Άσκηση 1

Παράλληλα και Διανεμημένα Συστήματα Υπολογιστών

Νοέμβριος 2017

του φοιτητή

Παπαδόπουλου Κωνσταντίνου
ΑΕΜ 8677

Ζητούμενα:

Μετατροπή της σειριακής υλοποίησης του προγράμματος για τον αλγόριθμο Bitonic Sort που δίνεται, σε πέντε διαφορετικές εκδόσεις χρησιμοποιώντας:

1. CilkPlus - α) αναδρομική β) επαναληπτική έκδοση

2. OpenMP - α) αναδρομική β) επαναληπτική έκδοση

3. Pthreads - αναδρομική έκδοση

Θα περιγραφούν η διαδικασία παραλληλισμού και οι έλεγχοι ορθότητας που χρησιμοποιήθηκαν. Θα σχολιαστεί επίσης η ταχύτητα των υπολογισμών, ειδικά σε σύγκριση με τη συνάρτηση *stdlib qsort()* και έχοντας κάνει δοκιμές στο σύστημα diades για p = [1:8] και q = [16:24], όπου 2^p είναι ο μέγιστος αριθμός των νημάτων που δημιουργούνται και 2^q είναι το μήκος του πίνακα των τυχαίων αριθμών που κατασκευάζουμε.

Extra Credit:

Έχει γίνει επίσης συνδυασμένη χρήση της συνάρτησης *stdlib qsort(),* η οποία θα αναλυθεί παρακάτω, στην υλοποίηση των εκδόσεων OpenMP και Pthreads για την κατάταξη μικρού μεγέθους υποπινάκων, έτσι ώστε να πετύχουμε καλύτερη απόδοση στο χρόνο εκτέλεσης.

CilkPlus

α) Αναδρομική Έκδοση

Η λέξη-κλειδί *cilk_spawn* καθορίζει ότι η λειτουργία της συνάρτησης “παιδί” μπορεί να εκτελείται παράλληλα με τον καλούντα. Είναι σημαντικό να σημειωθεί ότι το *cilk_spawn* επιτρέπει τον παραλληλισμό, δεν τον επιβάλλει . Επιτρέπεται, με αυτόν τον τρόπο, στο χρόνο εκτέλεσης να κλέψει τον κώδικα που ακολουθεί το *cilk_spawn* για να εκτελεστεί σε άλλο νήμα εργασίας.

Το *cilk_sync* καθορίζει ότι όλες οι συναρτήσεις “παιδιά” που δημιουργούνται από αυτή τη λειτουργία πρέπει να ολοκληρωθούν πριν η εκτέλεση του προγράμματος περάσει αυτή τη δήλωση.

Η τοποθέτηση του *cilk_spawn* στη συνάρτηση *recBitonicSort()* έγινε με βάση τη λογική, αλλά και με trial and error δοκιμές, καθώς η τοποθέτηση *cilk_spawn* στη συνάρτηση *bitonicMerge()* χειροτέρευε το χρόνο εκτέλεσης.

Συγκεκριμένα, οι *recBitonicSortCilk(lo, k, ASCENDING)* και *recBitonicSortCilk(lo+k, k, DESCENDING)* μπορούν να εκτελεσθούν παράλληλα με σκοπό να αυξηθεί συνολικά η ταχύτητα ταξινόμησης του πίνακα.

Πριν την κλήση της *bitonicMergeCilk(lo, cnt, dir)* πρέπει να έχουν επιστρέψει τα αποτελέσματά τους οι προηγούμενες συναρτήσεις και για αυτό τοποθετούμε το *cilk_sync* στη συγκεκριμένη θέση.

Κώδικας

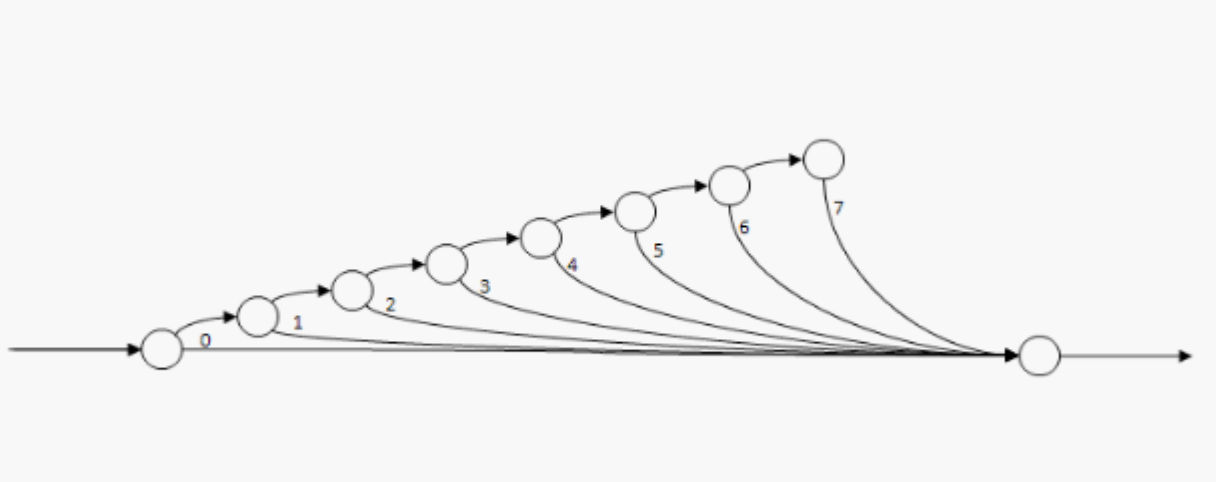
```
/** function recBitonicSortCilk()
   Uses Cilk to parallelize the process
```

```
**/
void recBitonicSortCilk(int lo, int cnt, int dir) {
  if (cnt>1) {
    int k=cnt/2;
    cilk_spawn recBitonicSortCilk(lo, k, ASCENDING);         //spawn
    cilk_spawn recBitonicSortCilk(lo+k, k, DESCENDING);
    cilk_sync;                                               //synchronizes
    bitonicMergeCilk(lo, cnt, dir);
  }
}
```

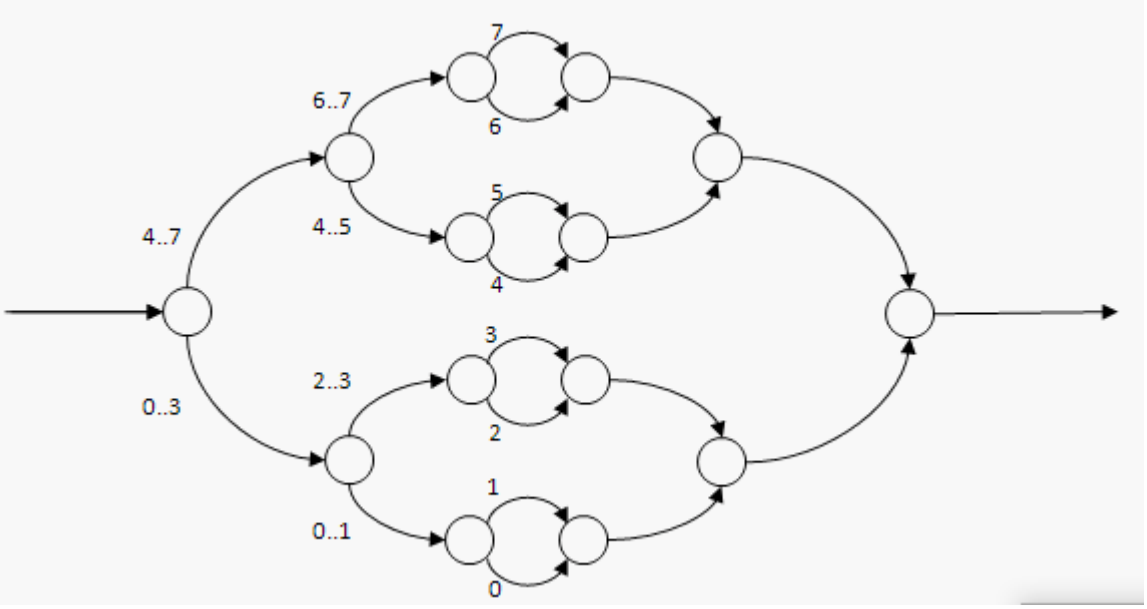
β) Επαναληπτική Έκδοση

Η υλοποίηση της επαναληπτικής έκδοσης είναι ιδιαίτερα απλή. Με τη χρήση του *cilk_for* πραγματοποιείται η εντολή για παραλληλοποίηση του συγκεκριμένου επαναληπτικού βρόχου for.

Η χρήση του *cilk_spawn* και *cilk_sync*, όπως έγινε πριν δεν ενδείκνυται για τη συγκεκριμένη περίπτωση. Παρουσιάζεται πολύ παραπάνω κόστος (overhead), διότι η δουλειά που γίνεται spawn είναι λίγη και ό,τι απομένει πρέπει να “κλαπεί” για κάθε επανάληψη, κάτι το οποίο κοστίζει αρκετά. Επίσης, έχει μικρό βαθμό παραλληλοποίησης μιας και υπάρχει μόνο ένας δημιουργός παράλληλης εργασίας τη φορά.



Αντιθέτως το *cilk_for* διαιρεί το έργο του βρόχου στο μισό και στη συνέχεια το διαιρεί στο μισό και πάλι, μέχρι να υπάρχουν αρκετά κομμάτια για να κρατήσει τους πυρήνες απασχολημένους, αλλά ταυτόχρονα και να ελαχιστοποιήσει την επιβάρυνση που επιβάλλεται από το *cilk_spawn*.



(https://www.cilkplus.org/tutorial-cilk-plus-keywords)

Κώδικας

```
/*
  imperative Cilk version of bitonic sort
*/
void impBitonicSortCilk() {

  int i,j,k;
```

```
  for (k=2; k<=N; k=2*k) {
    for (j=k>>1; j>0; j=j>>1) {
      cilk_for (i=0; i<N; i++) {
        int ij=i^j;
        if ((ij)>i) {
          if ((i&k)==0 && a[i] > a[ij])
             exchange(i,ij);
          if ((i&k)!=0 && a[i] < a[ij])
             exchange(i,ij);
        }
      }
    }
  }
}
```

**Γενικά, θα μπορούσε να γίνει παραλληλοποίηση και των συγκρίσεων (compare()), αν και δε θα προβούμε σε μια τέτοια υλοποίηση.*

OpenMP

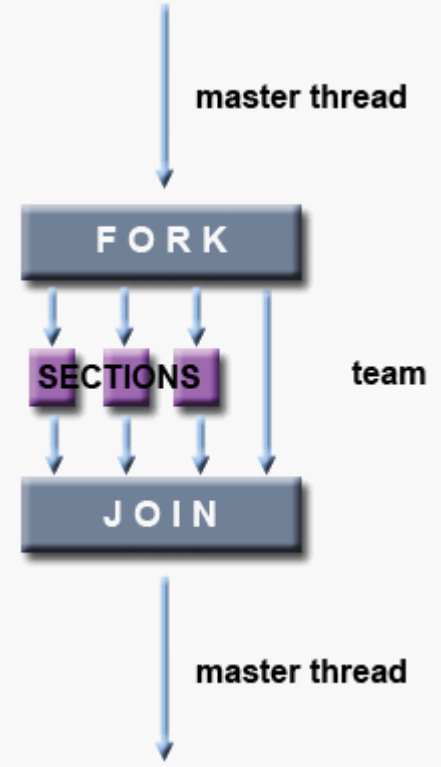
Οδηγούμενοι από την παραλληλοποίηση του κώδικα με τη CilkPlus μπορούμε τώρα με την ίδια λογική να παραλληλοποιήσουμε τα αντίστοιχα τμήματα κώδικα και με το API του OpenMP.

Και στις δύο εκδόσεις χρησιμοποιείται η οδηγία parallel του OpenMP. Αυτή η οδηγία ορίζει την περιοχή του κώδικα που επιθυμούμε να εκτελεστεί παράλληλα.

(https://computing.llnl.gov/tutorials/openMP/)

α) Αναδρομική Έκδοση

Στην αναδρομική έκδοση γίνεται χρήση της οδηγίας διαμοιρασμού εργασίας SECTIONS. Κάθε τμήμα εκτελείται μία φορά από ένα νήμα της ομάδας, ενώ διαφορετικά τμήματα εκτελούνται από διαφορετικά νήματα.



Για να μη δημιουργείται σε κάθε αναδρομική κλήση μη ελεγχόμενος αριθμός από νήματα, τοποθετούμε και το όρισμα threads στη συνάρτηση ελέγχοντας έτσι τον αριθμό των νημάτων - *recBitonicSortOpenMP(lo, k, ASCENDING, threads/2), recBitonicSortOpenMP(lo+k, k, DESCENDING, threads – threads/2).*

EXTRA CREDIT: Όταν ο αριθμός των threads είναι ίσος με 1, κάνουμε χρήση της συνάρτησης qsort(), ώστε να πετύχουμε καλύτερη απόδοση στο χρόνο εκτέλεσης.

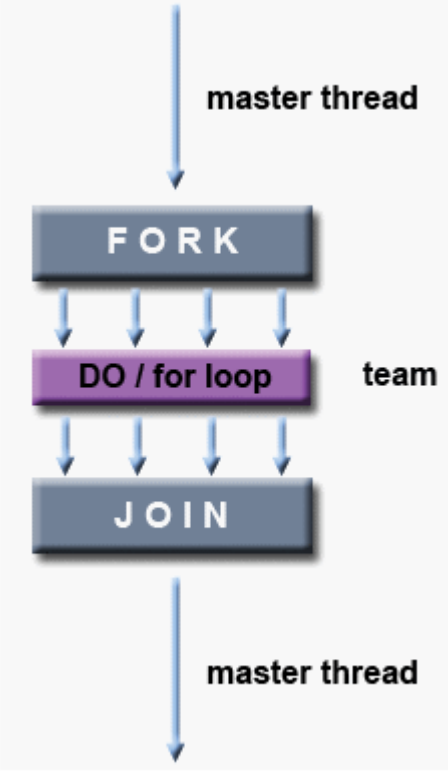
Κώδικας

```
/** function recBitonicSortOpenMP()
   Uses OpenMP to parallelize the process
**/
void recBitonicSortOpenMP(int lo, int cnt, int dir, int threads) {
  if (cnt>1) {
    int k=cnt/2;

 omp_set_num_threads(threads);

  if(threads == 1){
        qsort( a + lo, k, sizeof( int ), asc );             // --EXTRA CREDIT--
        qsort( a + ( lo + k ) , k, sizeof( int ), desc );

  }
  else{

    #pragma omp parallel sections
    {

    #pragma omp section
    recBitonicSortOpenMP(lo, k, ASCENDING, threads/2);

    #pragma omp section
```

```
    recBitonicSortOpenMP(lo+k, k, DESCENDING, threads - threads/2);

        }

  }

    bitonicMergeOpenMP(lo, cnt, dir);
  }
}
```

β) Επαναληπτική Έκδοση

Σε αυτήν την έκδοση γίνεται χρήση της οδηγίας διαμοιρασμού εργασίας FOR. Με αυτόν τον τρόπο πετυχαίνουμε παραλληλισμό δεδομένων, δηλαδή όλα τα νήματα θα εκτελέσουν το ίδιο τμήμα κώδικα, αλλά σε διαφορετικά δεδομένα. Ορίζουμε επίσης, με το *schedule(static,chunk)*, οι επαναλήψεις του βρόχου να χωρίζονται σε μέρη μεγέθους chunk και να μοιράζονται στατικά στα νήματα, καθώς αυτό παρατηρήθηκε ότι επιταχύνει ακόμη περισσότερο την εκτέλεση.



Κώδικας

```
/*
  imperative OpenMP version of bitonic sort
*/
void impBitonicSortOpenMP() {

    int i,j,k;

   for (k=2; k<=N; k=2*k) {
    for (j=k>>1; j>0; j=j>>1) {
      #pragma omp parallel for \
         shared(j,k) private(i) \
         schedule(static,chunk)
      for (i=0; i<N; i++) {
         int ij=i^j;
         if ((ij)>i) {
           if ((i&k)==0 && a[i] > a[ij])
              exchange(i,ij);
           if ((i&k)!=0 && a[i] < a[ij])
              exchange(i,ij);
         }

      }
    }
  }
}
```

Pthreads

Αναδρομική Έκδοση

Η υλοποίηση της αναδρομικής έκδοσης της Bitonic Sort με Pthreads ακολουθεί την ίδια λογική με τις προηγούμενες υλοποιήσεις.

Αρχικά ορίζουμε μία δομή (struct) διαμέσου της οποίας θα περνάμε τις παραμέτρους σε κάθε νήμα. Σε αυτές τις παραμέτρους περιλαμβάνονται οι *lo, cnt, dir,* καθώς και η *layer* που δηλώνει έμμεσα πόσα threads έχουμε φτιάξει.

EXTRA CREDIT: Όταν ο αριθμός των threads τείνει να γίνει μεγαλύτερος του μέγιστου αριθμού που έχουμε ορίσει να δημιουργούνται, κάνουμε χρήση της συνάρτησης qsort(), ώστε να πετύχουμε καλύτερη απόδοση στο χρόνο εκτέλεσης.

Κώδικας

```
//////////////////////////////////////////Pthreads implementation

typedef struct{
    int lo, cnt, dir, layer;
} sarg;

/** function PbitonicMerge()
    Using Pthreads.
 **/
void * PbitonicMerge( void * arg ){
```

```
        ....
        ....
        ....
        pthread_t thread1, thread2;
        arg1.lo = lo;
        arg1.cnt = k;
        arg1.dir = dir;
        arg1.layer = layer - 1;
        arg2.lo = lo + k;
        arg2.cnt = k;
        arg2.dir = dir;
        arg2.layer = layer - 1;
        pthread_create( &thread1, NULL, PbitonicMerge, &arg1 );
        pthread_create( &thread2, NULL, PbitonicMerge, &arg2 );

        pthread_join( thread1, NULL );
        pthread_join( thread2, NULL );
    }
    return 0;
}

/** function PrecBitonicSort()
    first produces a bitonic sequence by recursively sorting
    its two halves in opposite sorting orders, and then
    calls bitonicMerge to make them in the same order
    Using Pthreads.
 **/

void * PrecBitonicSort( void * arg ){
        ....
        ....
        ....
        int k = cnt / 2;
        if( layer >= threads ) {                    // --EXTRA CREDIT--
            qsort( a + lo, k, sizeof( int ), asc );
```

```
            qsort( a + ( lo + k ) , k, sizeof( int ), desc );
        }

        ....
        ....
        ....
    }
    return 0;
}
```

Έλεγχοι Ορθότητας

Ο έλεγχος της ορθότητας των αποτελεσμάτων γίνεται κυρίως με δύο τρόπους. Καταρχήν, η συνάρτηση *test()* ελέγχει, σκανάροντας όλο τον πίνακα, τη διαφορά των διαδοχικών στοιχείων του, ώστε να είμαστε σίγουροι ότι επιτυγχάνεται η αύξουσα ταξινόμησή του. Έπειτα, ένας άλλος, πιο εμπειρικός, τρόπος ελέγχου είναι η χρήση της *print()* μέσω της οποίας μπορούμε οπτικά να ελέγξουμε, μέσα από πολλές επαναλήψεις του προγράμματος, αν τα αποτελέσματά μας είναι τα επιθυμητά.

Δοκιμές στο Σύστημα Diades

Για την εξαγωγή μετρήσεων και τη μελέτη της ταχύτητας των υπολογισμών (για p = [1 : 8] και q = [16 : 24]) μέσω του Diades, χρησιμοποιήθηκε ένα bash script της μορφής:

```
#!/bin/bash

for q in {16..24}
do
printf "\n----------------------------------\n" >> prec-time-threads.txt
for p in {1..8}
do
./prec-time-threads $q $p >> prec-time-threads.txt
done
done
```

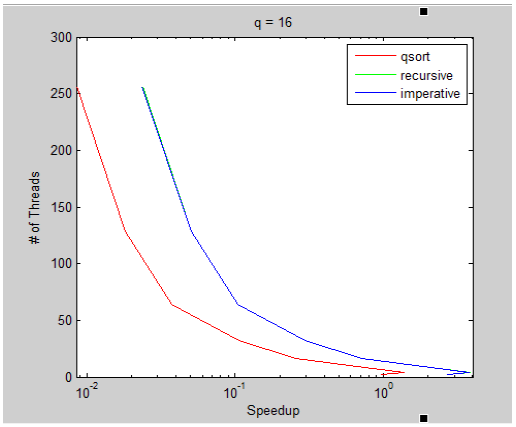
Στατιστικά Αποτελέσματα

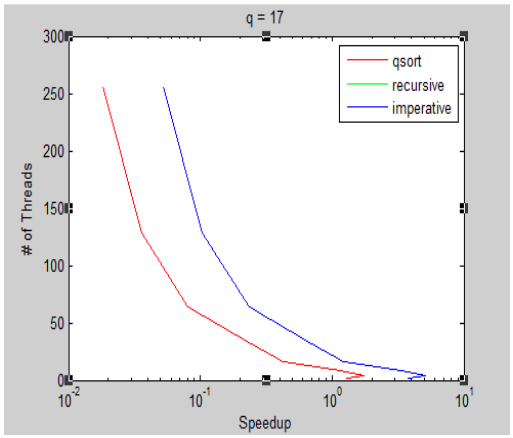
Στη συνέχεια παρουσιάζονται γραφήματα τα οποία σκοπεύουν να δώσουν μια καλύτερη εικόνα για την ταχύτητα των υπολογισμών ανάλογα με την υλοποίηση και την έκδοση που χρησιμοποιείται για την ταξινόμηση του πίνακα τυχαίων αριθμών. (βλ. Παράρτημα Α)

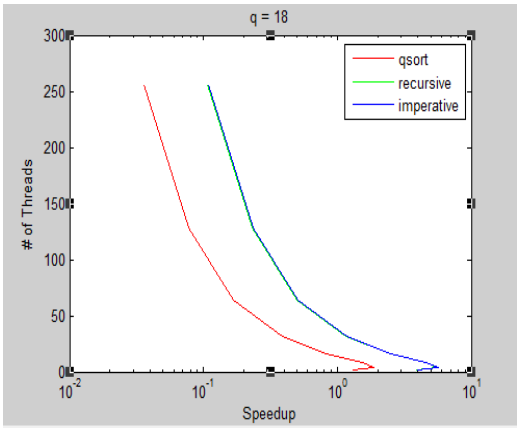
Οι τιμές έχουν προκύψει από μια κλασική στατιστική διαδικασία. Έχουμε αφαιρέσει τις ακραίες τιμές και μέσα από δέκα επαναλήψεις έχουμε υπολογίσει τη μέση τιμή του χρόνου εκτέλεσης για συγκεκριμένα μήκη πινάκων και συγκεκριμένο αριθμό νημάτων.

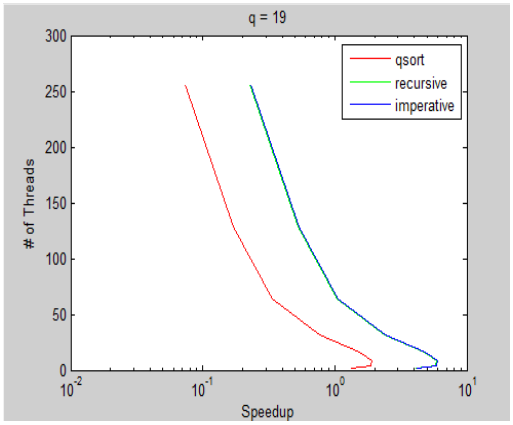
Γραφήματα

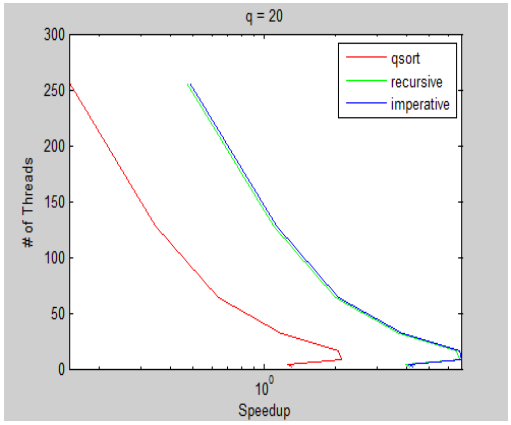
Pthreads

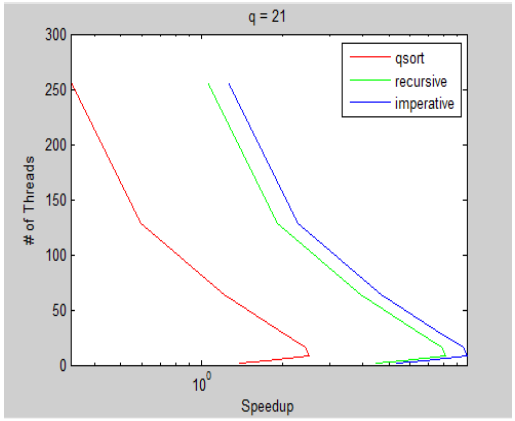


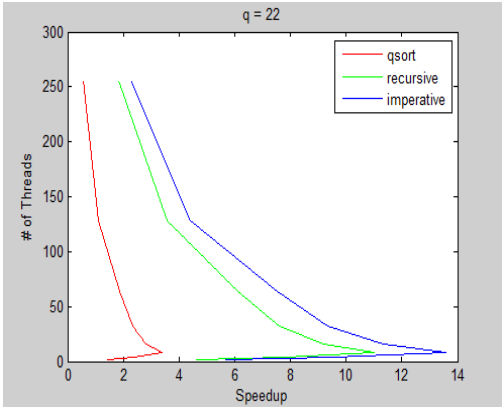


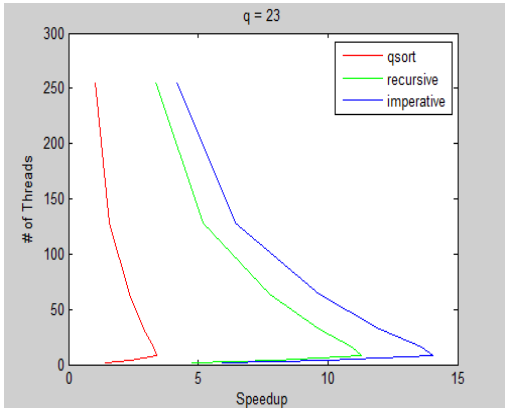


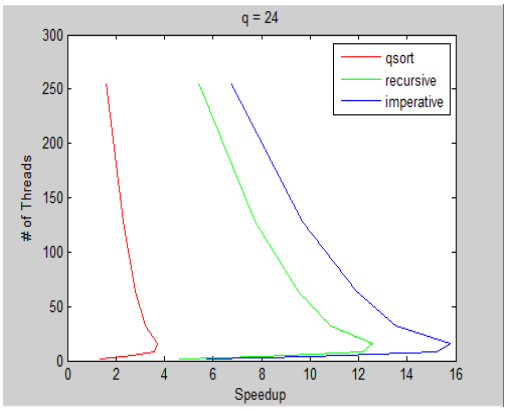




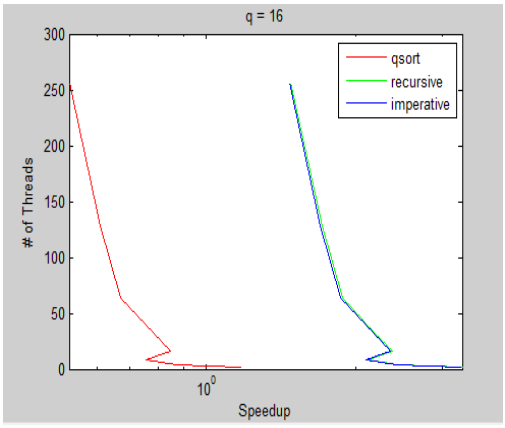


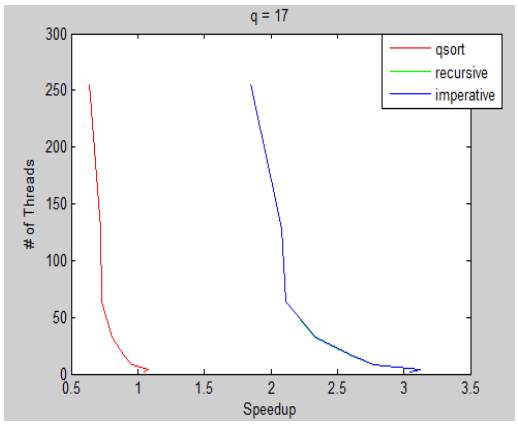


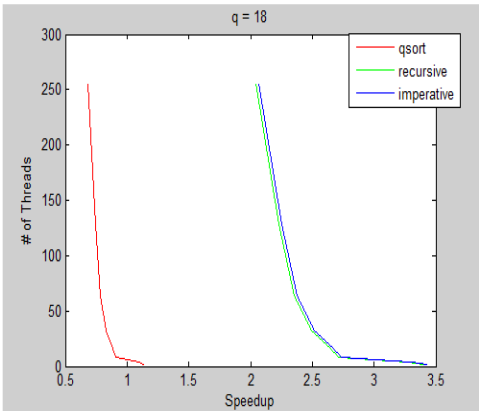


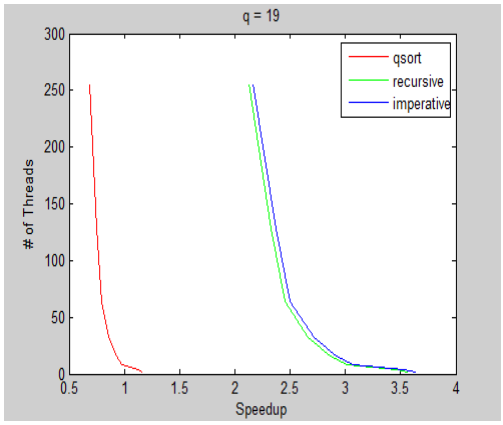


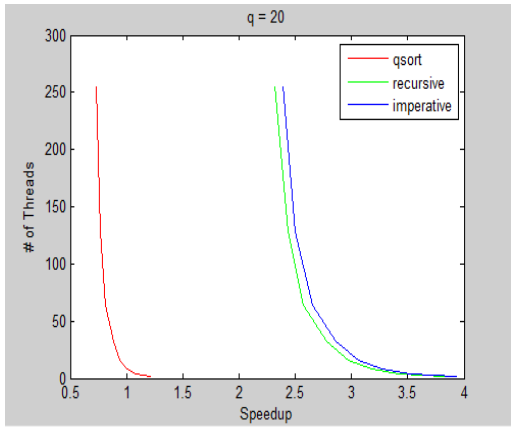
OpenMP (Recursive)

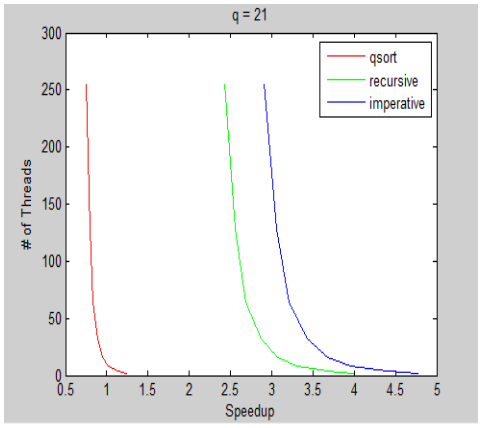


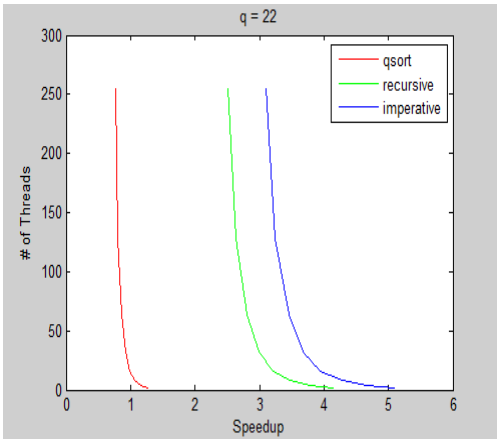


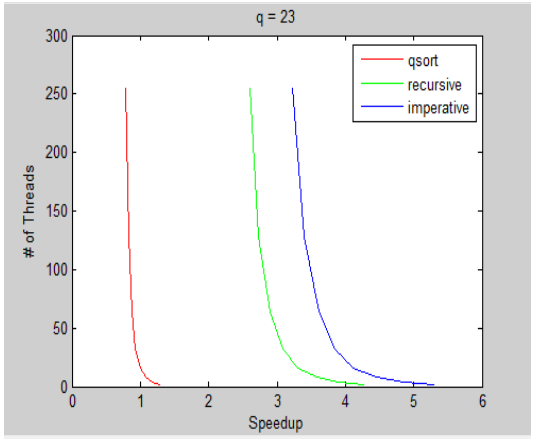


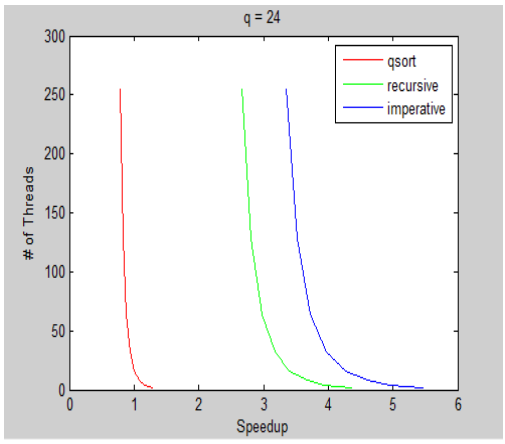




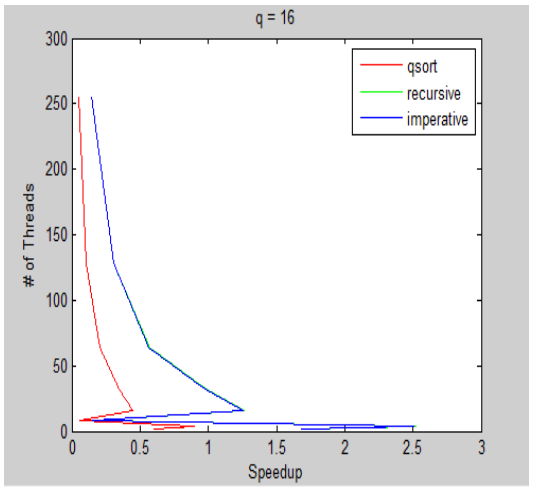


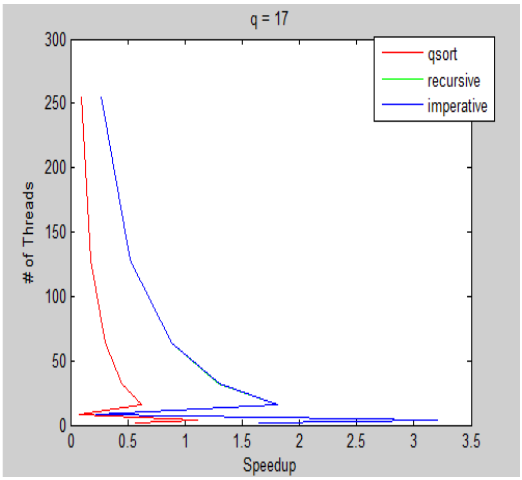


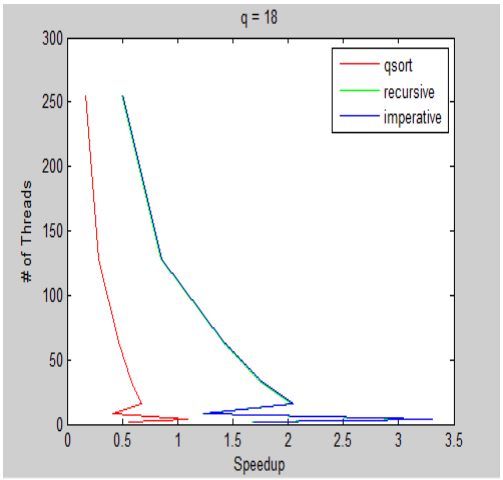


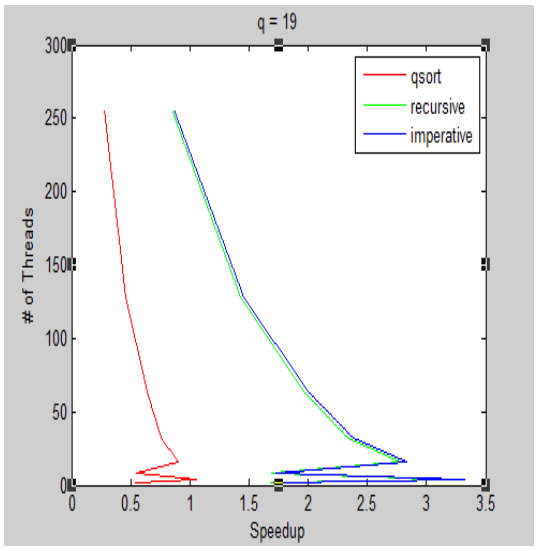


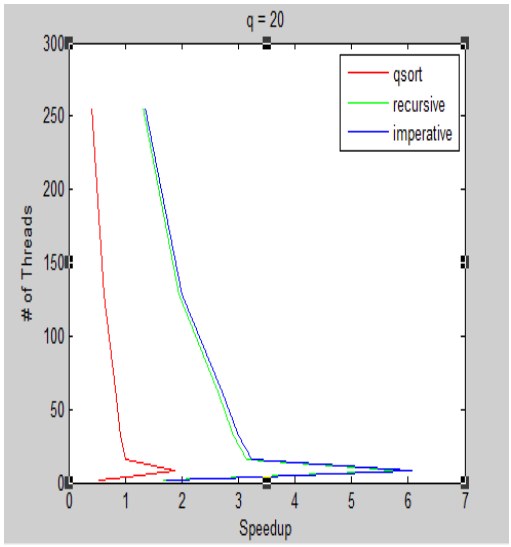
OpenMP (Imperative)

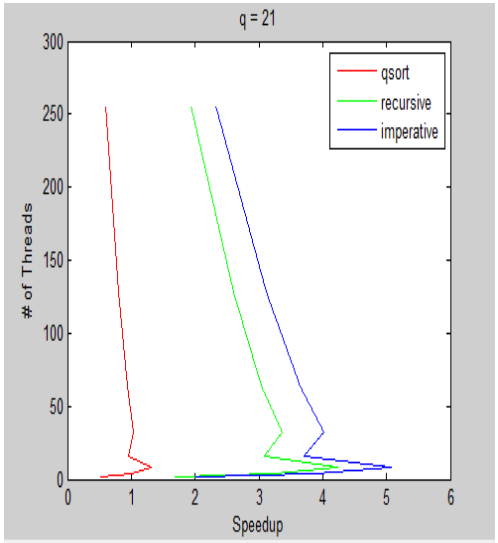


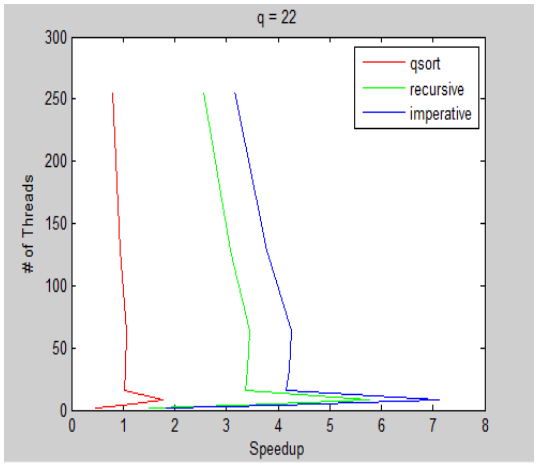


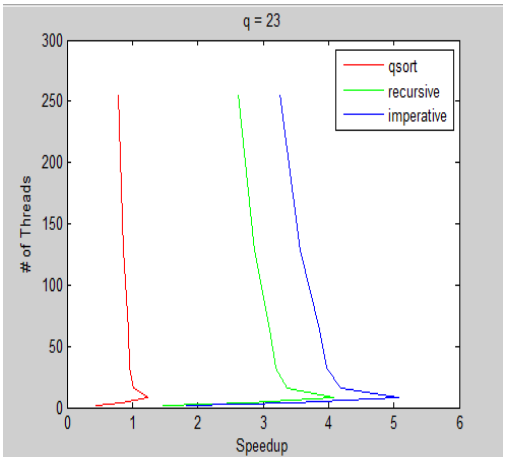


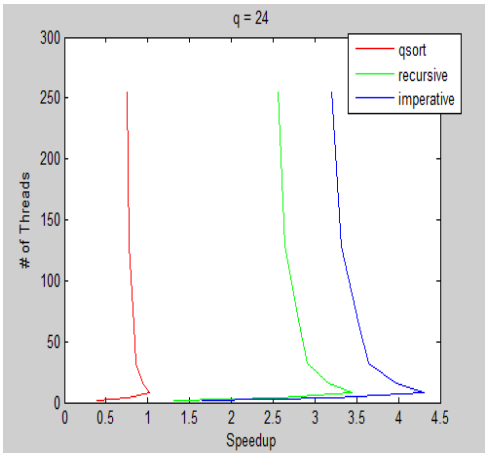




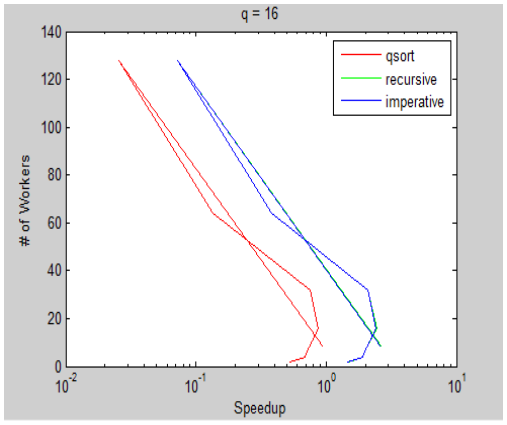


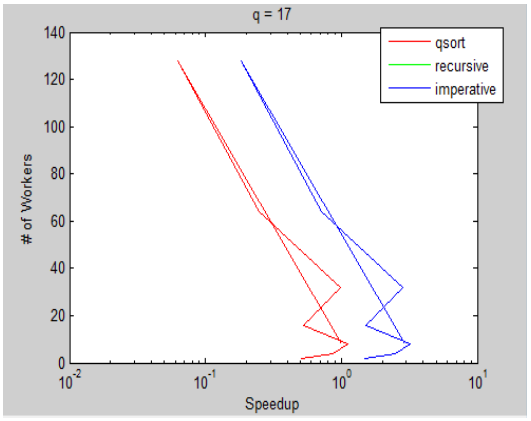


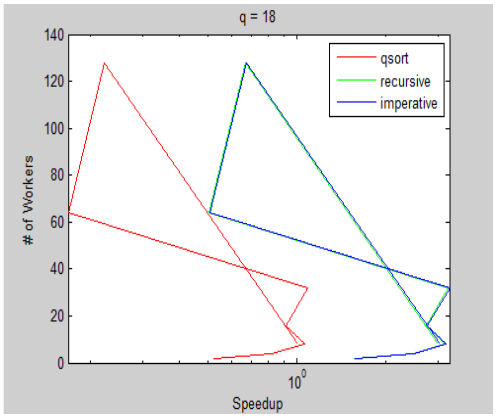


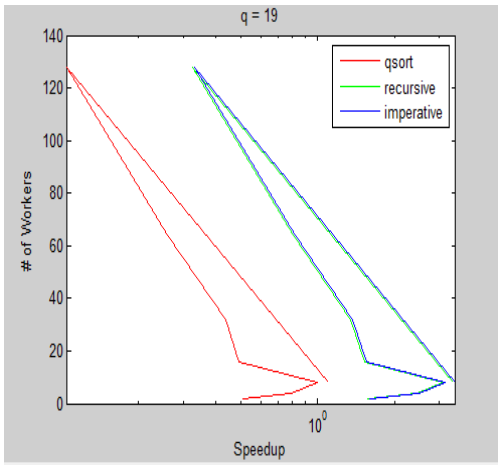


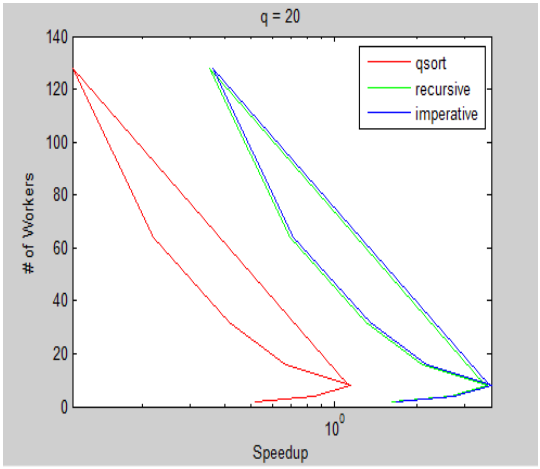
CilkPlus (Recursive)

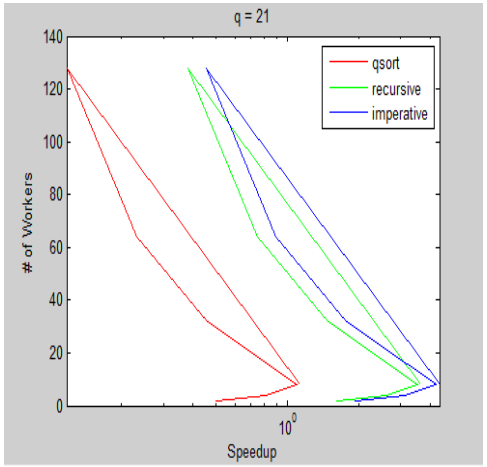


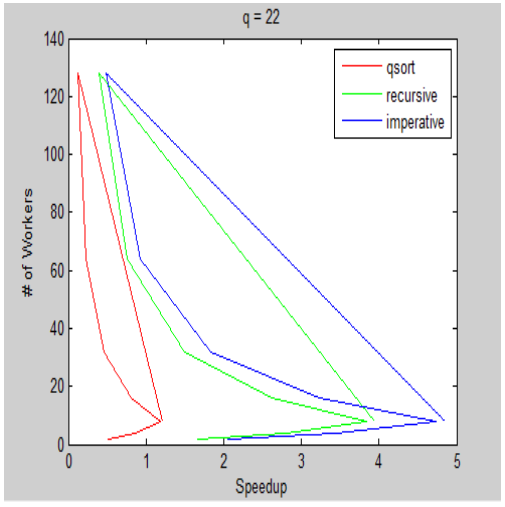


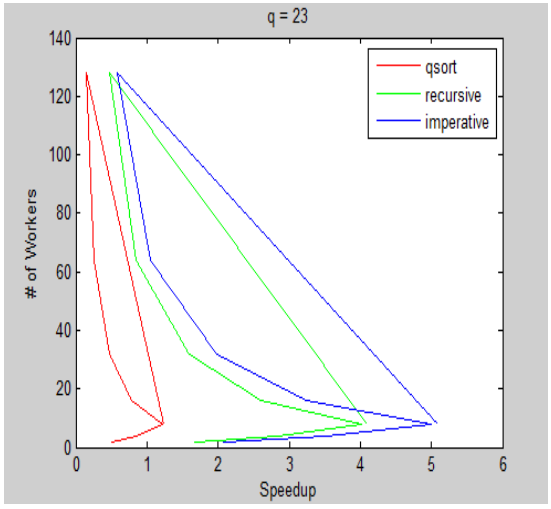


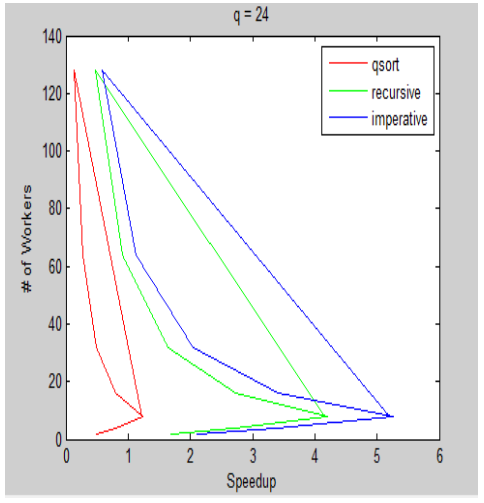




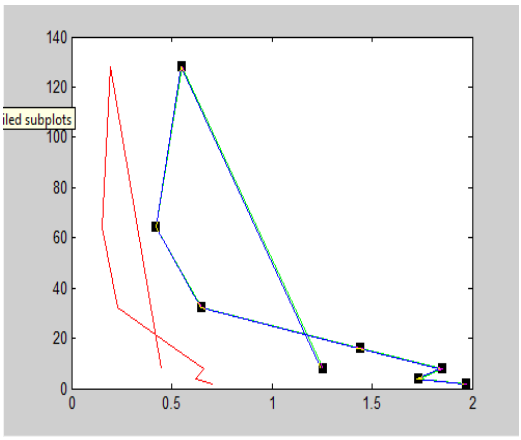


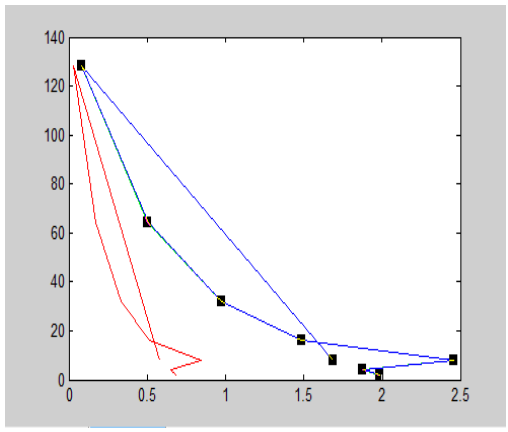


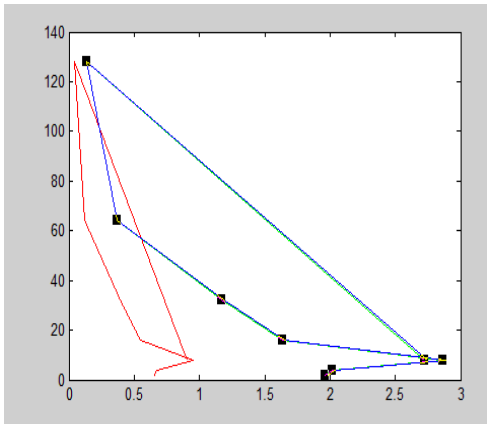


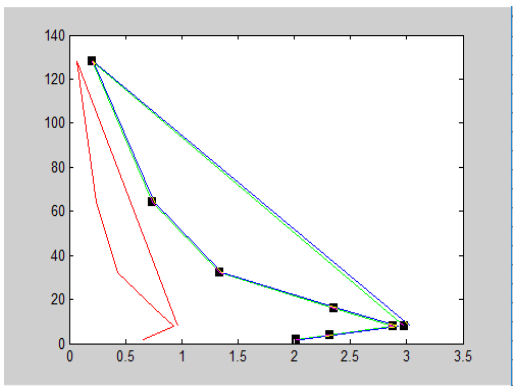


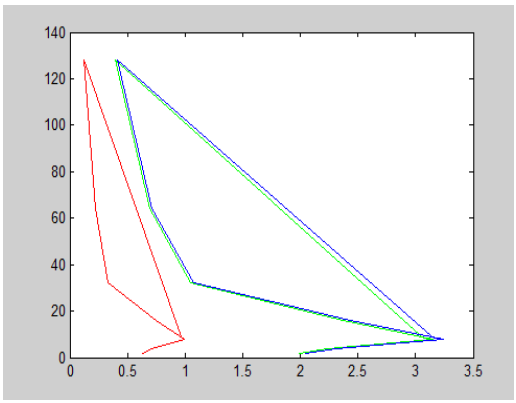
CilkPlus (Imperative)

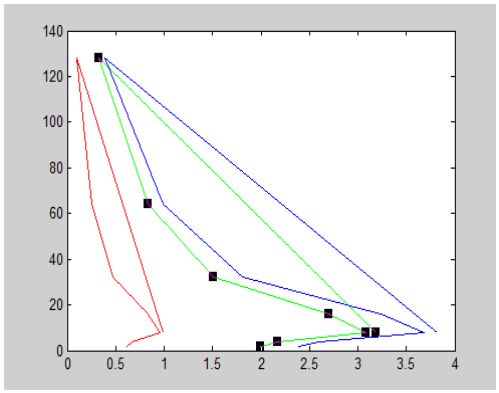


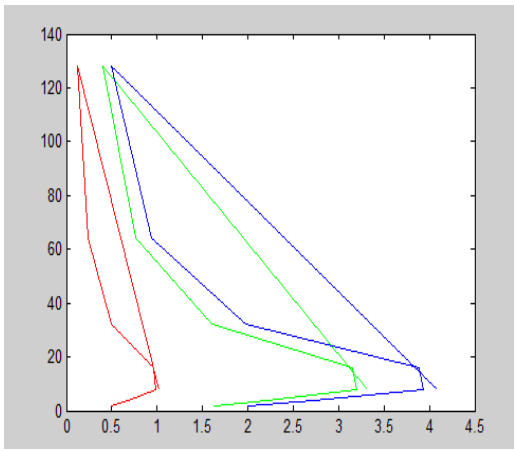


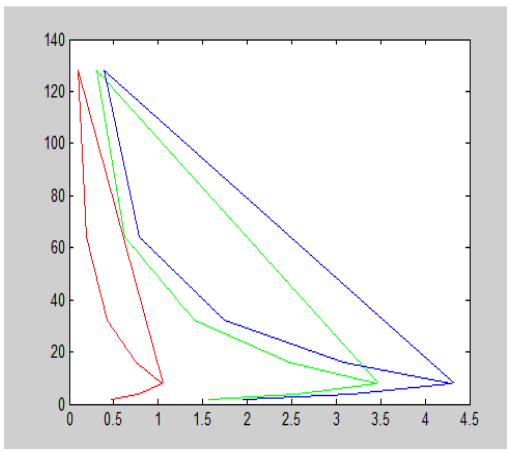


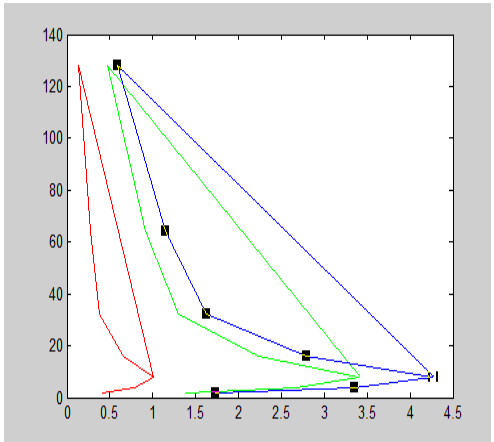












Συμπεράσματα

Βλέποντας τα παραπάνω διαγράμματα μπορούμε να βγάλουμε ορισμένα συμπεράσματα.

Η επιτάχυνση ($S_{latency} = L_{old} / L_{new}$) και στις πέντε εκδόσεις φαίνεται να ακολουθεί τη σχέση qsort < recursive bitonic < imperative bitonic. Δηλαδή, η παραλληλοποίηση αυξάνει την ταχύτητα εκτέλεσης περισσότερο στην imperative (πιο αργή), μετά στην recursive και τέλος στη qsort.

Παρατηρούμε ότι για σχετικά μικρά μήκη πινάκων, μέχρι 2^20 (δηλαδή για q = 20), οι recursive και imperative αλγόριθμοι έχουν παρόμοιες επιταχύνσεις (οι μπλε και οι πράσινες γραμμές ταυτίζονται).

Όταν ο λόγος $S_{latency} = (L_{old} / L_{new}) > 1$ τότε σημαίνει ότι το παράλληλο πρόγραμμα έχει μικρότερο χρόνο εκτέλεσης από το σειριακό του αντίστοιχο, άρα είναι πιο γρήγορο. Όσο πιο μεγάλος ο λόγος, τόσο καλύτερη είναι η βελτίωση που παρέχει το παράλληλο πρόγραμμα. Στην αντίθετη περίπτωση, δηλαδή όταν $S_{latency} < 1$, το παράλληλο είναι πιο αργό από το σειριακό. Έχοντας αυτό υπόψη μπορούμε να διαπιστώσουμε ότι:

Η επιτάχυνση είναι συνήθως βέλτιστη για 8 νήματα (για λιγότερα ή περισσότερα χειροτερεύει). (βλ. Παράρτημα Β)

Η επιτάχυνση είναι καλύτερη όσο αυξάνεται το μήκος του πίνακα (το q δηλαδή), εκεί η παραλληλοποίηση φαίνεται να έχει μεγαλύτερη επίδραση.

Για μικρό αριθμό νημάτων (συνήθως 8) και για όλο και μεγαλύτερο q, ο παράλληλος κώδικας “νικάει” ακόμα και την qsort.

Με αυτά τα διαγράμματα υπάρχει η δυνατότητα σύγκρισης και μεταξύ των διαφορετικών API για συγκεκριμένα q και p.

** Στην υλοποίηση CilkPlus όταν ζητείται να τεθούν 256 workers, δεν πραγματοποιείται αυτή η ενέργεια, αλλά το πρόγραμμα θέτει 8 workers, γεγονός που εξηγεί και τη διαφορετική μορφή του γραφήματος.

ΠΑΡΑΡΤΗΜΑ Α

Επιτάχυνση - Speedup

Στην αρχιτεκτονική υπολογιστών, η επιτάχυνση είναι μια διαδικασία για την αύξηση της απόδοσης μεταξύ δύο συστημάτων που επεξεργάζονται το ίδιο πρόβλημα. Πιο τεχνικά, είναι η βελτίωση της ταχύτητας εκτέλεσης μιας εργασίας που εκτελείται σε δύο παρόμοιες αρχιτεκτονικές με διαφορετικούς πόρους. Η ιδέα της επιτάχυνσης καθορίστηκε από το νόμο του Amdahl, ο οποίος επικεντρώθηκε ιδιαίτερα στην παράλληλη επεξεργασία. Ωστόσο, η επιτάχυνση μπορεί να χρησιμοποιηθεί γενικότερα για να δείξει την επίδραση στην απόδοση μετά από οποιαδήποτε βελτίωση των πόρων. (https://en.wikipedia.org/wiki/Speedup)

Με βάση τα παραπάνω θα χρησιμοποιήσουμε την επιτάχυνση, speedup, για τη σύγκριση των σειριακών υλοποιήσεων (αναδρομική και επαναληπτική) και της stdlib qsort() με τις αντίστοιχες παράλληλες.

Χρησιμοποιώντας το χρόνο εκτέλεσης έχουμε:

$$S_{latency} = L_{old} / L_{new}$$

ΠΑΡΑΡΤΗΜΑ Β

Εκτελώντας την εντολή *lscpu* στο σύστημα diades βλέπουμε χαρακτηριστικά του υπολογιστικού συστήματος στο οποίο τρέχουν τα προγράμματά μας. Κάποια από αυτά τα στοιχεία, όπως ο αριθμός των πυρήνων και αν υποστηρίζεται hyperthreading, μπορούν να μας βοηθήσουν να ερμηνεύσουμε κάποια από τα αποτελέσματά μας. Δεν είναι τυχαίο άλλωστε ότι συνήθως έχουμε βέλτιστη εκτέλεση των προγραμμάτων μας για 8 threads (8 πυρήνες x 1 [δεν υποστηρίζει hyperthreading]).

```
konserpap@diades:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             2
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 23
Model name:            Intel(R) Xeon(R) CPU   E5420 @ 2.50GHz
Stepping:              10
CPU MHz:               2493.842
BogoMIPS:              4987.53
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              6144K
NUMA node0 CPU(s):     0-7
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx lm
constant_tsc arch_perfmon pebs bts rep_good nopl aperfmperf pni dtes64
monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 xsave lahf_lm
tpr_shadow vnmi flexpriority dtherm
```

Πηγές

https://www.cilkplus.org/tutorial-cilk-plus-keywords

https://computing.llnl.gov/tutorials/openMP/

https://computing.llnl.gov/tutorials/pthreads/

https://en.wikipedia.org/wiki/Speedup