# MSc-CNT: Τεχνολογίες Υπολογισμού & Δικτύων
# ΠΑΡΑΛΛΗΛΑ & ΚΑΤΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ
# Άσκηση 1η - Β' Εξάμηνο 2016-17

**Αθανάσιος Ροκόπουλος (16009) – Μιχαήλ Γαλλιάκης (16003)**

***09/05/2017***

## Εκφώνηση:

**Α.**

Αναπτύξτε πρόγραμμα σε OpenMP το οποίο να υπολογίζει παράλληλα τα προθέματα (prefix sum computation) μιας ακολουθίας n στοιχείων A[1..n], σύμφωνα με τον μη-αναδρομικό αλγόριθμο που παρατίθεται στις διαφάνειες 8-9 του Μαθήματος #3 (Αλγόριθμοι Κοινής Μνήμης) που είναι αναρτημένες στο Eclass. Το μέγεθος της ακολουθίας θα πρέπει να το δίνει ο χρήστης από το πληκτρολόγιο. Τα στοιχεία της ακολουθίας αρκεί να είναι ακέραιοι αριθμοί και να παράγονται τυχαία κατά την αρχικοποίηση του προγράμματός σας.

Αρχικά αναπτύξτε και τρέξτε/δοκιμάστε το πρόγραμμά σας θεωρώντας ότι έχετε στη διάθεση σας p=n threads. Στη συνέχεια προσπαθήστε να γενικεύσετε/επεκτείνετε τον ανωτέρω αλγόριθμο υπολογισμού προθεμάτων για n>p και υλοποιήστε τον εκ νέου, θεωρώντας ότι τον αριθμό των threads που θα χρησιμοποιηθούν/εκκινηθούν τον δίνει επίσης ο χρήστης (όπως και το 'n') από το πληκτρολόγιο.

*Δοκιμάστε το πρόγραμμά σας στην υποδομή του εργαστηρίου, και μετρήστε το χρόνο που απαιτείται για διαφορετικά μεγέθη του input (μικρές, μεσαίες και μεγάλες τιμές του 'n') και για διαφορετικό αριθμό threads (π.χ. 1,2,4,8 threads). Δώστε συγκεντρωτικά σε έναν πίνακα και σε αντίστοιχη γραφική παράσταση την επιτάχυνση (speedup) που επιτυγχάνεται σε κάθε περίπτωση.*

**Β.**

Αναπτύξτε πρόγραμμα σε OpenMP το οποίο να ταξινομεί παράλληλα μία ακολουθία n στοιχείων A[1..N], σύμφωνα με τον αναδρομικό αλγόριθμο ταξινόμησης 'multisort' που παρατίθεται στις διαφάνειες #49-50 του αρχείου OpenMP.ppt.

Δοκιμάστε το πρόγραμμά σας στην υποδομή του εργαστηρίου, και μετρήστε το χρόνο που απαιτείται για διαφορετικά μεγέθη του input (μικρές, μεσαίες και μεγάλες τιμές του 'n') και για διαφορετικό αριθμό threads (π.χ. 1,2,4,8 threads). Δώστε συγκεντρωτικά σε έναν πίνακα και σε αντίστοιχη γραφική παράσταση την επιτάχυνση (speedup) που επιτυγχάνεται σε κάθε περίπτωση.

***<u>Παραδοτέα:</u> κώδικας, σχολιασμός/τεκμηρίωση, ενδεικτικά τρεξίματα/αποτελέσματα***

# Μέρος A:

Μέσα στο φάκελο erot1, που βρίσκεται μέσα στο zip αρχείο μαζί με αυτό το pdf, θα βρείτε τα ακόλουθα αρχεία, που αφορούν το πρώτο ερώτημα:

- serialPrefix.cpp (Πρόγραμμα που βρίσκει το Prefix sum, σειριακά)
- parallelPrefix_A.cpp (Βρίσκει το Prefix sum, παράλληλα για n=p [πρώτη προσέγγιση])
- parallelPrefix_A_Opt.cpp (Βρίσκει Pr. sum, παράλληλα για n=p [Βελτιστοποιημένο])
- parallelPrefix_B.cpp (Βρίσ. P.S, παράλληλα για p<n, [πρώτη προσέγγιση])
- parallelPrefix_B_Opt.cpp (Βρίσ. P.S, παράλληλα για p<n, [Βελτιστοποιημένο, μετά από συζήτηση μας..])
- OurLib.h (Δικιά μας βιβλιοθήκη για να έχουμε μικρότερα αρχεία και να μην επαναλαμβάνεται ο κώδικας...)
- run.sh (script για αυτόματο compile και με διάφορες δοκιμές τρεξίματος των παραπάνω προγραμμάτων)
- 2 output (output_2th.txt, output_4th.txt) (Κάποια output από το “run.sh”)

# Μέρος B:

Μέσα στο φάκελο erot2, που βρίσκεται μέσα στο zip αρχείο μαζί με αυτό το pdf, θα βρείτε τα ακόλουθα αρχεία, που αφορούν το πρώτο ερώτημα:

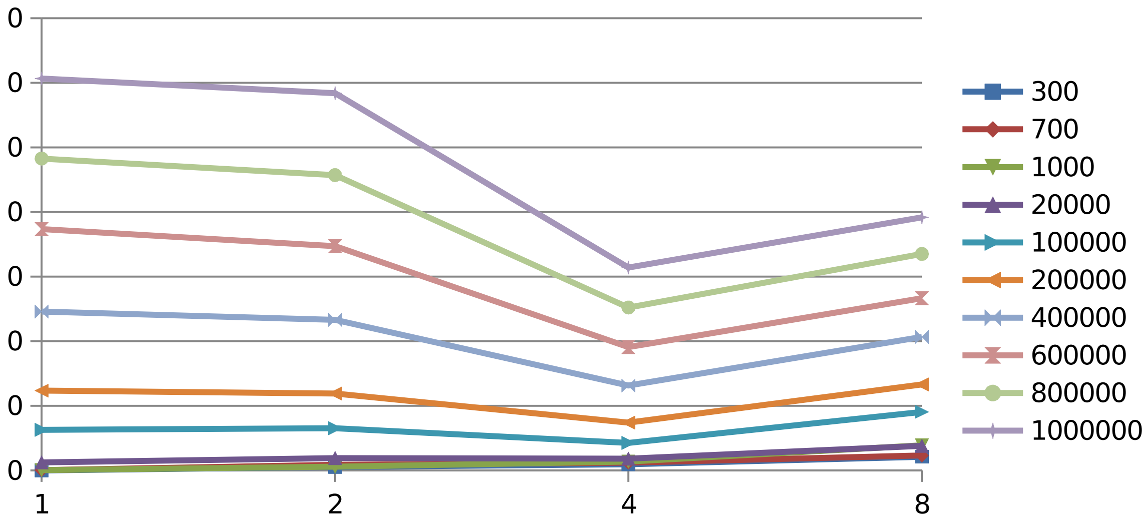
- msort.cpp (Ουσιαστικά το αρχείο που έχετε στο eclass, με τον παράλληλο mergesort)
- mergesort.cpp (O mergesort αλγόριθμος σας [από το msort.cpp], αλλά σειριακός)
- multisort.cpp (Η λύση με βάση τις οδηγίες που μας έχετε δώσει)
- OurLib.h (Δικιά μας βιβλιοθήκη για να έχουμε μικρότερα αρχεία και να μην επαναλαμβάνεται ο κώδικας...)
- run.sh (script για αυτόματο compile και με διάφορες δοκιμές τρεξίματος των παραπάνω προγραμμάτων)
- 3 output(output_2th.txt, output_4th.txt, output_4th_new.txt) (Κάποια output από “run.sh”)

Οι κώδικες **parallelPrefix_B_Opt.cpp** (περιλαμβάνει και τους δύο αλγόριθμους [για n>p και n=p]) και **multisort.cpp**, έχουν σχόλια.
Επίσης, όλα τα παραπάνω αρχεία, μπορείτε να τα βρείτε στο pc9 του εργαστηρίου, μέσα στον λογαριασμό του Θάνου (msc16009).

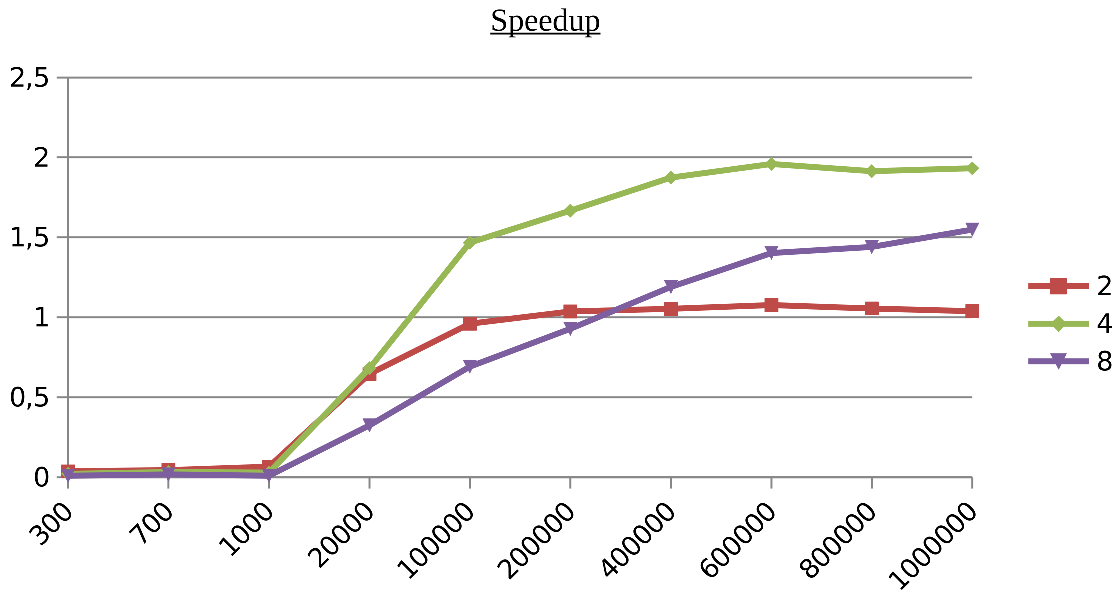
**Ακολουθούν οι μετρήσεις και κάποια screenshots από τους κώδικες που έχουν σχόλια.**

Μετρήσεις για τον αλγόριθμο parallelPrefix_B.cpp, στον 9 του εργαστηρίου:

| **time** | | Σειριακός | Παράλληλα | | |
|---|---|---|---|---|---|
| Δοκιμές | n\threads | 1 | 2 | 4 | 8 |
| 1 | 300 | 0,000001 | 0,000027 | 0,000048 | 0,000107 |
| 2 | 700 | 0,000002 | 0,000045 | 0,00006 | 0,000117 |
| 3 | 1000 | 0,000002 | 0,00003 | 0,000069 | 0,000194 |
| 4 | 20000 | 0,000062 | 0,000096 | 0,000091 | 0,000191 |
| 5 | 100000 | 0,000314 | 0,000327 | 0,000214 | 0,000454 |
| 6 | 200000 | 0,000617 | 0,000595 | 0,00037 | 0,000665 |
| 7 | 400000 | 0,001229 | 0,001166 | 0,000656 | 0,001033 |
| 8 | 600000 | 0,001868 | 0,001736 | 0,000954 | 0,001332 |
| 9 | 800000 | 0,002413 | 0,002285 | 0,001261 | 0,001675 |
| 10 | 1000000 | 0,003032 | 0,002919 | 0,00157 | 0,001958 |




| **speedup** | | Σειριακός | Παράλληλα | | |
|---|---|---|---|---|---|
| Δοκιμές | n\threads | 1 | 2 | 4 | 8 |
| 1 | 300 | 1 | 0,037037 | 0,020833 | 0,009346 |
| 2 | 700 | 1 | 0,044444 | 0,033333 | 0,017094 |
| 3 | 1000 | 1 | 0,066667 | 0,028986 | 0,010309 |
| 4 | 20000 | 1 | 0,645833 | 0,681319 | 0,324607 |
| 5 | 100000 | 1 | 0,960245 | 1,46729 | 0,69163 |
| 6 | 200000 | 1 | 1,036975 | 1,667568 | 0,92782 |
| 7 | 400000 | 1 | 1,054031 | 1,873476 | 1,189739 |
| 8 | 600000 | 1 | 1,076037 | 1,958071 | 1,402402 |
| 9 | 800000 | 1 | 1,056018 | 1,913561 | 1,440597 |
| 10 | 1000000 | 1 | 1,038712 | 1,93121 | 1,548519 |




- ✔ Φαίνεται ότι όσο αυξάνεται το n και ταυτόχρονα εκμεταλλευόμαστε τα φυσικά threads του μηχανήματος (Το pc9 έχει 4 threads), πετυχαίνουμε καλύτερο χρόνο εκτέλεσης (Τα 8 threads τρέχουν ψευδο παράλληλα).

- ✔ Επίσης, διακρίνεται ότι είναι προτιμότερο, εφόσον έχουμε 4 φυσικά threads, να χρησιμοποιήσουμε 8 threads από ότι 2, διότι με τα 8 εκμεταλλευόμαστε και τα 4 threads του μηχανήματος.

- ✔ Αν είναι πολύ μικρό το n είναι προτιμότερο να τρέξουμε τον σειριακό αλγόριθμο.

- ✔ Με 2 threads δεν πετυχαίνουμε πολύ speedup. διότι με βάση τον αλγόριθμο, πρακτικά γίνονται οι ίδιες πράξεις. Δηλαδή πχ [“Χονδρικά”] αν n=1000, ο σειριακός θα έκανε 1000 επαναλήψεις στο for, ενώ ο παράλληλος με 2 threads, θα κάνει 500 επαναλήψεις για ;ένα for παράλληλα και άλλες 500 επαναλήψεις (το 1 thread μόνο χωρίς το “0” που τα έχει έτοιμα τα prefix sum του...) για ένα δεύτερο for. Οπότε είναι αναμενόμενο να κάνει πάνω-κάτω τον ίδιο χρόνο με τον ακολουθιακό αλγόριθμο, για να βρεθούν όλα τα prefix sums.

Παρατηρήσεις:

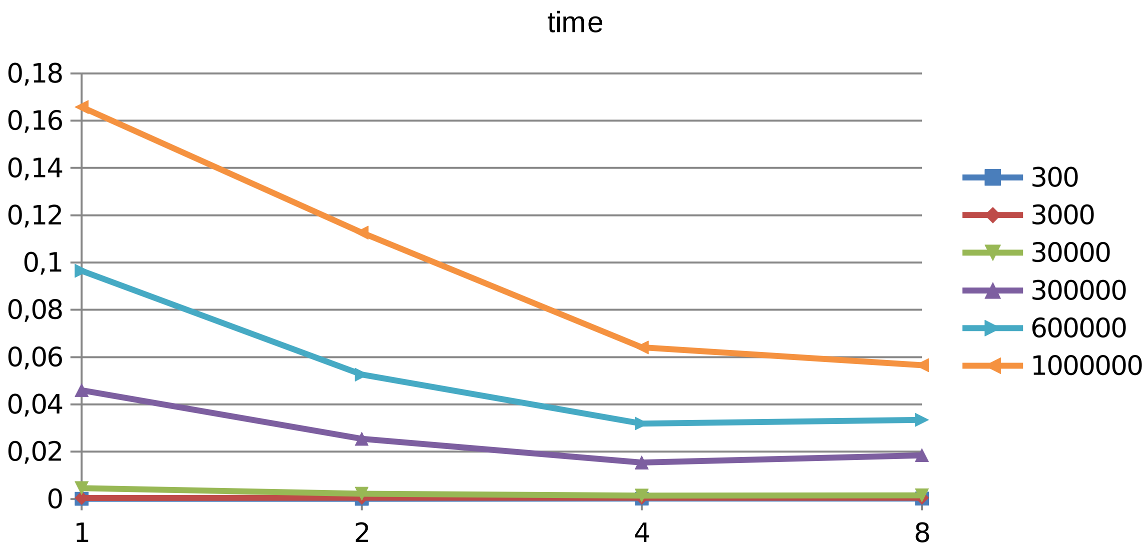
- Δεν φτιάξαμε πινακάκι με μετρήσεις και σχεδιάγραμμα για τον αλγόριθμο parallelPrefix_A_Opt.cpp, γιατί πρακτικά δεν έχουμε μηχάνημα με p=n, ώστε να πετύχουμε ουσιαστικά καλύτερη απόδοση από την λύση με τον σειριακό τρόπο
(Άρα να έχουμε και speedup).

- Στις μετρήσεις είναι ο parallelPrefix_B.cpp και όχι ο parallelPrefix_B_Opt.cpp, διότι αν και είναι σε θεωρητικό επίπεδο βέλτιστος ο δεύτερος, πρακτικά λόγω του ότι δεν έχει πολλά threads το μηχάνημα, είναι ελάχιστα πιο γρήγορος ο πρώτος.

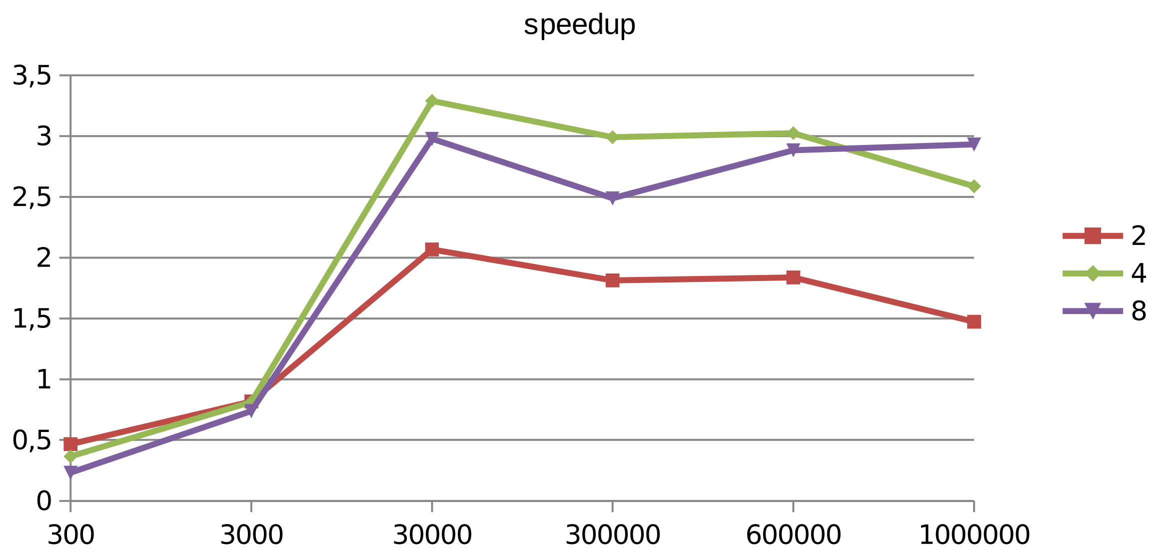
*[Δεν γράφουμε περισσότερα, γιατί τα έχουμε συζητήσει λίγο και από κοντά και μπορούμε να τα ξανασυζητήσουμε και πάλι στην εξέταση της άσκησης όταν θα γίνει].*

Μετρήσεις για τον αλγόριθμο multisort.cpp, στον 9 του εργαστηρίου:

| time | | Σειριακός | Παράλληλα | | |
|---|---|---|---|---|---|
| Δοκιμές | n\threads | 1 | 2 | 4 | 8 |
| 1 | 300 | 0,000041 | 0,000088 | 0,000112 | 0,000177 |
| 2 | 3000 | 0,000374 | 0,000458 | 0,00046 | 0,000507 |
| 3 | 30000 | 0,004547 | 0,002199 | 0,001382 | 0,001527 |
| 4 | 300000 | 0,046021 | 0,025386 | 0,015388 | 0,018494 |
| 5 | 600000 | 0,096549 | 0,052561 | 0,031921 | 0,033475 |
| 6 | 1000000 | 0,165766 | 0,112617 | 0,06407 | 0,056537 |




| speedup | | Σειριακός | Παράλληλα | | |
|---|---|---|---|---|---|
| Δοκιμές | n\threads | 1 | 2 | 4 | 8 |
| 1 | 300 | 1 | 0,465909 | 0,366071 | 0,231638 |
| 2 | 3000 | 1 | 0,816594 | 0,813043 | 0,737673 |
| 3 | 30000 | 1 | 2,067758 | 3,290159 | 2,977734 |
| 4 | 300000 | 1 | 1,81285 | 2,990707 | 2,488429 |
| 5 | 600000 | 1 | 1,836894 | 3,024623 | 2,884212 |
| 6 | 1000000 | 1 | 1,471945 | 2,587264 | 2,931991 |




- ✔ Φαίνεται ότι όσο αυξάνεται το n και ταυτόχρονα εκμεταλλευόμαστε τα threads του μηχανήματος, πετυχαίνουμε καλύτερο χρόνο εκτέλεσης. Που σημαίνει ότι με παράλληλο τρόπο μπορούμε να λύσουμε το ίδιο πρόβλημα (ταξινόμηση) πιο γρήγορα από ότι με τον παραδοσιακό ακολουθιακό τρόπο.
- ✔ Επίσης, διακρίνεται ότι είναι προτιμότερο, εφόσον έχουμε 4 φυσικά threads, να χρησιμοποιήσουμε 8 threads από ότι 2, διότι με τα 8 εκμεταλλευόμαστε και τα 4 threads του μηχανήματος.
- ✔ Αν είναι πολύ μικρό το n είναι προτιμότερο να τρέξουμε τον σειριακό αλγόριθμο.

Παρατηρήσεις:

- Οι συγκρίσεις του multisort, έγιναν με τον ακολουθιακό mergesort (Πρακτικά είναι ο παράλληλος που μας δώσατε [msort.c] αλλά χωρίς τις “εντολές” omp για παραλληλία...)
- Το limit για το παραπάνω πινακάκι είναι 10000
- Έγιναν αρκετές δοκιμές με διαφορετικά limit ώστε να δούμε έμπρακτα την διαφορά στην απόδοση κάθε φορά. (Περισσότερα στα αρχεία text από το run.sh file).
- Επειδή κάθε φορά που τρέχουμε το multisort, μπορεί να κάνει λίγο διαφορετικούς χρόνους κάθε φορά (πχ ανάλογα και με τον φόρτο του συστήματος εκείνη την ώρα), φαίνεται ότι τα 8 threads για n=1000000 είναι προτιμότερα. Κάτι το οποίο δεν ισχύει λογικά στην γενική περίπτωση, γιατί το μηχάνημα έχει 4 threads... [Το σχολιάσαμε και στην τάξη αλλά δεν αλλάξαμε τις μετρήσεις και το σχεδιάγραμμα...]

<u>3 screenshots από το parallelPrefix_B.cpp:</u>

parallelPrefix_B_Opt.cpp ×

```cpp
#include "OurLib.h"

void calcPrefixSumNequalsP(int *(&),int) ;
int main(int argc, char *argv[])
{
    //cout<<"It has " << std::thread::hardware_concurrency()<<" thread(s) the machine!\n";

    int n = 0 ;
    int threads = 0 ;
    double ctime1, ctime2;

    if (argc > 1)
        stringstream(argv[1]) >> n; //περνιέται παραμετρικά το n
    if (n==0)
        n = readNum() ; //Δείνει ο χρήστης το n με την βοήθεια της readNum

    if (argc > 2)
        stringstream(argv[2]) >> threads; //περνιέται παραμετρικά ο αριθμός των threads
    if (threads<=0)
        threads = readNum("threads",0,n) ;  //Δείνει ο χρήστης τον αριθμό των threads
    else if (threads>n)                     //με την βοήθεια της readNum
        threads = n ;   // Αν threads είναι περισσότερα από το n τότε να γίνεται threads=n ;

    omp_set_num_threads(threads);// Δηλώνουμε πόσα threads θα έχουμε στο openMP.

    int A[n] ;  //Δημιουργείται ο πίνακας A (Αρχικός).
    fillArrayWithRandNum(A,n) ; //Γεμίζει ο πίνακας A με τυχαίους αριθμούς από το 1-10.
    //printArray(A,n) ;

    int tid, prevPrefIndex;
    int logos = n/threads ;   // υπολογίζεται ο λόγος
    int dr = n % threads ;    // υπολογίζεται το υπόλοιπο της διαίρεσης
    int B[n] ; //Δημιουργείται ο πίνακας B (Για αποτελέσματα).
    int *lastValues = new int[threads] ; //Δημιουργείται ο πίνακας για τις τελευταίες τιμές από το κάθε thread...
    int begin[threads],end[threads] ; //Φτιάχνονται 2 πίνακες που θα παίρνουν τους "δείκτες" αρχής και τέλους
                                      //ώστε να μπορεί κάθε thread να υπολογίζει σειριακά το δικό του κομμάτι.
```

parallelPrefix_B_Opt.cpp ×

```cpp
    int begin[threads],end[threads] ; //Φτιάχνονται 2 πίνακες που θα παίρνουν τους "δείκτες" αρχής και τέλους
                                      //ώστε να μπορεί κάθε thread να υπολογίζει σειριακά το δικό του κομμάτι.

    ctime1 = omp_get_wtime(); //Ξεκινάμε να χρονομετράμε τον αλγόριθμο:
    #pragma omp parallel shared(A,B,n) private(tid)
    {
        tid = omp_get_thread_num();  //Κάθε thread παίρνει το id του
        begin[tid] = tid*logos + ((tid<dr)?tid:dr);  //Υπολογίζεται ο δείκτης αρχής για κάθε thread
        end[tid] = begin[tid] + logos  + ((tid<dr)?1:0) -1 ; //Υπολογίζεται ο δείκτης τέλους για κάθε thread
                                                             //Υπολογίζεται πχ για n=18 & th= 4, θα είναι 5,5,4,4
        B[begin[tid]]=A[begin[tid]] ;   //Ο πρώτος αριθμός δεν χρειάζεται να υπολογιστεί.
        for (int i=begin[tid]+1 ; i <= end[tid]; i++) //Από τον δευτερο αριθμό μέχρι το τελευταίο
            B[i] = B[i-1] + A[i];                     //για κάθε thread, υπολογίζεται το PrefixSum σειριακά.
        lastValues[tid] = B[end[tid]] ; //Ο πίνακας lastValues στην θέση του tid παίρνει τον τελευταίο υπολογισμένο Prefix
    }       //Σταματάει η παραλληλία
    calcPrefixSumNequalsP(lastValues, threads); //Υπολογίζονται τα prefix του lastValues με τον αλγόριθμο για n=p.
    #pragma omp parallel shared(A,B,n) private(tid,prevPrefIndex)
    {
        tid = omp_get_thread_num();  //Κάθε thread παίρνει το id του πάλι
        if (tid>0) //Το 0 thread έχει έτοιμα τα Prefix του, οπότε δεν κάνει κάτι
        {
            prevPrefIndex = tid-1 ; //Υπολογίζεται ο δείκτης για τον lastValues.
            for (int i=begin[tid] ; i <= end[tid]; i++) // Κάθε thread, για το κομμάτι του πίνακα που τον αφορούν
                B[i] += lastValues[prevPrefIndex];      //υπολογίζει τα τελικά prefix με την βοήθεια του lastValues.
        }
    }  //Σταματάει η παραλληλία
    ctime2 = omp_get_wtime();//Σταματάμε να χρονομετράμε τον αλγόριθμο!
    if (argc <= 3){ //Εμφανίζονται τα μηνύματα αν έχουν δοθεί λιγότεροι από τρεις παραμέτροι.
        cout<<"******** Final ***********" <<"\n" ;
        printArray(A,n) ;
        printArray(B,n) ;
        cout<<"********  End  ***********" <<"\n" ;
    }
    cout<<fixed<<"[Parallel_B_Opt]time: "<<ctime2-ctime1<<" n= "<<n //Εμφανίζεται μήνυμα με χρόνο, n, threads,φυσικά threads.
        <<" thr= "<<threads<<"(ph:"<<std::thread::hardware_concurrency()<<")\n";
    return 0 ;
}

void calcPrefixSumNequalsP(int *(&Array),int n) //Συνάρτηση που υπολογίζει τα prefixSum για n=p
{
```

```cpp
parallelPrefix_B_Opt.cpp
    return 0 ;
}

void calcPrefixSumNequalsP(int *(&Array),int n) //Συνάρτηση που υπολογίζει τα prefixSum για n=p
{
    int **A = new int*[2] ;
    A[0] = Array ;
    A[1] = new int[n];

    int chunk = 1 ; // Θέλουμε chunk = 1, γιατί έχουμε n=p όποτε σε κάθε επανάληψη
                    // του for να κάνει κάθε thread από μόνο μια πράξη.

    int repetitions = ceil(log2(n)); // Βρίσκονται οι επαναλήψεις-βήματα
    int jump = 1; //Στην αρχή το jump είναι 1
    int tid;
    int index = 0 ;  //Δείκτης για να γίνονται εναλλαγές μεταξύ των 2 πινάκων πρακτικά.
    for (int k=1;k<=repetitions;k++){ // Για όσα είναι τα βήματα
        #pragma omp parallel shared(A,n,repetitions,chunk,jump) //Τρέχουμε παράλληλα
        {
            #pragma omp for schedule(static,chunk)
            for (int i=jump/2; i < n; i++) //Το jump είναι δια 2, γιατί υπάρχουν ήδη τα
            {                              //τα υπολογισμένα prefix μέχρι εκεί...
                //tid = omp_get_thread_num();
                if (i<jump)
                    A[!index][i] = A[index][i]; //Αντιγράφονται απλά(με παράλληλο τρόπο), γιατί γνωρίζουμε το Prefix
                else
                    A[!index][i] = A[index][i] + A[index][i-jump];  //Υπολογίζεται το Prefix παράλληλα
                //printInfo(tid,k,i,A[!index][i]) ;
            }
        }
        jump <<= 1 ; //Γίνεται shift κατά μια θέση στο δυαδικό, οπότε το jump παίρνει την αμέσως
                     //επόμενη δύναμη του 2 από αυτό που ήταν (πχ από 100 [4] -> 1000 [8]).
        index = !index ;    //Ο δείκτης πρακτικά αλλάζει (0->1,1->0) ώστε μετά από κάθε βήμα, ο πίνακας
                            //των αποτελεσμάτων, να γίνεται αρχικός για την επόμενη επανάληψη.
    }
    Array = A[index]; //Επιστρέφεται πρακτικά ο πίνακας με τα αποτελέσματα...
}
```

<u>4 screenshots από το multisort.cpp:</u>

```cpp
multisort.cpp
 * OpenMP - Erotima B - Prefix
 * Date: 2/05/2017
 */
#include "OurLib.h"

#define DEFAULTLIMIT 10000

int Limit = DEFAULTLIMIT ;

void multisort(int*, long, int*) ;

int main(int argc, char *argv[])
{
    int n = 0;
    int threads = 0 ;
    double ctime1, ctime2;
    //cout<<"It has " << std::thread::hardware_concurrency()<<" thread(s) the machine!\n";

    if (argc > 1)
        stringstream(argv[1]) >> n; //περνιέται παραμετρικά το n
    if (n<=0)
        n = readNum() ; //Δείνει ο χρήστης το n με την βοήθεια της readNum

    if (argc > 2)
        stringstream(argv[2]) >> threads; //περνιέται παραμετρικά ο αριθμός των threads
    if (threads==0)
        threads = readNum("threads",0,n) ; //Δείνει ο χρήστης τον αριθμό των threads
    else if (threads<0)                    //με την βοήθεια της readNum
        threads = std::thread::hardware_concurrency() ; // Αν η μεταβλητή threads είναι αρνητική (έχει περαστεί παραμετρικά)
                                                        // τότε τα threads θα είναι όσα έχει το μηχάνημα που τρέχει το πρόγραμμα.
    if (argc > 3)
        stringstream(argv[3]) >> Limit;  //περνιέται παραμετρικά το Limit
    if (Limit<=0)
        Limit = DEFAULTLIMIT ;  //Αν το limit είναι είναι μικρότερο του 1, τότε το limit είναι το default.

    int data[n], tmp[n];

    omp_set_num_threads(threads);  // Δηλώνουμε πόσα threads θα έχουμε στο openMP.
    generate_list(data, n); //υπολογίζονται 0-n αριθμοί και διασκορπίζονται στον πίνακα data.
    if (argc <= 4){ //Δεν εμφανίζεται ο αρχικός, μη ταξινομημένος πίνακας, αν έχουμε 4 ή περισσότερους παραμέτρους.
        cout<<"******** Final ***********" <<"\n" ;
        printArray(data, n);
    }
    ctime1 = omp_get_wtime();  //Ξεκινάμε να χρονομετράμε τον αλγόριθμο:
    #pragma omp parallel
    {
    #pragma omp single
        multisort(data, n, tmp);
    }
    ctime2 = omp_get_wtime(); //Σταματάμε να χρονομετράμε τον αλγόριθμο!
    if (argc <= 4){  //Δεν εμφανίζεται ο τελικός, ταξινομημένος πίνακας, αν έχουμε 4 ή περισσότερους παραμέτρους.
        printArray(data, n);
        cout<<"********  End  ***********" <<"\n" ;
    }

    cout<<fixed<<"[Multisort]time: "<<ctime2-ctime1<<" n= "<<n //Εμφανίζεται μήνυμα με χρόνο, n, threads,φυσικά threads, limit.
            <<" thr= "<<threads<<"(ph:"<<std::thread::hardware_concurrency()<<") ";
    cout<<"Limit="<<Limit<<endl ;
    return 0 ;
}
/*
```

multisort.cpp ×

```
void merge(int * X, long n, int * tmp) {
    int i = 0;
    int j = n/2;
    int ti = 0;

    while (i<n/2 && j<n) {
        if (X[i] < X[j]) {
            tmp[ti] = X[i];
            ti++; i++;
        } else {
            tmp[ti] = X[j];
            ti++; j++;
        }
    }
    while (i<n/2) { /* finish up lower half */
        tmp[ti] = X[i];
        ti++; i++;
    }
    while (j<n) { /* finish up upper half */
        tmp[ti] = X[j];
        ti++; j++;
    }
    memcpy(X, tmp, n*sizeof(int));

} // end of merge()

/*
    Είναι ακριβώς η ίδια που μας δώσατε, με διαφορά ότι είναι ακολουθιακή και όχι παράλληλη. Χρησιμοποιείται
    στην περίπτωση που το n στην multisort είναι μικρότερο από ένα καθορισμένο όριο. Οπότε πρακτικά
    ταξινομεί ένα πίνακα X, n στοιχείων.
*/
void mergesort(int * X, int n, int * tmp)
{
    if (n < 2) return;

    mergesort(X, n/2, tmp);
    mergesort(X+(n/2), n-(n/2), tmp);

    /* merge sorted halves into sorted list */
    merge(X, n, tmp);
}
```

multisort.cpp ×

```
void multisort(int *X, long n, int * tmp)
{
    if (n < Limit)  //Το limit είναι Global μεταβλητή.
    {               //Αν το n είναι μικρότερο από ένα καθορισμένο όριο τότε
                    //εκτελείτε η σειριακή mergesort για τα n(<limit) στοιχεία.
        mergesort(X, n, tmp);
        /*
        Αντί για την quicksort, χρησιμοποιήσαμε την mergesort (σειριακή) όταν το n,
        μετά τις αναδρομικές κλήσεις, είναι μικρότερο από κάποιο όριο (limit).
        Επειδή την είχαμε έτοιμη (mergesort) από το παράδειγμα σας στο eclass (msort.c).
        */
        return ;
    }

    int qua = (n/4) ; //Ένα τέταρτο του n
    int dr = n%4 ; //Υπόλοιπο διαίρεσης
    int partA = qua ; //Πρώτο κομμάτι
    int partB = qua + ((dr>1)?1:0); //Δεύτερο κομμάτι
    int partC = qua + ((dr>2)?1:0) ; //Τρίτο κομμάτι
    int partD = qua + ((dr>0)?1:0); //Τέταρτο κομμάτι

    //Για να γίνει το merge πρέπει τα δύο μισά του πίνακα να είναι ταξινομημένα.
    //Οπότε, αν το υπόλοιπο της διαίρεσης (1,2,3) το πάρει ένα οποιοδήποτε από τα 3 μέρη (πχ partB)
    //υπάρχει πρόβλημα! Στην merge αν είναι ζυγός το n, πχ n=12, καταλαβαίνει 6 & 6 ταξινομημένα και
    //αν είναι περιττός το n, πχ n=13 καταλαβαίνει 6 & 7 (Μια μονάδα το πολύ μεγαλύτερο το 2ο από το 1ο).
    //Οπότε δεν μπορείς πχ να έχεις 5 και 7 ταξινομημένα στοιχεία...
    //Ο παραπάνω τρόπος που μοιράζει το υπόλοιπο της διαίρεσης λύνει το πρόβλημα, γιατί:
    // Αν πχ n = 1001 τότε [pa=250, pb=250], [pc=250, pd=251] -> 500 , 501 (Τελικό merge)
    // Αν πχ n = 1002 τότε [pa=250, pb=251], [pc=250, pd=251] -> 501 , 501 (Τελικό merge)
    // Αν πχ n = 1003 τότε [pa=250, pb=251], [pc=251, pd=251] -> 501 , 502 (Τελικό merge)
    // Αν πχ n = 1004 τότε [pa=251, pb=251], [pc=251, pd=251] -> 502 , 502 (Τελικό merge)

    //Πρακτικά υπολογίζονται οι 4 υποπίνακες του X, ώστε να κληθούν στις 4 αναδρομικές κλήσεις.
    int *XA = X ;
    int *XB = XA+partA;
    int *XC = XB+partB ;
    int *XD = XC+partC ;
    //Πρακτικά υπολογίζονται οι 4 υποπίνακες του tmp, ώστε να κληθούν στις 4 αναδρομικές κλήσεις.
    int *tmpA = tmp ;
    int *tmpB = tmpA+partA;
    int *tmpC = tmpB+partB ;
    int *tmpD = tmpC+partC ;

    //cout<< omp_get_thread_num()<<endl ;

    //Εκτελούνται παράλληλα (σε tasks) οι 4 αναδρομικές κλήσεις, για κάθε τέταρτο του X.
    #pragma omp task firstprivate (XA, partA, tmpA)
    multisort(XA, partA, tmpA);

    #pragma omp task firstprivate (XB, partB, tmpB)
    multisort(XB,partB, tmpB);
```

```
    //Πρακτικά υπολογίζονται οι 4 υποπίνακες του X, ώστε να κληθούν στις 4 αναδρομικές κλήσεις.
    int *XA = X ;
    int *XB = XA+partA;
    int *XC = XB+partB ;
    int *XD = XC+partC ;
    //Πρακτικά υπολογίζονται οι 4 υποπίνακες του tmp, ώστε να κληθούν στις 4 αναδρομικές κλήσεις.
    int *tmpA = tmp ;
    int *tmpB = tmpA+partA;
    int *tmpC = tmpB+partB ;
    int *tmpD = tmpC+partC ;

    //cout<< omp_get_thread_num()<<endl ;

    //Εκτελούνται παράλληλα (σε tasks) οι 4 αναδρομικές κλήσεις, για κάθε τέταρτο του X.
    #pragma omp task firstprivate (XA, partA, tmpA)
    multisort(XA, partA, tmpA);

    #pragma omp task firstprivate (XB, partB, tmpB)
    multisort(XB,partB, tmpB);

    #pragma omp task firstprivate (XC, partC, tmpC)
    multisort(XC, partC, tmpC);

    #pragma omp task firstprivate (XD, partD, tmpD)
    multisort(XD, partD , tmpD);

    //Περιμένουμε να τελειώσουν όλα τα tasks.
    #pragma omp taskwait

    //Εκτελούνται παράλληλα (σε tasks) τα 2 πρώτα merge, για τα δύο μισά του X.
    #pragma omp task
    merge(XA, partA + partB, tmpA);

    #pragma omp task
    merge(XC, partC + partD, tmpC);

    //Περιμένουμε να τελειώσουν όλα τα tasks.
    #pragma omp taskwait
    //Εκτελείτε σειριακά το καθολικό-τελικό merge για όλον τον X.
    merge(X, n, tmp);
}
```

<u>Μια τελευταία σύντομη παρατήρηση:</u>

Όπως είπαμε και στην τάξη, κάπου μπορεί να μην έχουν δηλωθεί κάποιες μεταβλητές ως shared, αλλά δεν μας πειράζει γιατί είναι η default επιλογή και το πρόγραμμα παίζει μια χαρά έτσι.

# Τέλος