά ένα πάλι με την χρήση της #pragma omp atomic.
}
Από την άλλη, αν το NEW_THREAD είναι 0 , τότε καλούμε σειριακά την truncated_radix_sort για MAXBINS φορές με τα κατάλληλα ορίσματα όπως πρίν.
Κάθε φορά που αλλάζουν τα active_threads ή το NEW_THREAD συγχρονίζουμε τις μεταβλητές με την εντολή
#pragma omp flush(active_threads) ή NEW_THREAD αντίστοιχα ώστε κάθε στιγμή όλα τα νήματα να βλέπουν την ίδια τιμή.

Για το compile της υλοποίησης αυτής χρησιμοποιήθηκε η εντολή

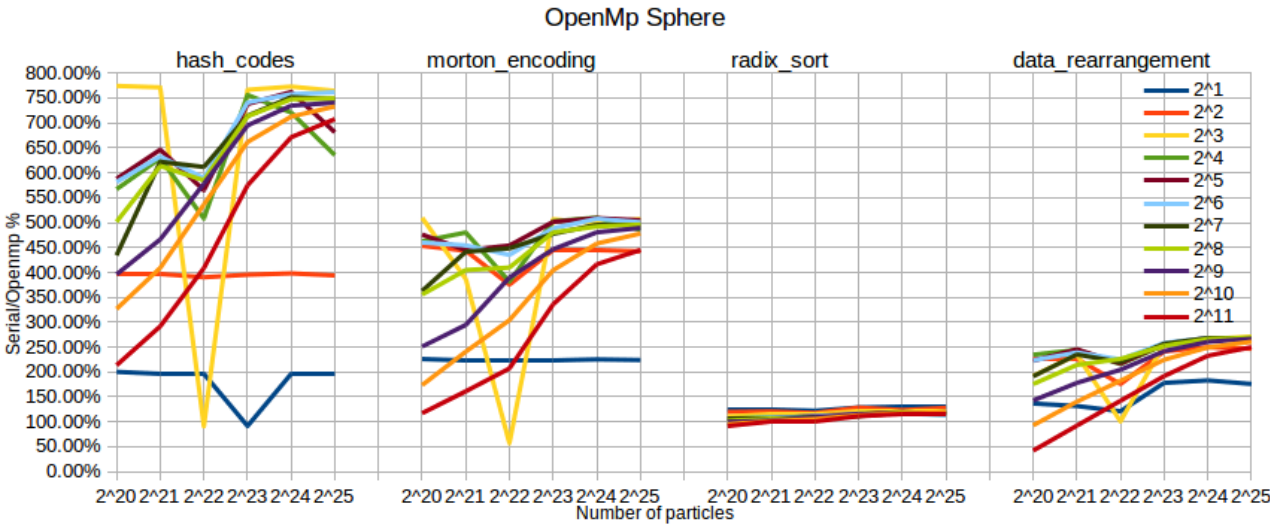
gcc -O3 -fopenmp -std=gnu99

Σχόλιο: Η συγκεκριμένη υλοποίηση παρότι επιτρέπει την δημιουργία πολλών περισσότερων συνολικών νημάτων, δίνει πολύ καλά αποτελέσματα χωρίς επιπλέον επιβάρυνση. Εν αντιθέσει με των Pthreads που όπως περιγράψαμε προηγουμένως, για παρόμοια υλοποίηση δεν δίνουν καλά αποτελέσματα. Αυτό ίσως οφείλεται στην καλύτερη διαχείριση της δημιουργίας νέων νημάτων απο το OpenMp, χωρίς να επιβαρύνει σημαντικά το σύστημα όπως η pthread_create().

Αποτελέσματα υλοποίησης OpenMp



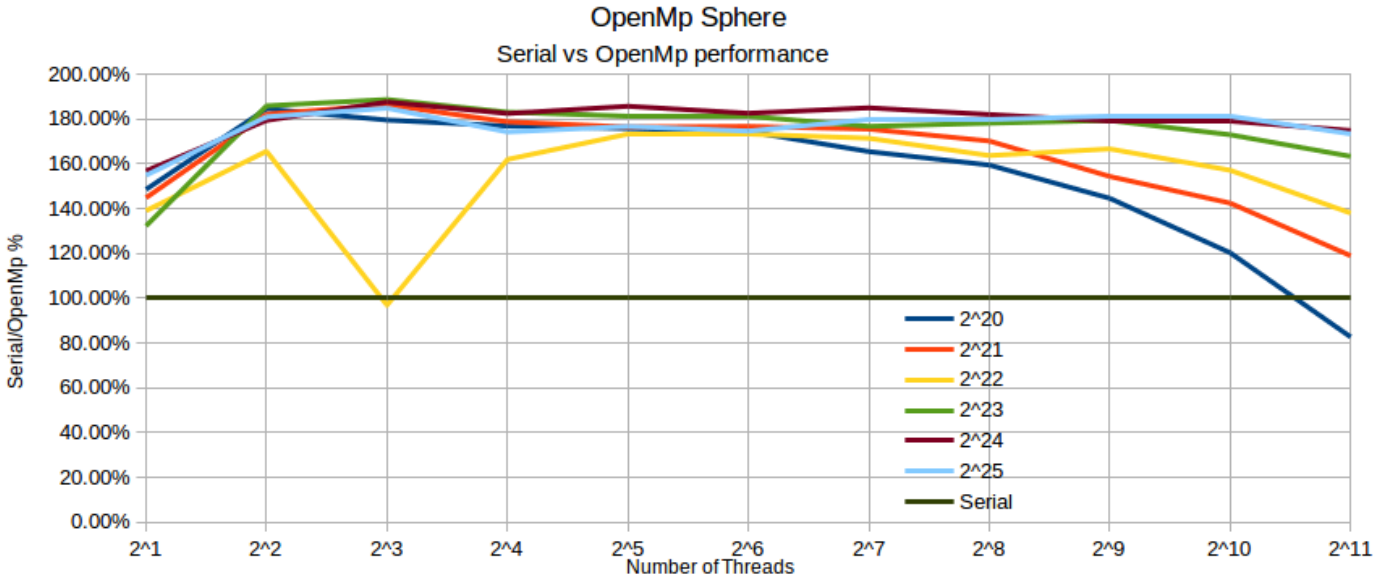
Ανάλογη είναι και η συμπεριφορά της υλοποίησης με Openmp. Παρατηρούμε βελτίωση της compute_hash_codes εώς και 775%, της morton_encoding εώς και 530% , της data_rearrangement εώς και 280% και της radix 160%.

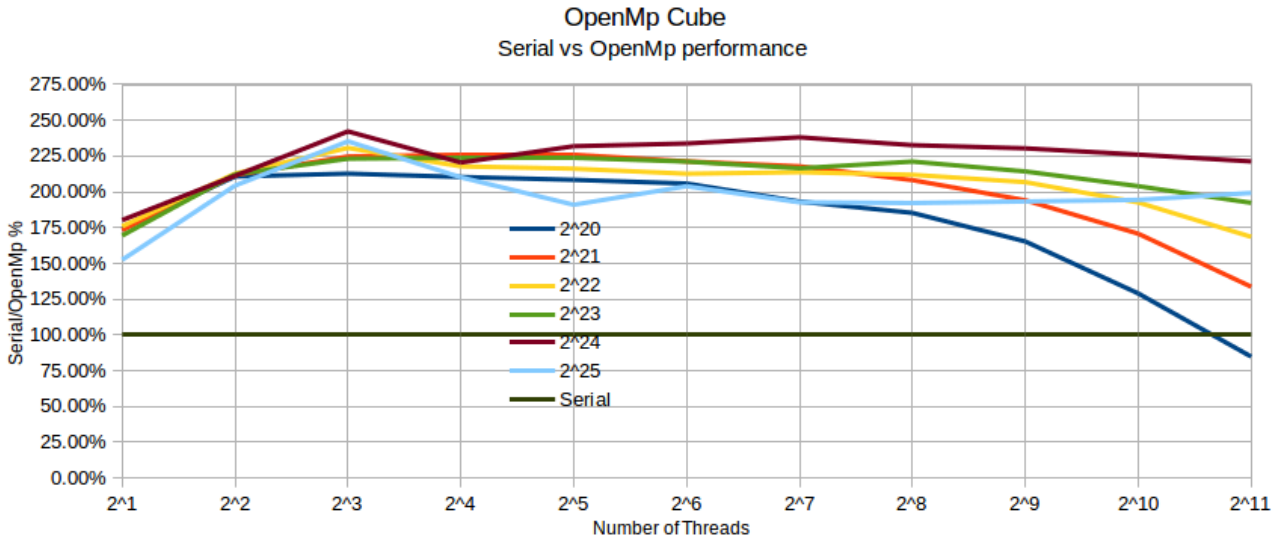


Παράλληλα παρατηρούμε στο διάγραμμα για σφαίρα και μερικές απρόσμενες μεταβολές για 2^3 νήματα, όπου θα περιμέναμε την καλύτερη βελτίωση. Αυτό ασφαλώς οφείλεται σε υπερφόρτωση του server την συγκεκριμένη χρονική περίοδο που το πρόγραμμα μας έτρεχε για 2^3 νήματα και 2^22 με 2^23 σωματίδια (με το runtests.sh αυτά εκτελούνται στην σειρά)!

Εκτελώντας ξανά το πρόγραμμα για αυτές τις τιμές σε άλλη χρονική στιγμή παίρνουμε πολύ καλύτερα αποτελέσματα.

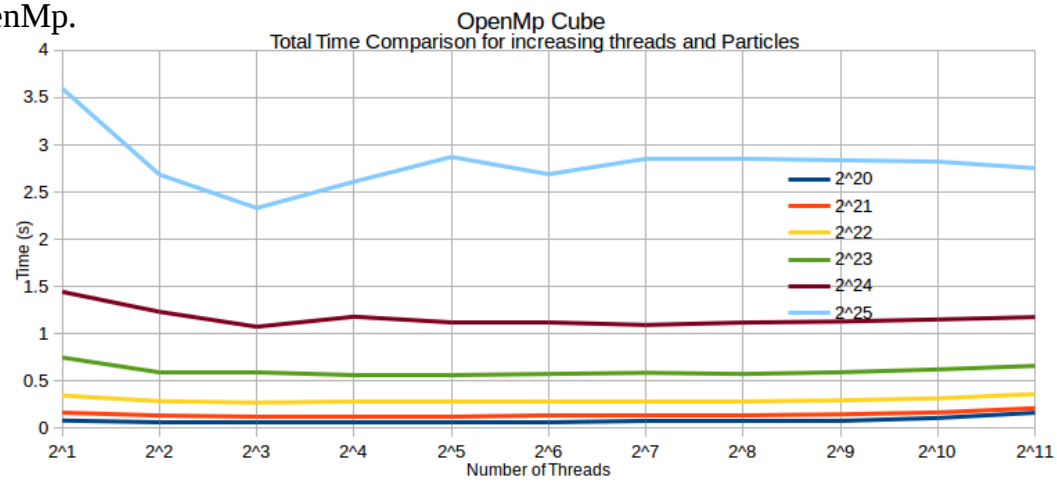
Serial/Parallel %	2^22	2^23
compute_hash_codes()	768.05%	768.83%
morton_encoding()	505.83%	507.67%
radix_sort()	148.04%	142.52%
data_rearrangement()	245.94%	253.59%
Total time	230.67%	223.15%



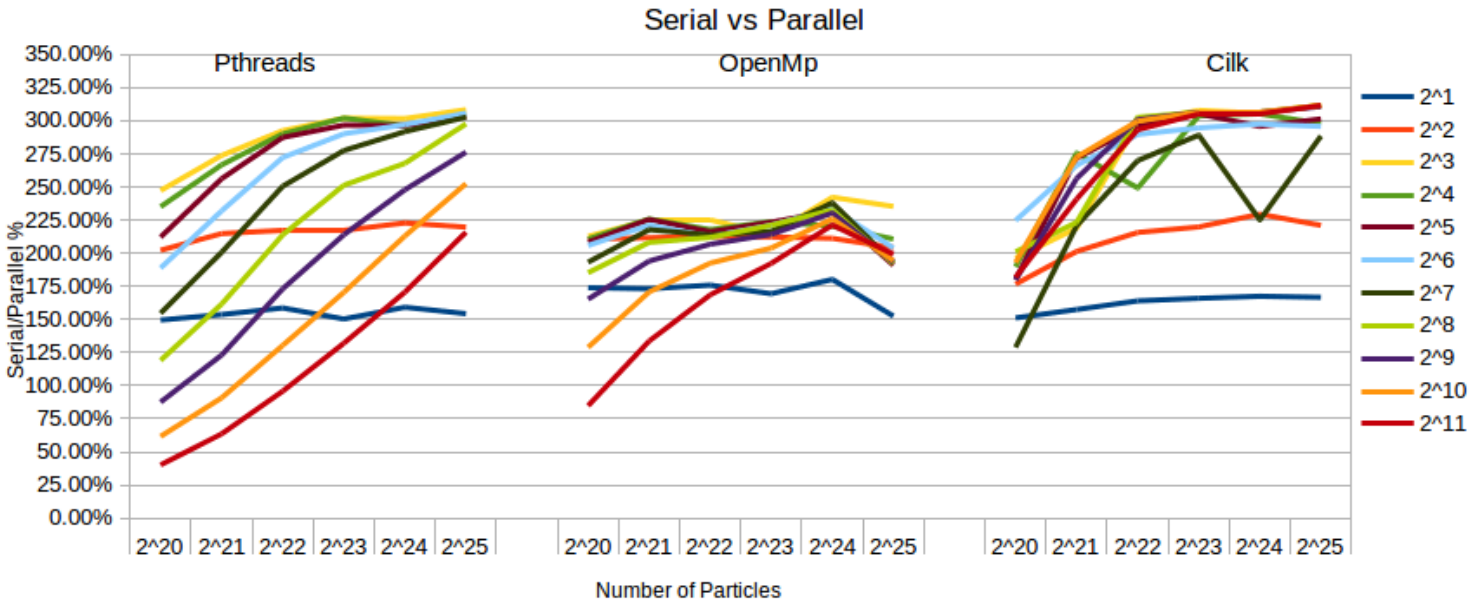


Στα τελευταία δύο διαγράμματα παρατηρούμε μια βελτίωση έως και 242% σε σχέση με το σειριακό πρόγραμμα. Στο διάγραμμα για την σφαίρα οι επιδώσεις είναι πεσμένες λόγο αυξημένης δραστηριότητας στον server.

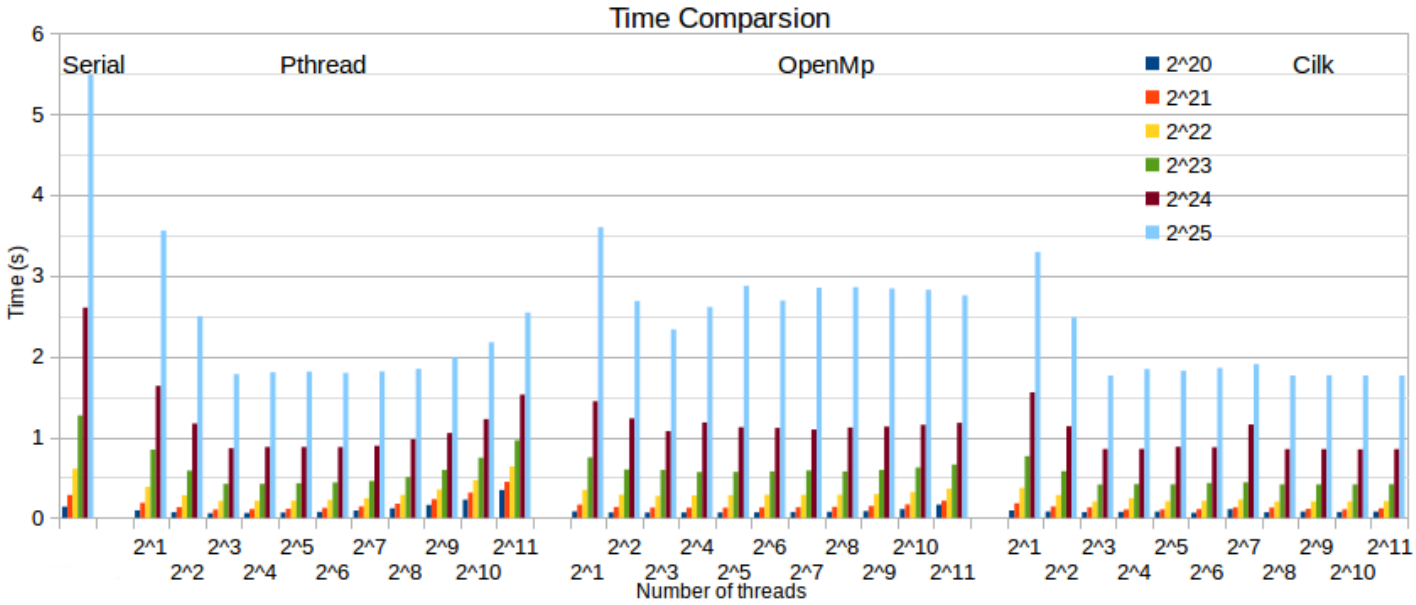
Παρατηρούμε επίσης μία μειωμένη απόδοση της radix_sort σε σχέση με τις προηγούμενες υλοποιήσεις. Ίσως αυτό οφείλεται σε λιγότερο καλή διαχείριση recursive διαδικασιών απο το OpenMp.



Σύγκριση Αποτελεσμάτων

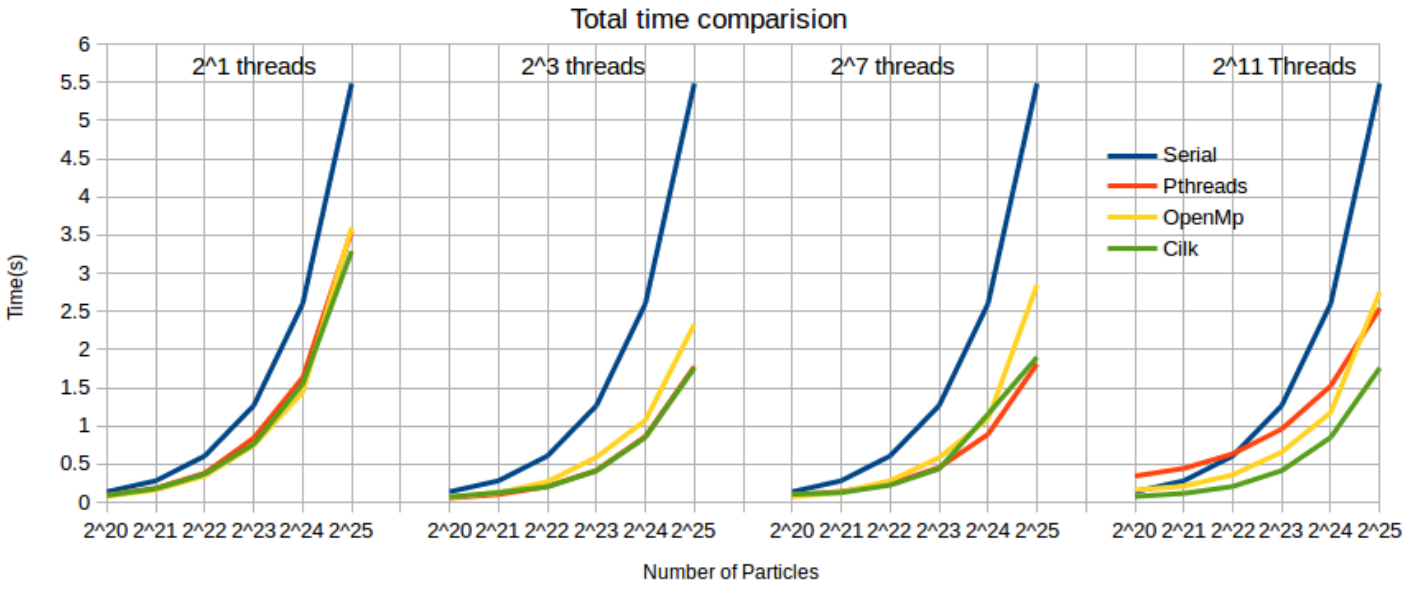


Δόθηκε μάχη για την πρώτη θέση στο ποσοστό βελτίωσης του συνολικού χρόνου σε σχέση με το σειριακό πρόγραμμα. Τελικά η cilk προηγείται στα αποτελέσματα μας με μέγιστο ποσοστό 311% ακολουθούν τα Pthreads με ποσοστό 308,8% ενώ στην τελευταία θέση με επίσης πολύ καλή βελτίωση η υλοποίηση με OpenMp και βέλτιστο ποσοστό 242%. Φυσικά αυτά τα αποτελέσματα τονίζεται ότι είναι ενδεικτικά καθώς επηρεάζονται σημαντικά από το εκάστοτε μηχάνημα και τις διεργασίες που τρέχουν σε αυτό την δεδομένη χρονική στιγμή.



Η υλοποίηση της cilk έχει για **2^25 σωματίδια** τον ελάχιστο χρόνο εκτέλεσης 1.760412 seconds ενώ ελάχιστος για τα Pthreads 1.778957 και Openmp 2.331383 Προφανώς οι χρόνοι αυτοί είναι για 2^3 νήματα, δηλαδή την βέλτιστη επιλογή για έναν 8πύρηνο υπολογιστή όπως ο server diades.

Για τα ίδια σωματίδια ο χειρότερος χρόνος εκτέλεσης για την cilk είναι 3.2901 , για Pthreads 3.554765 ενώ για OpenMp 3.597078 την ώρα που ο σειριακός κώδικας εκτελείται σε 5.486026 seconds!



Εδώ βλέπουμε την συμπεριφορά των διάφορων υλοποιήσεων για {2^1,2^3,2^7,2^11} νήματα. Παρατηρούμε ότι μόνο για 2^11 νήματα και [2^20,2^21] σωματίδια οι υλοποιήσεις των Pthreads και OpenMp παρουσιάζουν χειρότερους χρόνους απο τον σειριακό.

Ενδεικτικά αναφέρω πως στο δικό μου μηχάνημα τα αποτελέσματα που παίρνω είναι καλύτερα για Pthreads, ακολουθεί η υλοποίηση με OpenMp με μικρή διαφορά και στο τέλος βρίσκεται η Cilk. Προφανώς τα αποτελέσματα που πήρα απο το server diades διαφέρουν λόγο των υπόλοιπον διεργασιών που εκτελούσαν οι άλλοι χρήστες εκείνο χρονικό διάστημα.

Τελικό συμπεράσμα είναι ότι η παραλληλοποίηση ενός προγράμματος προσφέρει καλύτερες επιδόσεις, αρκεί βέβαια να μην γίνεται κατάχρηση των νημάτων αλλά να επιλέγονται με γνώμονα τις δυνατότητες του συστήματος και επί των πλείστων θεμιτή είναι η παραλληλοποίηση για ιδιαίτερα μεγάλα προβλήματα, καθώς τότε υπερτερεί σημαντικά ο παράλληλος προγραμματισμός!