ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ ΑΠΘ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ

Παράλληλα και Διανεμημένα Συστήματα

Ιωαννης Παναγιώτης Μπουντουρίδης

AEM 8872

October 31, 2018

Εργασία 1

1 Εισαγωγή

Πρωτού αρχίσουμε την υλοποίηση της qsort_seq στην cilk, openmp και pthreads θα ήταν σημαντικό να κρίνουμε κατα πόσο αξίζει να ταξινομούμε παράλληλα εναν πίνακα. Στο φακελο pthread_VS_serial υπάρχουν δύο c προγράμματα. Αυτά ταξινομούν παράλληλα και σε σειρά αντίστοιχα πίνακες μικρού μήκους. Εκτελέστηκαν πάνω απο 100 χιλιάδες πειραματικές δοκιμές και προέκυψε οτι ένας πίνακας εως και 4096 στοιχείων ταξινομεί τα στοιχεία πιο γρήγορα οταν εκτελείται σε σειρά παρά οταν εκτελείται παράλληλα. Παρολο που τα παραπάνω πειράματα αποτελούν πειράματα τύχης, η στατιστική είναι ενα σπουδαίο εργαλείο για να βγάζεις βάσιμα συμπεράσματα. Γι αυτό τον λόγο το προγραμμα pthreads που θα υλοποιηθεί στην συνέχεια θα ταξινομεί τους πίνακες μικρού μήκους σειριακά. Ας ξεκινήσουμε.

2 Ανάλυση κώδικα (Cilk)

```
void qsort_seq(int *v, int n) {
  if (n > 1) {
    int p = partition(v, n);
    cilk_spawn qsort_seq(v,p);
    qsort_seq(&v[p+1],n-p-1);
    cilk_sync;
  }
}
```

Για την υλοποίηση του αλγόριθμου qsort_seq μεσω της cilk απαιτείται να γίνει spawn ενα κομμάτι qsort_seq(v,p) ώστε να εκτελείται παράλληλα. Το δεύτερο κομμάτι qsort_seq(&v[p+1],n-p-1) εφόσον η συνάρτηση qsort_seq καλείται απο την main() το αναλαμβάνει αυτή. Η cilk_sync περιμένει τα δύο κομμάτια αυτά να τελειώσουν.Μέσω της cilk_rts_set_param() ρυθμίζουμε τα threads που θα εκτελεστούν. Σε περίπτωση που εκανα cilk_spawn και το δεύτερο κομμάτι του πίνακα δηλαδή cilk_spawn qsort_seq(&v[p+1],n-p-1) τοτε οι χρόνοι ταξινόμησης ήταν αρκετά πιο αργοί σε σχέση με τον κώδικα που εκανε ένα spawn. Βλέπουμε πως το overhead πολλών thread μειώνει σημαντικά την ταχύτητα του προγραμματος μας.

3 Ανάλυση κώδικα (Openmp)

Αρχείο: qsort-main.c

```
gettimeofday(...);
#pragma omp parallel
{
#pragma omp single
{
  qsort_seq(a, n);
}
}
gettimeofday(...);
```

Αρχείο: qsort-sequential.c

```
void qsort_seq(int *v, int n) {
if (n > 1) {
int p = partition(v, n);
#pragma omp task
{
qsort_seq(v,p);
}
qsort_seq(&v[p+1],n-p-1);
}
}
```

Για την υλοποίηση της qsort_seq με openmp όμοια με την cilk, η #pragma omp task καθε φορά καλεί ένα thread να εκτελέσει το κομμάτι της ταξινόμησης qsort_seq(v,p) παράλληλα. Η #pragma omp parallel υπάρχει ώστε ο κώδικας που βρίσκεται εσωτερικά της να εκτελεστεί παράλληλα. Χωρίς την βοήθεια της #pragma omg single η εντολή qsort_seq() θα εκτελούνταν μια φορά απο κάθε thread του υπολογιστή μια φορά. Με την υπαρξη της αναλαμβάνει ένα thread να την εκτελέσει μια φορά. Μέσω της omp_set_num_threads() ρυθμίζουμε τα threads που θα εκτελέσουν την qsort_seq

4 Ανάλυση κώδικα (pthreads)

```
pthread_mutex_t mt;
int MAXTHREADS =8;
int threadsEnded=0;
typedef struct {
  int *v;
  int n;
} params;
void qsort_parallel(int *v, int n,int pthreads) {
  //init mutex
  pthread_mutex_init(&mt, NULL);
  //create params to pass
  params item;
  params *p =&item;
  item.v = v;
  item.n = n;
  //change MAXTHREADS
  MAXTHREADS = pthreads;//default is 8
  //call parallel void*
  parallelthread(p);}
```

Η συνάρτηση qsort_parallel() έχει ώς όρισματα τον πίνακα προς ταξινόμηση *v, τον αριθμό των pthreads που μας έδωσε ο χρήστης και το μέγεθος του πίνακα n. Αρχικά αποθηκεύω τον πίνακα και το μέγεθος του στην struct params και καλώ την parallelthread()

```
void *parallelthread(void* arg){
  params *pa = (params *) arg;
  int *v = pa->v;
  int n = pa->n;
  if(n>4096){
  pthread_mutex_lock(&mt);
  --MAXTHREADS;
  int remaining_threads = MAXTHREADS;
  int threads_respawn = threadsEnded;
  if(threadsEnded>=1){
    --threadsEnded;
  }
  pthread_mutex_unlock(&mt);
  if(remaining_threads>=0||threads_respawn>0){
  int p = partition(v,n);
  params *pa1 = malloc(sizeof(params *));
  pa1->v =v;
  pa1->n = p;
  params *pa2 = malloc(sizeof(params *));
  pa2->v = &v[p+1];
  pa2->n = n-p-1;
  pthread_t thread;
  pthread_create(&thread,NULL,parallelthread,pa1);
  parallelthread(pa2);
  pthread_join(thread,NULL);
  free(pa1);
  free(pa2);
  }
  else{
  qsort_seq(v,n);
  pthread_mutex_lock(&mt);
  ++threadsEnded;
  pthread_mutex_unlock(&mt);
  }
  }else{
  qsort_seq(v,n);
  pthread_mutex_lock(&mt);
  ++threadsEnded;
  pthread_mutex_unlock(&mt);
  }
}
```

Η parallelthread() παίρνει τα ορίσματα *v και n μεσω της struct και ελέγχει αν το μέγεθος του πίνακα είναι μεγαλύτερο του 4096. Ειδάλως τον ταξινομεί σειριακά. Σε περίπτωση που το μεγεθος του πίνακα ξεπερνά τα 4096 στοιχεία καλεί την pthread_mutex_lock() και μειώνει την MAXTHREADS κατά 1. Το threads_respawn ειναι μια ακέραια τιμή που δείχνει πόσα threads τελείωσαν την δουλειά τους. Αυτή η μεταβλητή μας βοηθά να αξιοποιήσουμε threads που δε χρησιμοποιούνται. Αν λοιπόν υπάρχουν remaining_threads ή threads_respawn κανω partition() τον πίνακα δημιουργώ δύο νέες struct params που τις εκχωρώ το πρώτο και το δεύτερο κομμάτι αντίστοιχα.Καλώ ενα thread για το δεύτερο κομμάτι του partition() ενω για το πρώτο το εκτελώ αναδρομικά. Τέλος στις περιπτώσεις που τα threads εκτελούν σειριακά την ταξινόμηση αφου τελειώσουν αυξάνουν κατά 1 την μεταβλητή threadsEnded η οποία ενημερώνει τα επόμενα threads οτι είναι διαθέσιμα να ξανά δουλέψουν. Για ευνόητους λόγους πριν αυξήσω κατά ενα την μεταβλητή threadsEnded καλώ την pthread_mutex_lock() ώστε να μην υπάρξει ταυτόχρονη πρόσβαση της μεταβλητής από άλλα threads.

5 Διαγράμματα

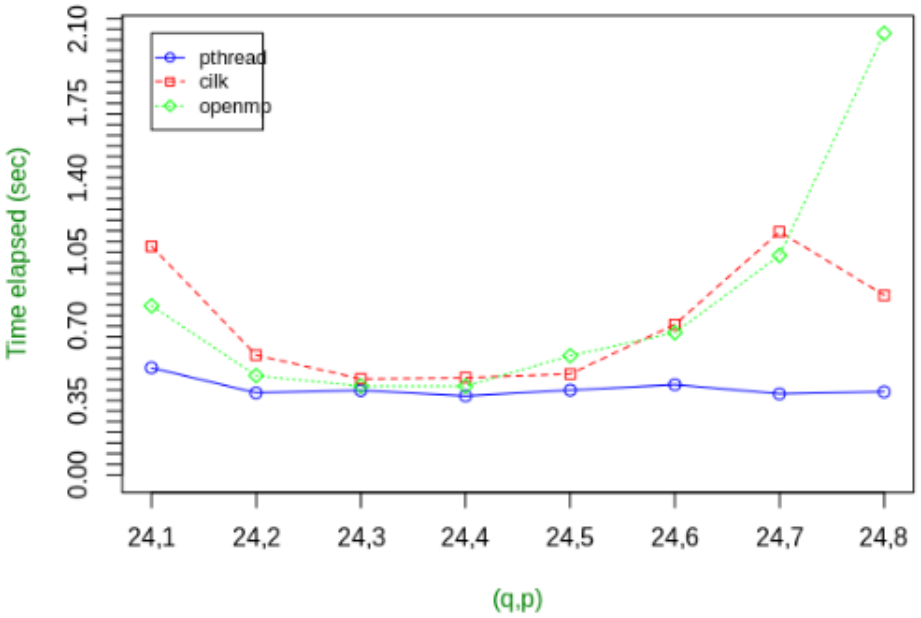

Figure 1: Εκτελέσεις με q = 24 για p = 1,2,3...,8

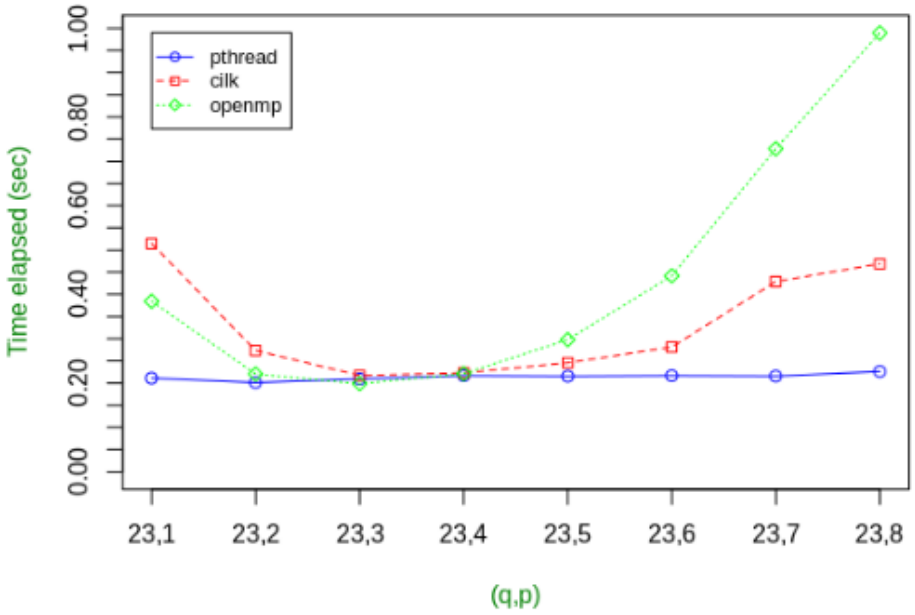

Figure 2: Εκτελέσεις με q = 23 για p = 1,2,3...,8

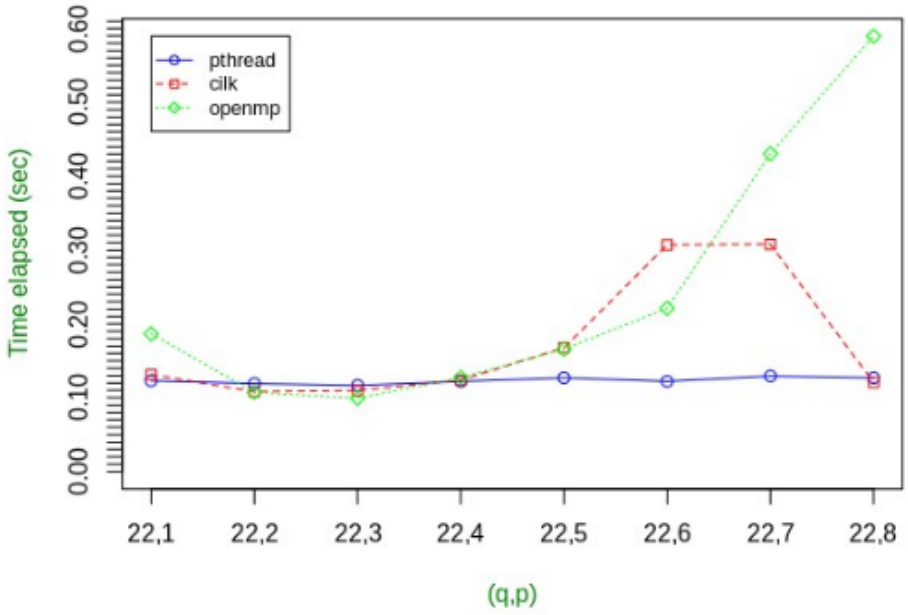

Figure 3: Εκτελέσεις με q = 22 για p = 1,2,3...,8

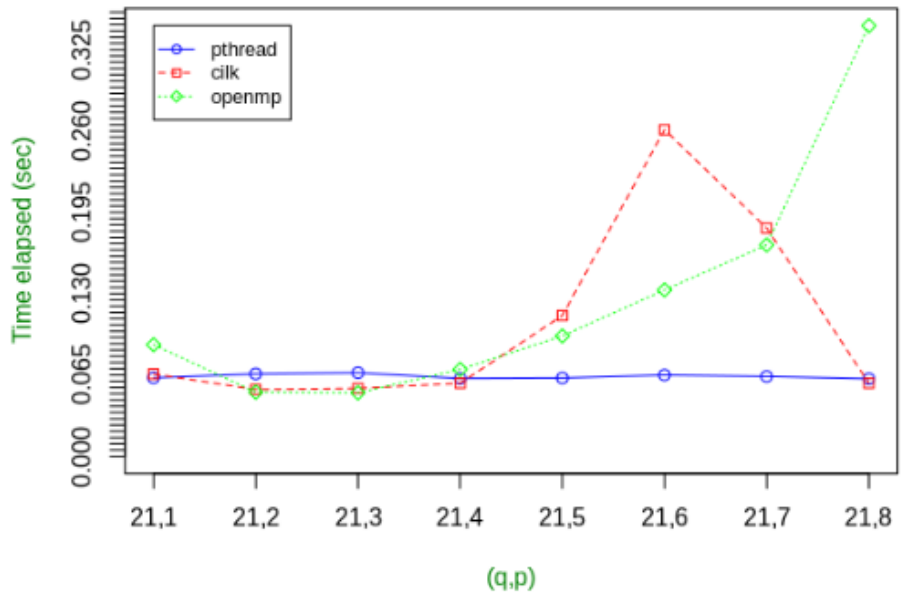

Figure 4: Εκτελέσεις με q = 21 για p = 1,2,3...,8

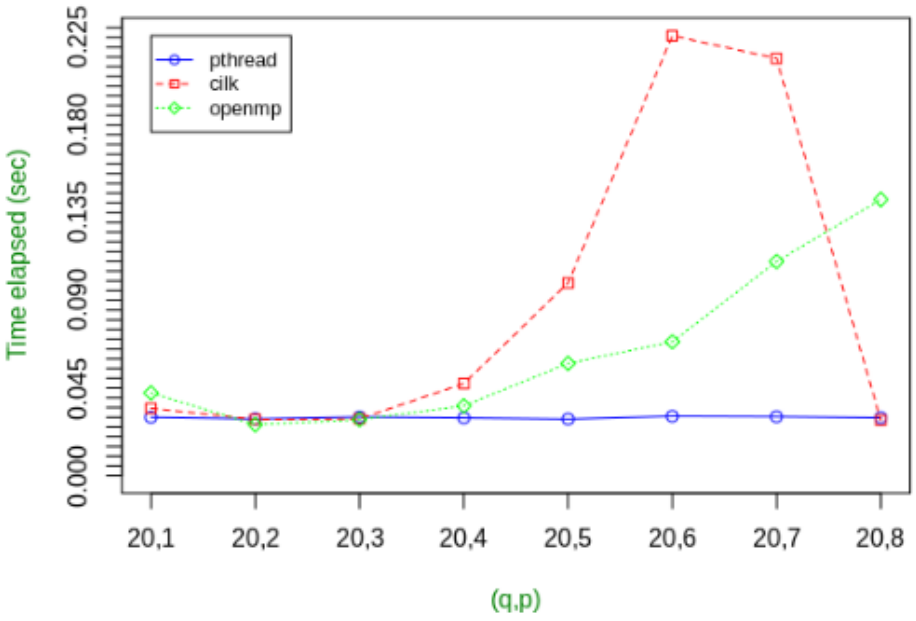

Figure 5: Εκτελέσεις με q = 20 για p = 1,2,3...,8

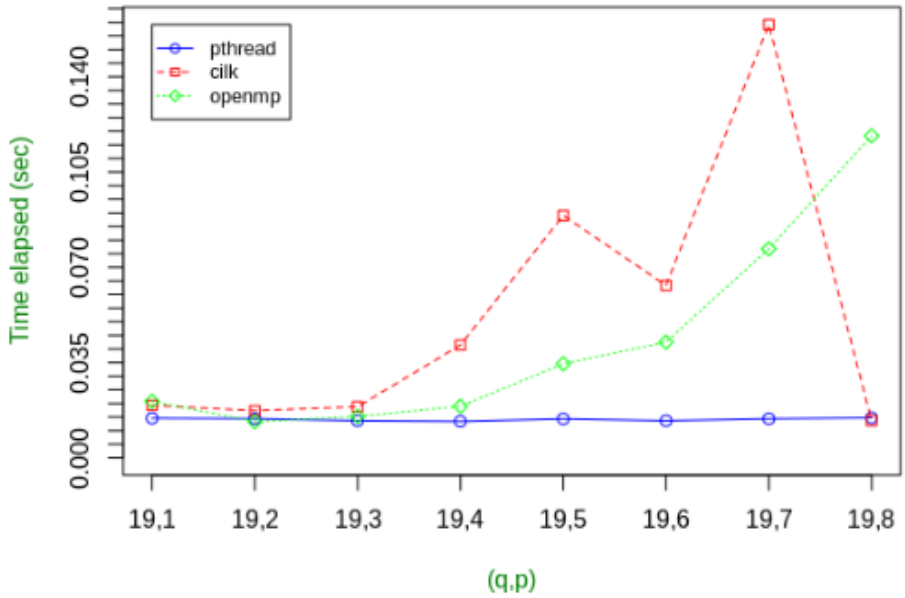

Figure 6: Εκτελέσεις με q = 19 για p = 1,2,3...,8

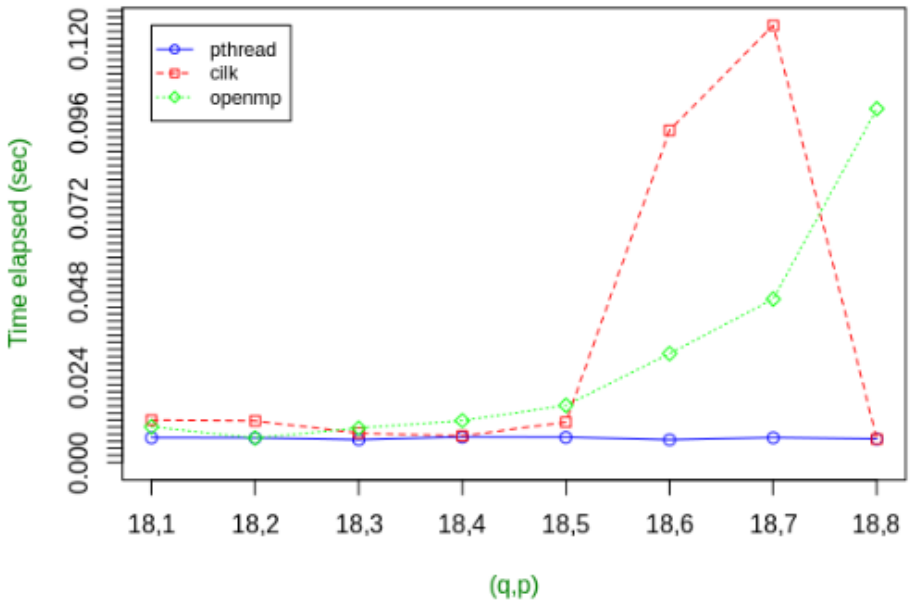

Figure 7: Εκτελέσεις με q = 18 για p = 1,2,3...,8

6 Συμπεράσματα

Είδαμε πως στις περισσότερες περιπτώσεις η pthread εκτελείται πιο γρήγορα έναντι της cilk και openmp. Ειδικά στην περίπτωση p>4 τα threads αυξάνουν αρκετά, φαίνεται πως η cilk και η openmp αργούν σημαντικά. Το overhead των threads αποτελεί ενα σημαντικό παράγοντα.Άλλωστε αποδείξαμε πως απο ένα σημείο και πέρα η παραλληλη ταξινόμηση ενος αρκετά μικρού πίνακα γίνεται πιο αργά απ οτι σειριακά. Όσον αφορά την pthread υλοποίηση σε σχέση με την συνάρτηση qsort_seq που μας δίνεται, οι χρόνοι εκτέλεσης είναι απο 3 εως 4 φορες πιο γρήγοροι.

7 Κώδικας

https://drive.google.com/file/d/11W3X0pLptTxEK0B0t7Ly6sDkseVfvRlp/view?usp=sharing