Παράλληλα και Διανεμημμένα

Παράλληλη υλοποίηση της Διτονικής ταξινόμησης

Πέτρος Μητσέας 7925 - 7ο εξάμηνο, τομέας ηλεκτρονικής και υπολογιστών

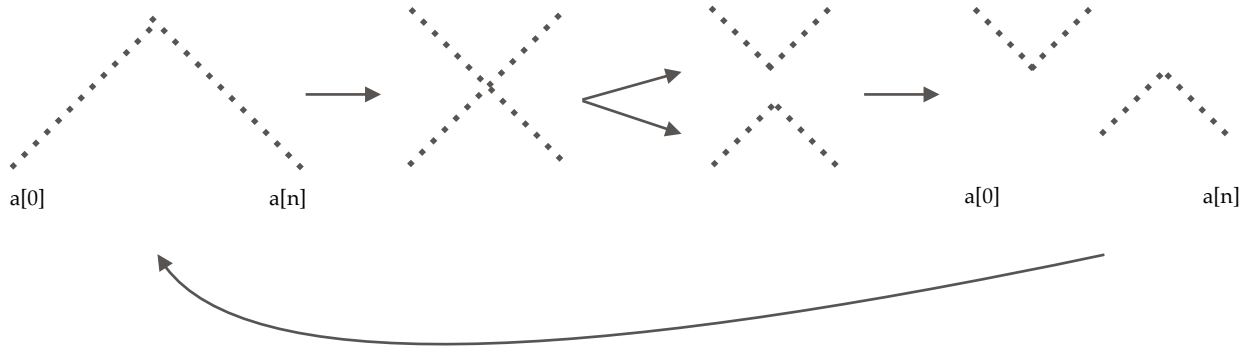
Εισαγωγη

Στην αναφορά αυτή περιγράφεται η λειτουργία του αλγόριθμου διτονικής ταξινόμησης καθώς και η παραλληλοποίηση του με Pthreads, OpenMP και Cilk. Τα αποτελέσματα που παρουσιάζονται αφορούν την υλοποίηση στο σύστημα “Διάδης”.

Περιγραφή του αλγορίθμου

Ο Bitonic βασίζεται σε δύο συναρτήσεις, την Sort και τη Merge. Γραφικά εξηγείται η λειτουργία τους παρακάτω:

Bitonic Merge



Η Merge τρέχει πάνω σε μια διτονική ακολουθία όπως φαίνεται στο σχήμα. Συγκρίνει το πρώτο στοιχείο της άυξουσας με το πρώτο της φθίνουσας, το δεύτερο με το δεύτερο και ούτω καθεξής. ‘Επειτα ανάλογα με το αποτέλεσμα της σύγκρισης ανταλλάσει τα στοιχεία. Το αποτέλεσμα είναι δύο ακολουθείες, οπου η μία είναι εξ ολοκλήρου μεγαλύτερη απο την άλλη. Οι δύο νέες ακολουθίες είναι επίσης διτονικές. Επομένως η διαδικασία αυτή μπορεί να εκτελεσθεί αναδρομικά, μέχρι ο αρχικός πίνακας να ταξινομηθεί πλήρως.

Bitonic Sort

Η συνάρτηση αυτή καλείται για την ταξινόμηση του πίνακα κατα την επιθυμητή φορά. Έστω οτι αυτή είναι αύξουσα. Η Sort κάνει χρήση της Merge. Επομένως η τελική ακολουθία πρέπει να έχει τη μορφή του προηγούμενου σχήματος (διτονική). Το πρόβλημα ανάγεται στη δημιουργία μιας αύξουσας ακολουθίας για το μισό του πίνακα, μιας φθίνουσας για το υπόλοιπο μισό και εφαρμογή της Merge. Η Sort λοιπόν μπορεί να κληθεί αναδρομικά, μέχρι οι υποακολουθίες που δημιουργούνται να έχουν απο ένα στοιχείο. Έπειτα καλείται η Merge "απο κάτω προς τα πάνω", συγχωνεύοντας τις διτονικές υποακολουθίες.

Παραλληλισμός

Απο τα παραπάνω γίνεται φανερό οτι, αφού σε κάθε αναδρομή η Sort καλεί τον εαυτό της σε καθένα απο τα δύο μισά του πίνακα, οι εργασίες που εκτελούνται σε κάθε μισό μπορούν να γίνουν παράλληλα με τη χρήση νημάτων (threads). Η bitonicMerge μπορεί επίσης να εκτελεστεί ταυτόχρονα, σε καθεμιά απο τις διτονικές που δημιουργεί. Η παραλληλοποίηση εξηγείται αναλυτικότερα παρακάτω.

Ανάλυση του αλγορίθμου

Σε αυτή την παράγραφο επιχειρείται μια επεξήγηση της bitonic, με τμηματική παράθεση του κώδικα που την υλοποιεί (σε C). Παρουσιάζονται μόνο τα κομμάτια του κώδικα εκείνα, που σχετίζονται με τον αλγόριθμο. Αναφορές σε αρχικοποιήσεις κλπ παραλείπονται.

Pthreads (χωρίς qsort)

```
int main(int argc, char **argv) {

        . . .
    if (N<16384){
        MAX_THREAD_A=1;
    }
    else{
        MAX_THREAD_A=TH;
    }
    MAX_THREAD_B=MAX_THREAD_A;
    a = (int *) malloc(N * sizeof(int));

    init();
    gettimeofday (&startwtime, NULL);
    sort();
    gettimeofday (&endwtime, NULL);

         . . .

    test();
    printf("Recursive wall clock time = %f\n", seq_time);
  pthread_exit(NULL);
}
```

Η εκτέλεση του προγράμματος γίνεται απο το τερματικό με την εντολή

./bitonic arg1 agr2

οπου 2^arg1 το μέγεθος του πίνακα και 2^arg2 = MAX_THREAD_A. Η main ελέγχει αρχικά αν η εντολή εκτέλεσης του προγράμματος έγινε σωστά.

Για μικρό μέγεθος πίνακα, στην πράξη παρατηρείται οτι ο σειριακός αλγόριθμος εκτελείται πιο γρήγορα απο τον παράλληλο. Επομένως το πρόγραμμα ελέγχει αν N<2^14 και σε αυτή την περίπτωση εκτελεί τον αλγόριθμο σειριακά. Έπειτα αρχικοποιείται ο πίνακας a και λαμβάνει τυχαίες τιμές με κλήση της init().

Καλείται η sort() που περιέχει τη ρουτίνα ταξινόμησης, εκτελείται έλεγχος των αποτελεσμάτων απο την test() και τυπώνεται στην κονσόλα το αποτέλασμα του ελέγχου και ο χρόνος εκτέλεσης. Η test(), ελέγχει αν κάθε στοιχείο του ταξινομημένου πίνακα είναι μεγαλύτερο ή ίσο (για αύξουσα φορά) απο το προηγούμενο του. Αν εντοπίσει έστω και ένα λάθος, τυπώνει “TEST Failed”.

```
void sort() {
    if (MAX_THREAD_A==1) recBitonicSortfunc(0, N, ASCENDING);
    else {
        pthread_t thread;
        struct Array newArray;
        struct Array *pointer;
        pointer=&newArray;
        pointer->lo=0;
        pointer->cnt=N;
        pointer->dir=ASCENDING;
        number_1++;
        pthread_create(&thread, NULL, recBitonicSort, (void *)pointer);
        pthread_join(thread, NULL);
        number_1--;
    }
}
```

Εδώ, αν έχει επιλεχθεί σειριακή εκτέλεση, η sort καλεί την σειριακή recBitonicSortfunc. Αλλιώς δημιουργεί ένα νέο thread το οποίο εκτελεί τη συνάρτηση recBitonicSort.

Ο number_1 δείχνει πόσα thread -με συνάρτηση την recBitonic sort- τρέχουν. Γι αυτό μετα τη δημιουργία του νήματος αυξάνεται κατα ένα και αφου το νήμα τελειώσει την εργασία μειώνεται. Οι μετρητές είναι χρήσιμοι για να μην υπερβεί το πρόγραμμα τα μέγιστα threads που επιθυμούμε να φτιαξει.

```
void *recBitonicSort(void *args){
    struct Array *array;
    array=(struct Array *)args;
    int cnt1=array->cnt;
    int lo1=array->lo;
    int dir1=array->dir;
    if (cnt1>1) {
        if (number_1<MAX_THREAD_A)
        {
            number_1=number_1+2;

                    . . .

         for (int t=0; t<2; t++){
                pthread_create(&threads[t], NULL, recBitonicSort, (void *)pointer[t]);
            }
            for(int t=0; t<2; t++) {
                pthread_join(threads[t],NULL);
            }
            number_1=number_1-2;
            number_2++;
            pthread_create(&thread, NULL, bitonicMerge1, (void *)array);
            pthread_join(thread, NULL);
            number_2--;
        }
        else recBitonicSortfunc(lo1, cnt1, dir1);
    }
    pthread_exit(NULL);
}
```

Ο κορμός της συνάρτησης ταξινόμησης θα μπορούσαμε να πούμε οτι είναι η void* recBitonicSort.

Αν το cnt<1 δηλαδή ο πίνακας στον οποίον επιδρά έχει 1 στοιχείο, η συνάρτηση τερματίζει (base case της αναδρομής). Αλλιώς, συνεχίζει με την επόμενη if.

Στην περίπτωση οπού δεν έχει συμπληρωθεί ο αριθμός των επιθυμητών threads, η συνάρτηση χωρίζει τον πίνακα στη μέση και δημιουργεί απο 1 thread για κάθε μισό. Τα νέα thread τρέχουν την ίδια συνάρτηση αλλα για διαφορετικό κομμάτι του πίνακα.

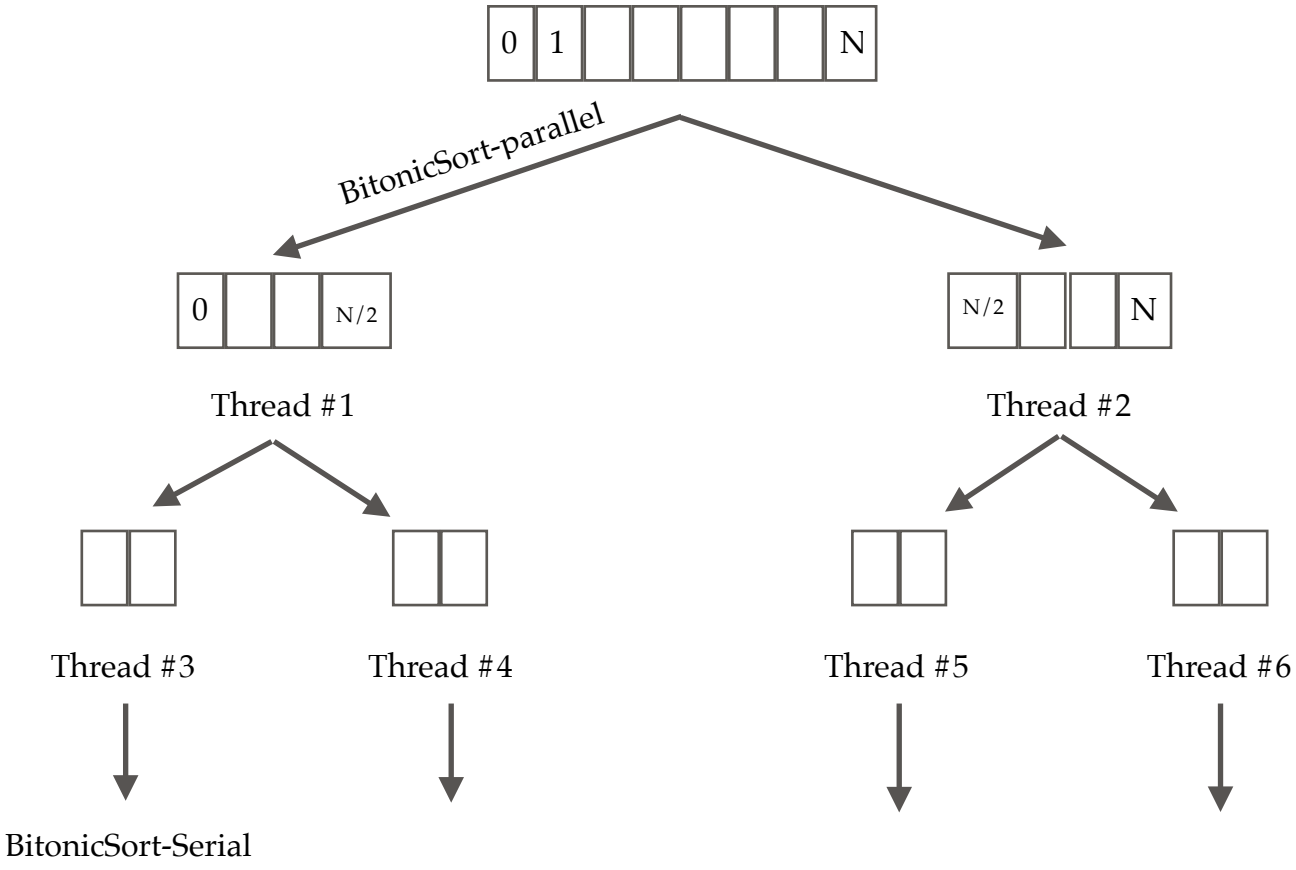
Αν ξεπεραστεί ο αριθμός των επιθυμητών threads, απο εκεί και πέρα το πρόγραμμα εκτελείται σειριακά.

Έπειτα το πρόγραμμα περιμένει να ολοκληρωθούν οι εργασίες απο τα επιμέρους νηματα, τα κάνει join και τρέχει την bitonicMerge.

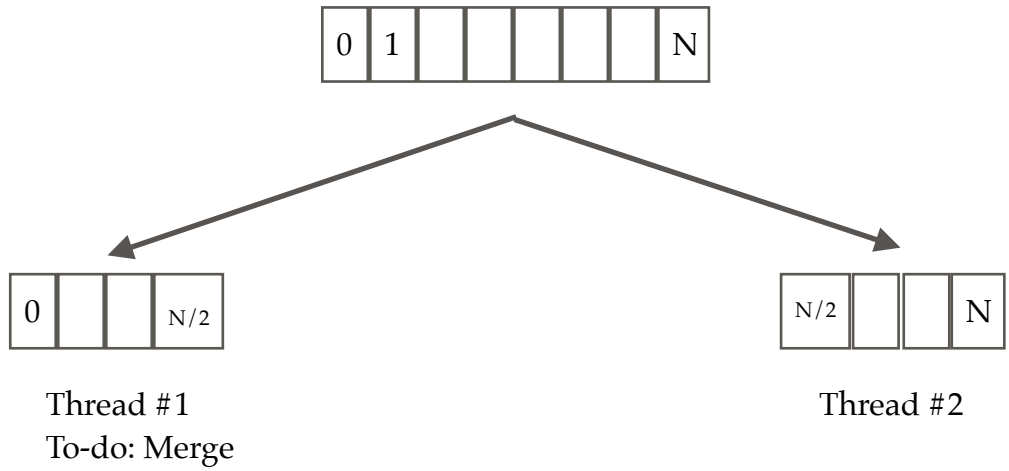
Ο δεύτερος παραλληλισμός θα γίνει στην bitonicMerge. Τα σημαντικά σημεία του κώδικα της void* bitonicMerge εξηγούνται παρακάτω.

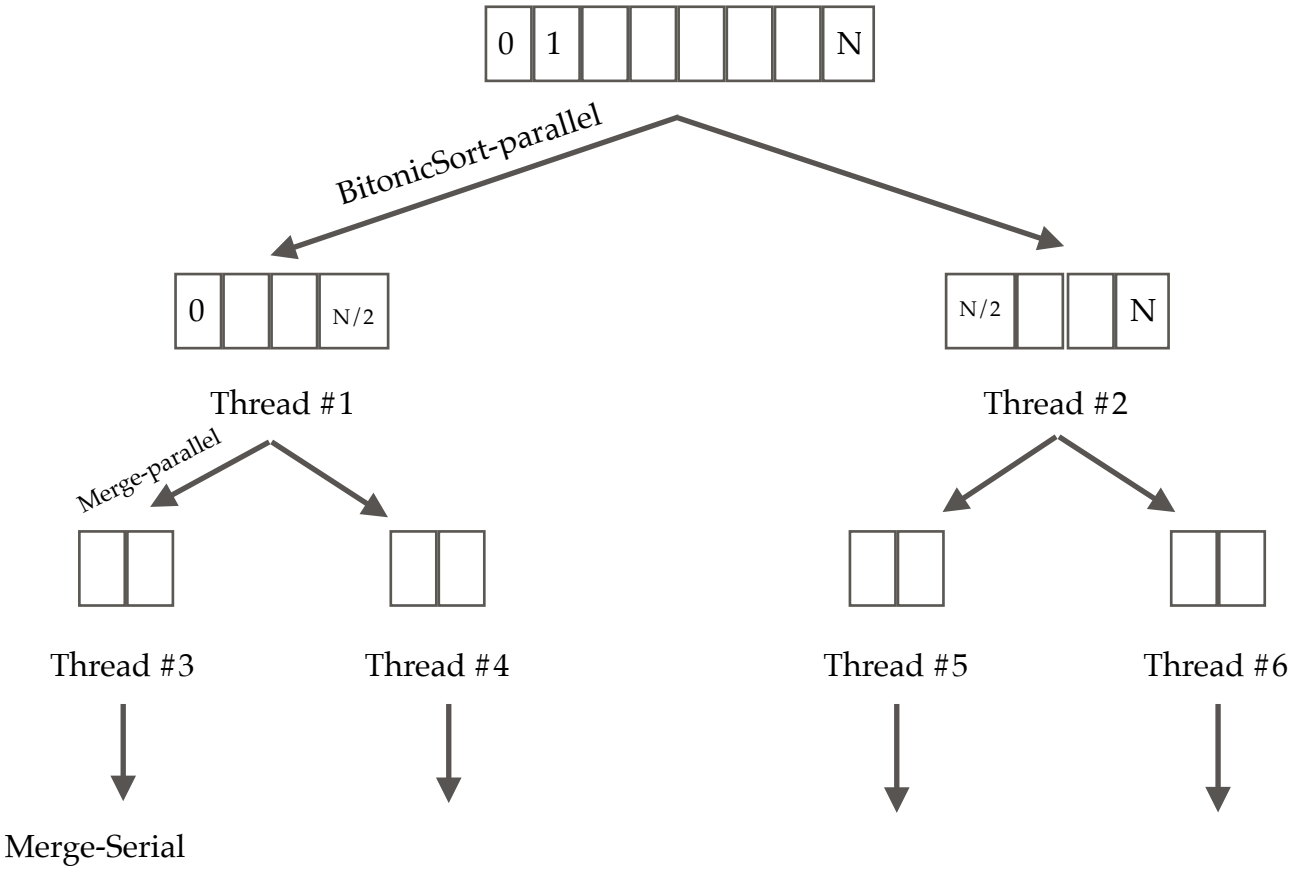
```
if (cnt1>1){
        if (number_2<(MAX_THREAD_B-number_1)){
            number_2=number_2+2;

                    . . .

            for(int t=0; t<2; t++) {
                pthread_join(threads[t], &status);
            }
            number_2=number_2-2;

        }
        else {
            for (i=lo1; i<lo1+k; i++) compare(i, i+k, dir1);
            bitonicMerge(lo1, k, dir1);
            bitonicMerge(lo1+k, k, dir1);
        }

    }
```

Η bitonicMerge δημιουργεί αναδρομικά όσα threads επιτρέπονται, λαμβάνοντας υπ όψιν αυτά που ήδη έχουν δημιουργηθεί απο την Sort. (Χρησιμοποιώντας ενα δεύτερο μετρητή). Ειδάλλως τρέχει σειριακά. Να σημειωθεί οτι στη ρίζα του δέντρου, οπου θα έχουν γίνει join τα νήματα της Sort, δεν θα έχουμε idle threads, καθώς τα νήματα που ελευθερώθηκαν θα εκτελέσουν δουλεία της Merge. Αυτό θα φανεί καλύτερα γραφικά. Έστω οτι τα μέγιστα threads είναι 6.



Μόλις δημιουργηθούν 6 threads, το καθένα θα εκτελέσει τον υπόλοιπο κώδικα Sort και Merge σειριακά. Έστω οτι το κατώτερο layer έχει ολοκληρωθεί. Τα threads 3 4 και 5 6 γίνονται join και η δουλειά που έχει μείνει για τα νήματα 1 και 2 είναι να καλέσουν τη Merge. Τώρα όμως τα ελεύθερα νήματα είναι 6-2=4, τα οποία θα διατεθούν για την bitonicMerge κάθε νήματος.





OpenMP και Cilk

Ο ίδιος αλγόριθμος δοκιμάστηκε με τα API OpenMP και Cilk. Τα API αυτά είναι υψηλότερου επιπέδου σε σχέση με την βιβλιοθήκη pthread.h.

Για το OpenMP χρημοποιήθηκαν οι εντολές #pragma omp parallel, sections, master, barrier και lock. Η κύρια διαφορά με τον προηγούμενο κώδικα είναι το κλείδωμα των μετρητών. Κατί τέτοιο αποτρέπει την ταυτόχρονη εγγραφή των μεταβλητών αυτών απο τα threads, και ότι αυτό συνεπάγεται. Ωστόσο τίθεται το θέμα speed vs accuracy καθώς το κλείδωμα και ξεκλείδωμα των μετρητών, επιδρά αρνητικά στην ταχύτητα εκτέλεσης. Απο την άλλη, αν δεν κλειδωθούν, κάποιες φορές θα παράγονται λάθος αποτελέσματα στην μέτρηση των ενεργών νημάτων, με αποτέλεσμα να δημιουργούνται μερικά παραπάνω ή λιγότερα νήματα. Παρόλα αυτά μέσα απο δοκιμές φαίνεται οτι στη συγκεκριμένη περίπτωση οι επιπτώσεις είναι πολύ μικρές, ενω η ταξινόμηση αυτή καθ' αυτή δεν επηρεάζεται. Οπότε είναι προτιμότερο το πρόγραμμα να τρέχει γρηγορότερα. Γι αυτό στα pthreads δεν χρησιμοποιήθηκε mutex_lock.

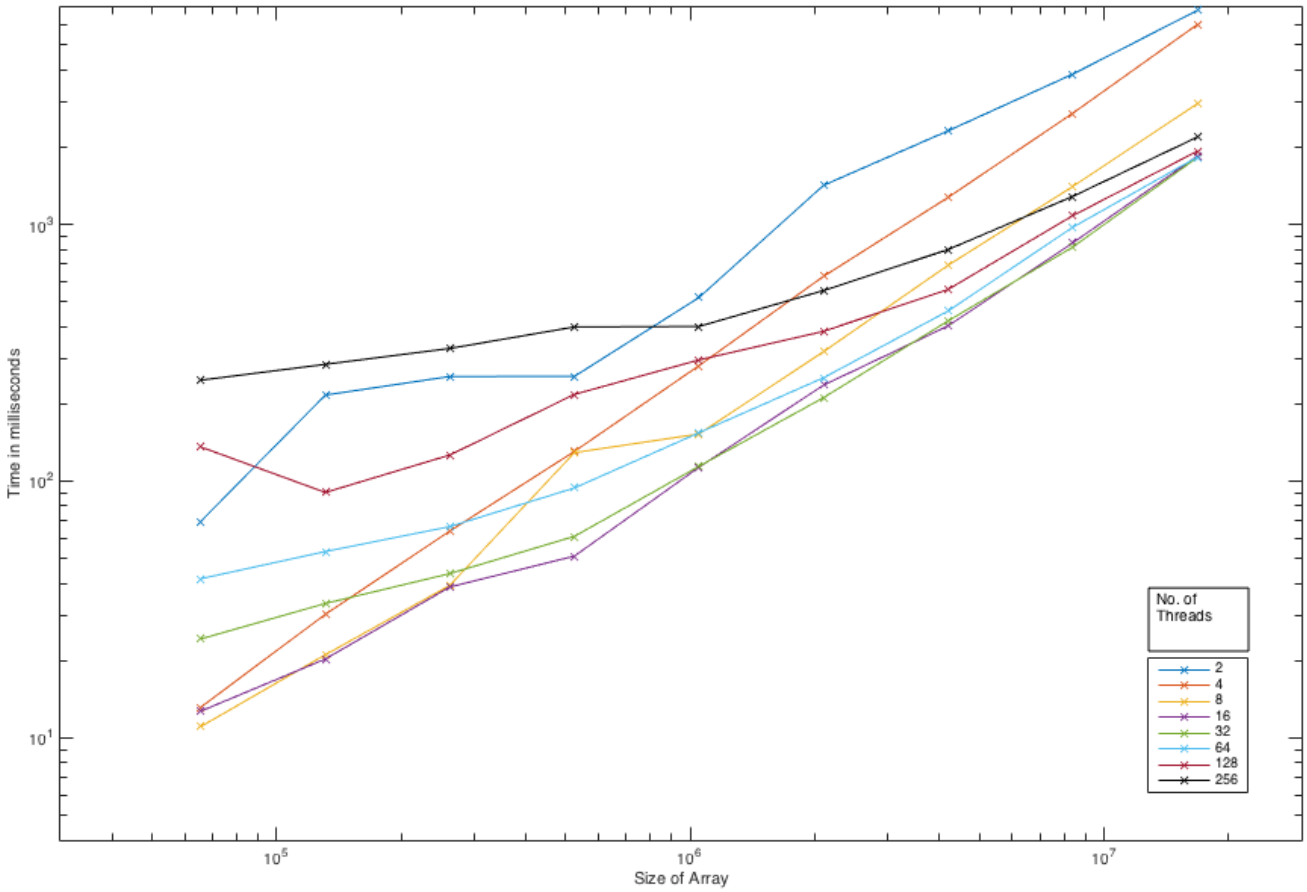
Ακόμη, ένα πρόβλημα που εμφανίστηκε στο OpenMP είναι οτι η εκτέλεση κρεμάει, όταν επιχειρήσουμε να δημιουργήσουμε πάνω απο 2^6 νήματα και ο πίνακας έχει πάνω απο 2^{23} στοιχεία

Για το Cilk, χρησιμοποιήθηκαν οι εντολές cilk_spawn και clik_sync. Σε γενικές γραμμές, η υλοποίηση με pthread παρουσίασε τον καλύτερο χρόνο έναντι των cilk και openMP

Δοκιμές

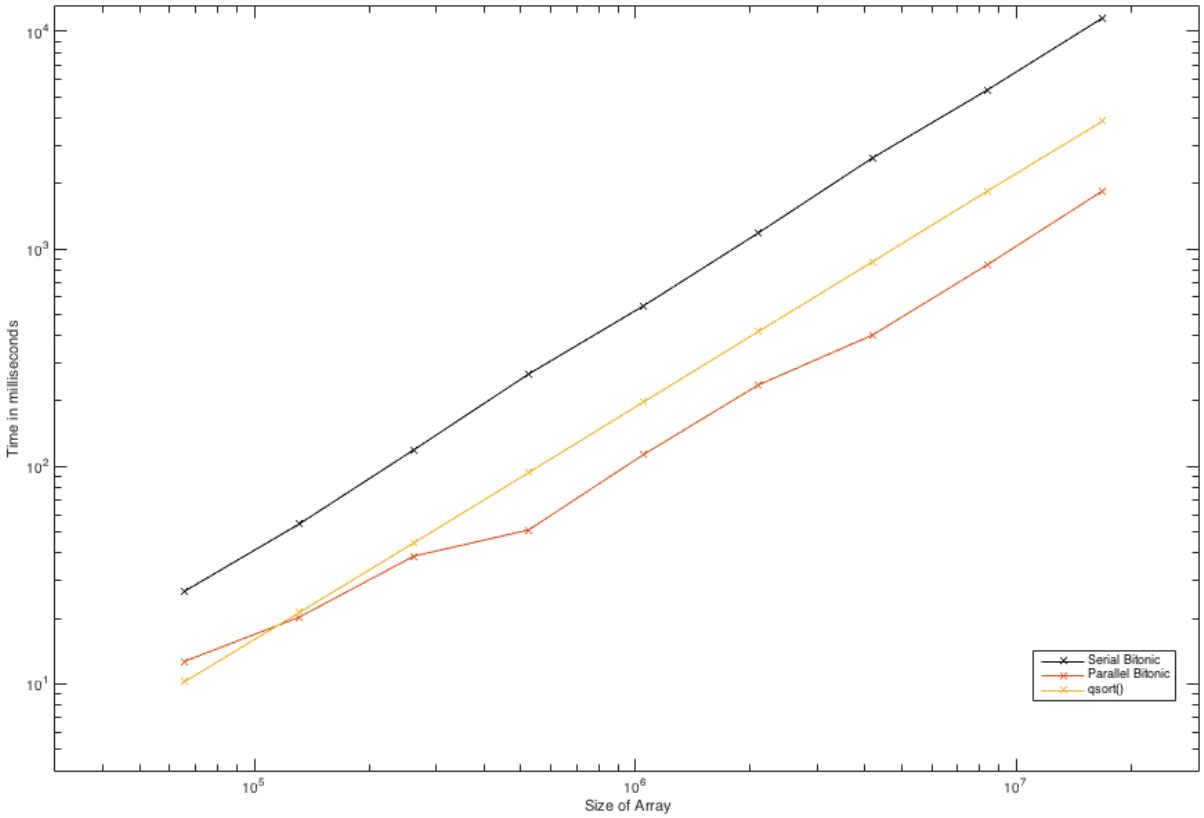
Στο σύστημα "Διάδης" δοκιμάστηκαν:
- Ο σειριακός Bitonic (bitonic_serial.c)
- Ο παράλληλος Bitonic, όπως εξηγήθηκε παραπάνω (pthread, openMP και Cilk)
- Η qsort (qsort.c)
- Υβριδικός παράλληλος αλγόριθμος bitonic-qsort (pthread)

Η δοκιμή περιλαμβάνει μέγεθος πίνακα απο 2^{16} έως 2^{24} στοιχεία και μέγεθος thread απο 2 εως 2^8.



Η απόδοση του παράλληλου bitonic με pthreads, για διάφορα μεγέθη πίνακα

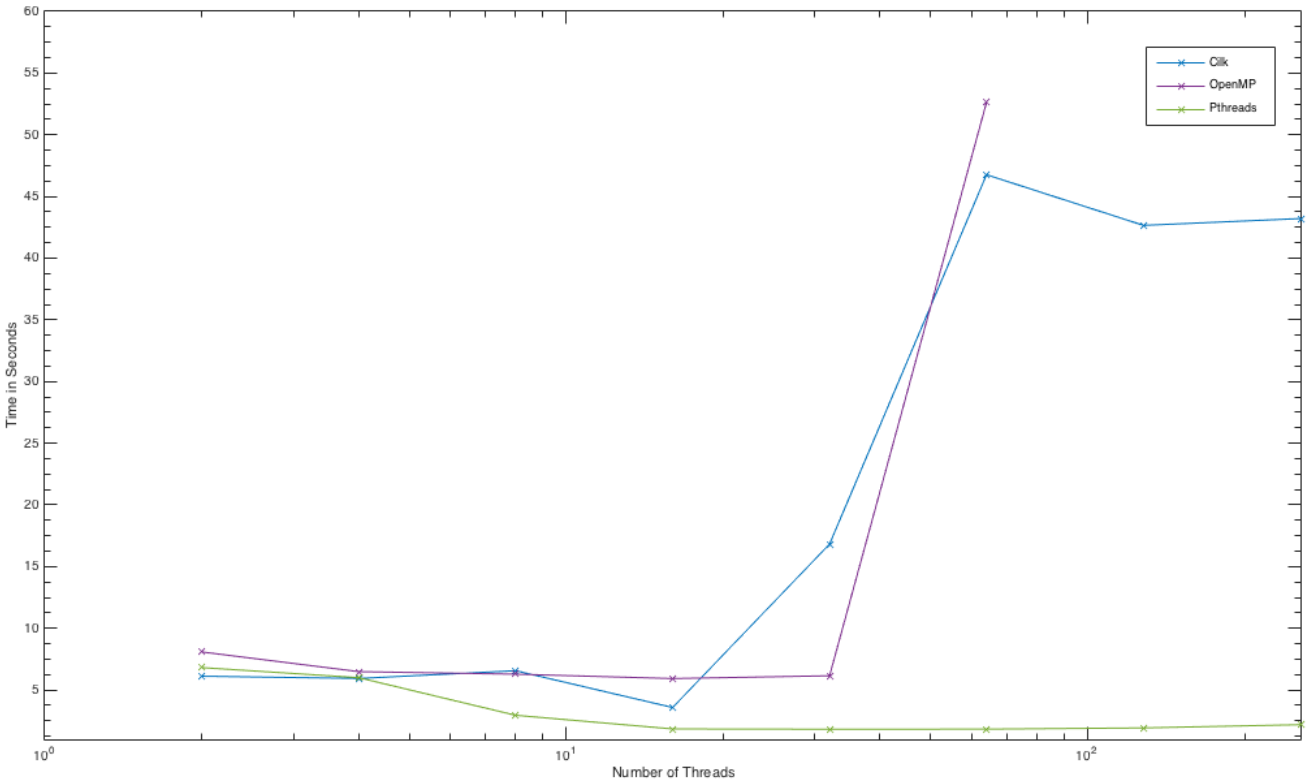
Παρατηρούμε οτι στο συγκεκριμένο hardware, πετυχαίνουμε ταχύτερη υλοποίηση με 16 threads. Για μικρό μέγεθος πίνακα δεν συμφέρει να δημιουργηθούν πολλά νήματα. Το αντίθετο συμβαίνει όταν ο πίνακας είναι μεγάλος. Ακολουθεί ένα συγκριτικό διάγραμμα μεταξύ της σειριακής υλοποίησης, της παράλληλης με 16 threads και της ταξινόμησης με τη χρήση της qsort().



Time of Serial Bitonic	Time of qsort()	Time of Parallel Bitonic (pthread)	No. of Elements	**Parallel to Serial ratio (%)**	**Parallel to qsort ratio (%)**
0.026687	0.010236	0.012710	65536	**47%**	**124%**
0.054393	0.021309	0.020288	131072	**37%**	**94%**
0.119294	0.044530	0.038713	262144	**32%**	**87%**
0.265901	0.093674	0.051052	524288	**19%**	**54%**
0.546797	0.197657	0.113283	1048576	**20%**	**57%**
1.181783	0.416074	0.236516	2097152	**20%**	**57%**
2.624918	0.874436	0.402333	4194304	**15%**	**46%**
5.382882	1.846747	0.846668	8388608	**15%**	**46%**
11.491346	3.854398	1.843228	16777216	**16%**	**47%**

Για μεγάλο αριθμό στοιχείων, η παράλληλη bitonic εκτελείται περίπου στο 1/6 του χρόνου της σειριακής και στο 1/2 του χρόνου της qsort().

Παρακάτω φαίνονται τα διαγράμματα του χρόνου για τις Cilk (cilk.c), OpenMP, (openmp.c) και pthreads (bitonic_parallel.c) συναρτήσει του αριθμού των threads για δεδομένο πίνακα 2^{24} στοιχείων



Καλύτερος χρόνος των Cilk, OpenMp απο την σειριακή διτονική (για λιγοτερα απο 32 νηματα). Χειρότερος χρόνος απο την qsort().

Τέλος είναι εφικτή η δημιουγία ενός υβριδικού αλγόριθμου bitonic-qsort, προκειμένου να εκμεταλευτούμε την ταχύτητα της qsort() για μικρό μήκος πίνακα. Ο αλγόριθμος, χωρίζει τον πίνακα σε υποακολουθίες, μέχρι να εξαντληθεί ο αριθμός των threads που επιθυμούμε να διαθέσουμε. Έπειτα, η δουλειά που έχει μείνει σε κάθε νήμα δεν εκτελείται απο την σειριακή bitonic, αλλά απο την qsort, ώστε να φτιαχτεί η διτονική, και να κληθεί μετά η bitonicMerge(). Η υλοποίηση έγινε με pthreads, και βρίσκεται στο αρχείο bitonic_qsort.c

Στο διάγραμμα παρακάτω φαίνεται ο λόγος bitonic-qsort / bitonic-parallel (16 threads) ως προς το μέγεθος του πίνακα. Φαίνεται ο αριθμός των νημάτων εκείνος, που έδωσε καλύτερα αποτελέσματα (ratio < 1).