

Παράλληλα και Διανεμημένα Συστήματα

Εργασία 1

1) pthread

Για να ταξινομήσω τον μονοδιάστατο πίνακα με τους τυχαίους αριθμούς, χρησιμοποιώντας παραλληλισμό, δημιουργώ δύο νέες συναρτήσεις.

1) *pthread_recBitonicSort*

```

void* pthread_recBitonicSort(void* myData) {
    data *dt;

    dt = (data*)myData;

    int l, c, dr, dpt;
    l = dt->l;
    c = dt->c;
    dr = dt->dir;
    dpt = dt->depth;

    if (c>1) {
        if (dpt < d) {

            pthread_t t1, t2;
            data dt1, dt2;

            dt1.l = l;
            dt1.c = c/2;
            dt1.dir = ASCENDING;
            dt1.depth = dpt+1;

            dt2.l = l+c/2;
            dt2.c = c/2;
            dt2.dir = DESCENDING;
            dt2.depth = dpt+1;

            pthread_create(&t1, NULL, pthread_recBitonicSort, &dt1);
            pthread_create(&t2, NULL, pthread_recBitonicSort, &dt2);

            pthread_join(t1, NULL);
            pthread_join(t2, NULL);

            pthread_bitonicMerge((void*)myData);

        }else{
            qsort(&a[l], c/2, sizeof(int), Asc);
            qsort(&a[l+c/2], c/2, sizeof(int), Desc);
        }
    }
}

```

```

    bitonicMerge(l, c, dr);
  }
}
return 0;
}

```

Η pthread_recBitonicSort είναι void* συνάρτηση και παίρνει σαν όρισμα void*. Το όρισμα που της περνάω είναι struct της μορφής

```

typedef struct {
    int lo;      // starting point
    int cnt;     // length
    int dir;     // direction (asc or desc)
    int depth;   // depth of tree
}data;

```

όπου lo είναι το 1^ο στοιχείο του πίνακα a, cnt το μήκος του, dir η διεύθυνση ταξινόμησης(αύξουσα ή φθίνουσα) και depth το βάθος στο δέντρο που δημιουργείται, λόγω της αναδρομικής διαδικασίας.

Με αυτή τη αναδρομική συνάρτηση δημιουργώ σε κάθε κλίση της δύο threads με την εντολή pthread_create. Το κάθε thread καλεί τη παραπάνω συνάρτηση και έτσι επιτυγχάνεται ο παραλληλισμός. Το 1^ο thread ταξινομεί κατά αύξουσα σειρά το 1^ο μισό του πίνακα, ενώ το 2^ο thread κατά φθίνουσα το 2^ο μισό του πίνακα a και αυξάνουμε το βάθος του δέντρου κατά 1. Στη συνέχεια, περιμένουμε τα threads να τελειώσουν τη δουλειά τους, με την εντολή pthread_join, ώστε να έχουμε σωστούς χρονισμούς και να μην χάνουμε δεδομένα. Έπειτα, καλώ την pthread_bitonicMerge για να ενώσω τα δύο αντίθετα ταξινομημένα κομμάτια σε ένα, με φορά ταξινόμησης αυτήν που ορίζει η μεταβλητή dr.

Σε περίπτωση που το βάθος είναι μεγαλύτερο από τον αριθμό των threads (d = max depth),

```

d = atoi(argv[2]); // 2^d threads - d: max depth

```

καλούμε δύο qsort. Η μία ταξινομεί το 1^ο μισό κατά αύξουσα, ενώ η άλλη το 2^ο μισό κατά φθίνουσα φορά. Στο τέλος, ενώνω τις δύο qsort με τη δοθήσα bitonicMerge.

/* Επέλεξα αυτόν τον τρόπο για (dpt > d) για να μοιάζει με την
recBitonicSort που δόθηκε σαν βάση.

Με τη χρήση μόνο μιας qsort πετυχαίνω καλύτερο χρόνο παρόλα αυτά.

```
/*  
if (dr == ASCENDING){  
    qsort(&a[1], c, sizeof(int), Asc);  
}else{  
    qsort(&a[1], c, sizeof(int), Desc);  
}  
*/  
  
*/
```

2) *pthread_bitonicMerge*

```

void* pthread_bitonicMerge(void *myData) {

    data *dt;

    dt = (data*)myData;

    int l, c, dr, dpt;
    l = dt->l;
    c = dt->c;
    dr = dt->dir;
    dpt = dt->depth;

    if (c > 1) {
        int i, k;
        k=c/2;
        for (i=l; i<l+k; i++){
            compare(i, i+c/2, dr);
        }

        if (dpt < d) {

            pthread_t t1, t2;
            data dt1, dt2;

            dt1.l = l;
            dt1.c = c/2;
            dt1.dir = dr;
            dt1.depth = dpt+1;

            dt2.l = l+c/2;
            dt2.c = c/2;
            dt2.dir = dr;
            dt2.depth = dpt+1;

            pthread_create(&t1, NULL, pthread_bitonicMerge, &dt1);
            pthread_create(&t2, NULL, pthread_bitonicMerge, &dt2);

            pthread_join(t1, NULL);
            pthread_join(t2, NULL);
        } else {

            bitonicMerge(l, c/2, dr);
            bitonicMerge(l+c/2, c/2, dr);
        }
    }

    return 0;
}

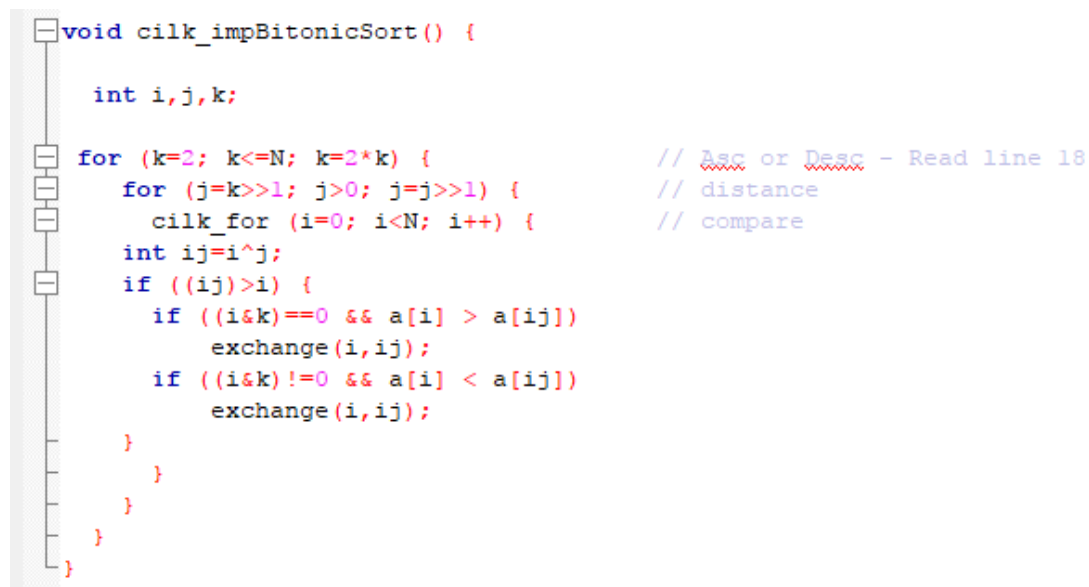
```

Δουλεύω ομοίως με την `pthread_recBitonicSort`. Δηλαδή, υλοποιώ δύο threads κάθε φορά και παραλληλοποιώ το πρόβλημα. Τέλος, ενώνω τα δύο αντίθετα ταξινομημένα κομμάτια σε ένα, με φορά ταξινόμησης αυτήν που ορίζει η μεταβλητή `dr`, όπως προαναφέρθηκε.

2) cilkplus

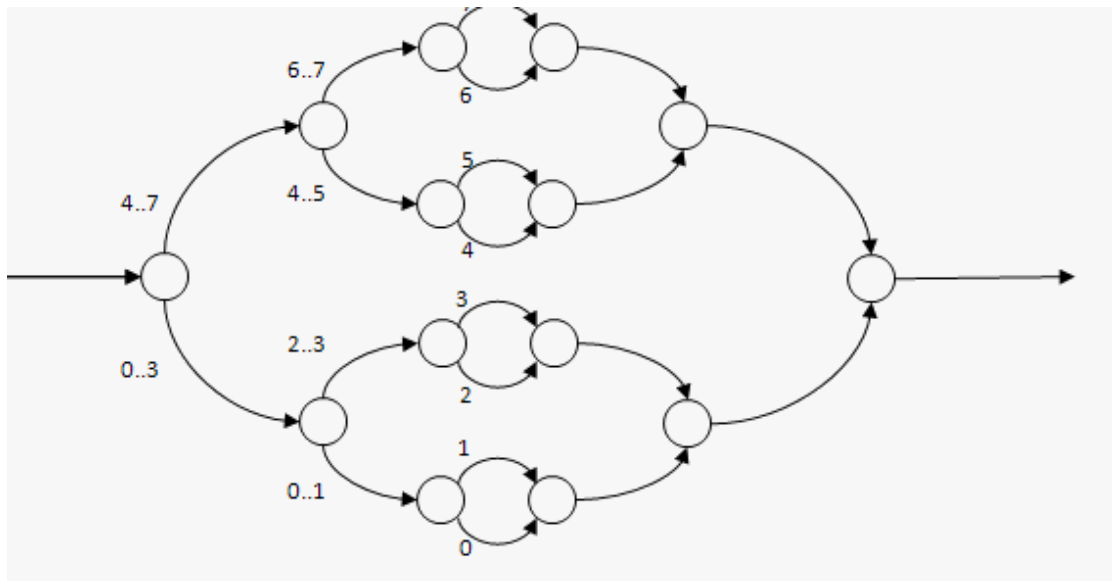
Δημιουργώ τρεις νέες συναρτήσεις.

1) `cilk_impBitonicSort`



```
void cilk_impBitonicSort() {  
    int i,j,k;  
    for (k=2; k<=N; k=2*k) { // Asc or Desc - Read line 18  
        for (j=k>>1; j>0; j=j>>1) { // distance  
            cilk_for (i=0; i<N; i++) { // compare  
                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);  
                }  
            }  
        }  
    }  
}
```

Στην 3^η for γίνεται η σύγκριση και την αντικαθιστώ με την εντολή `cilk_for` η οποία είναι τυποποιημένη και μου παραλληλοποιεί την επαναληπτική διαδικασία(σε αντίθεση με τα `pthread` που η παραλληλοποίηση γίνεται χειροκίνητα). Η `cilk_for` λειτουργεί με τέτοιο τρόπο, όπως φαίνεται γραφικά στο παρακάτω σχήμα



2) *cilk_recBitonicSort*

```

void cilk_recBitonicSort(int lo, int cnt, int dir, int depth) {
    if (cnt>1) {
        int k=cnt/2;

        if (depth < d){
            cilk_spawn cilk_recBitonicSort(lo, k, ASCENDING, depth+1);
            cilk_recBitonicSort(lo+k, k, DESCENDING, depth+1);
            cilk_sync;

            cilk_bitonicMerge(lo, cnt, dir, depth);

        }else{
            qsort(&a[lo], k, sizeof(int), Asc);
            qsort(&a[lo+k], k, sizeof(int), Desc);

            bitonicMerge( lo, cnt, dir);
        }
    }
}

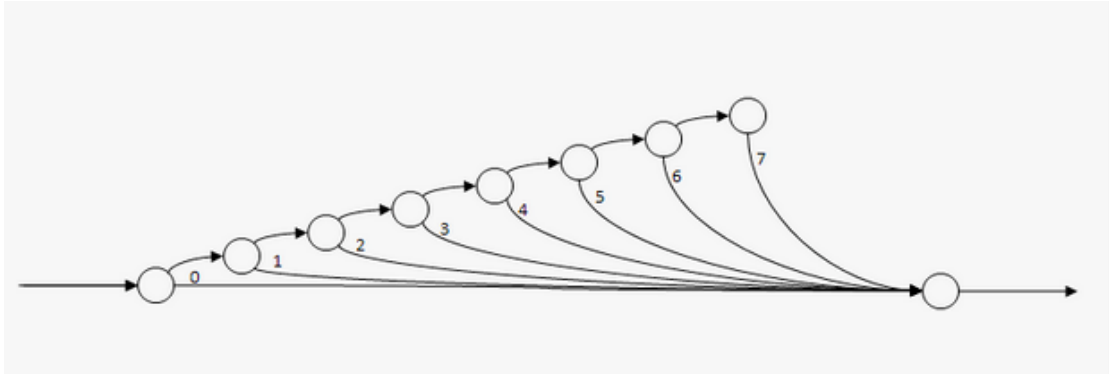
```

Προσθέτω το όρισμα `int depth` όπως ορίστηκε στα `pthread`s(ήταν μέσα σε `struct`).

Η λειτουργία της `cilk_spawn` είναι παρόμοια με την `pthread_create`. Για κάθε συνάρτηση που “βλέπει” από εκεί και κάτω δημιουργεί `thread`. Επίσης, λειτουργία της `cilk_sync` είναι παρόμοια με την `pthread_join`, δηλαδή περιμένει να τελειώσουν την δουλειά τους τα παραπάνω

Threads. Σε περίπτωση που χρειαζόμαστε περισσότερα από τα δυνατά threads, καλώ την qsort για να βελτιώσω την ταχύτητα του κώδικα.

Γραφικά η cilk_spawn έχει αυτή τη μορφή:



3) cilk_btonicMerge

```
void cilk_bitonicMerge(int lo, int cnt, int dir, int depth) {
    if (cnt > 1) {
        int k = cnt / 2;
        int i;

        for (i = lo; i < lo + k; i++) // don't need cilk_for cause 2^p threads already work
            compare(i, i + k, dir);

        if (depth < d) {
            cilk_spawn cilk_bitonicMerge(lo, k, dir, depth + 1);
            cilk_bitonicMerge(lo + k, k, dir, depth + 1);
            cilk_sync;
        } else {
            bitonicMerge(lo, k, dir);
            bitonicMerge(lo + k, k, dir);
        }
    }
}
```

Ομοίως με την pthread_recBitonicSort ενώνω και ταξινομώ τα κομμάτια του πίνακα. Με τη βοήθεια της cilk_spawn πετυχαίνω παράλληλη εκτέλεση.

3) OpenMP

Δουλεύω ακριβώς με τον ίδιο τρόπο με την cilk.

Δημιουργώ τρεις συναρτήσεις.

1) omp_impBitonicSort


```

void omp_impBitonicSort() {
    omp_set_num_threads(1<<d);          // 2^d threads

    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
            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);
                }
            }
        }
    }
}

```

2) omp_recBitonicSort

```

void omp_recBitonicSort(int lo, int cnt, int dir, int depth) {
    if (cnt>1) {
        int k=cnt/2;

        if (depth < d){
            #pragma omp task
            {
                omp_recBitonicSort(lo, k, ASCENDING, depth+1);
            }
            #pragma omp task
            {
                omp_recBitonicSort(lo+k, k, DESCENDING, depth+1);
            }
            #pragma omp taskwait


            omp_bitonicMerge(lo, cnt, dir, depth);

        }else{
            qsort(&a[lo], k, sizeof(int), Asc);
            qsort(&a[lo+k], k, sizeof(int), Desc);

            bitonicMerge(lo, cnt, dir);
        }
    }
}

```

3) *omp_bitonicMerge*



```
void omp_bitonicMerge(int lo, int cnt, int dir, int depth) {  
    if (cnt>1) {  
        int k=cnt/2;  
        int i;  
  
        #pragma omp parallel for  
        for (i=lo; i<lo+k; i++)  
            compare(i, i+k, dir);  
  
        if (depth < d){  
            #pragma omp task  
            {  
                omp_bitonicMerge(lo, k, dir, depth+1);  
            }  
            #pragma omp task  
            {  
                omp_bitonicMerge(lo+k, k, dir, depth+1);  
            }  
  
            #pragma omp taskwait  
        }else{  
            bitonicMerge(lo, k, dir);  
            bitonicMerge(lo+k, k, dir);  
        }  
    }  
}
```

Δουλεύω ακριβώς με τον ίδιο τρόπο με την cilk.

Η #pragma omp parallel for μου παραλληλοποιεί την επαναληπτική διαδικασία for. (Σε αντιστοιχία με την cilk_for)

Η #pragma omp task παραλληλοποιεί την αναδρομική συνάρτηση που βρίσκεται μέσα στο block {}. (Σε αντιστοιχία με την cilk_spawn).

Η #pragma omp taskwait περιμένει να τελειώσουν τα threads την δουλειά τους. (Σε αντιστοιχία με την cilk_sync).

Μετρήσεις:

*** Επειδή δεν έχω laptop, διότι κάηκε η μητρική και η κάρτα γραφικών, αναγκάστηκα να κάνω την εργασία σε υπολογιστή της Βεργίνας (ο οποίος δεν υποστηρίζει cilk) ή στο laptop συναδέλφου. Κατάφερα να κάνω τον κώδικα μου να δουλέψει και να πετύχω ικανοποιητικούς χρόνους, αλλά δεν έχω αποθηκεύσει δείγματα. Γι' αυτό το λόγο, θα προσθέσω τα δείγματα ενός συναδέλφου, τα οποία είναι αρκετά κοντά σε αυτά που πετύχαινε και ο δικός μου κώδικας.

Credits: Κρίστι Νικόλλα

Ευχαριστώ για την κατανόηση! ***

	Recursive	Imperative	Qsort	Recursive with pthreads			
max layer number array size 2^n	—	—	—	1	2	4	8
16	0.0634	0.0492	0.0106	0.0117	0.0131	0.0176	0.1526
20	1.4074	1.1563	0.1717	0.1653	0.1448	0.1958	0.3546
24	30.6799	26.3612	3.3472	2.9331	2.4367	3.2722	5.2071

	Imperative with Cilkplus		Recursive with Cilkplus			
max layer number array size 2^n	—		1	2	4	8
16			0.0165	0.0215	0.0128	0.0105
20			0.2954	0.2267	0.1932	0.1821
24			6.2785	4.0330	3.2797	3.3691

	Imperative with Openmp		Recursive with Openmp			
max layer number array size 2^n	—		1	2	4	8
16			0.0146	0.0141	0.0170	0.0213
20			0.3290	0.1966	0.2457	0.3365
24			7.1114	3.6522	4.6080	6.3954

Παρατηρούμε ότι με χρήση παράλληλου κώδικα η Bitonic Sort μπορεί να ανταγωνιστεί και για 1,2, 4 threads ακόμα και να περάσει την qsort.

Βλέπουμε ότι όταν χρησιμοποιούμε αριθμό Threads κοντά στον αριθμό των πυρήνων της CPU, πετυχαίνουμε καλύτερους χρόνους.

Δημήτρης Παππάς 8391

