1/4/2019

ΗΡΥ 418 -Αρχιτεκτονική Παράλληλων και Κατανεμημένων Υπολογιστών

Άσκηση 1: Χρήση OpenMP και pthreads

Ομάδα LAB41839801

ΓΑΛΑΝΟΣ ΓΕΩΡΓΙΟΣ Α.Μ.: 2014030033
ΜΠΑΪΚΟΥΣΗΣ ΙΩΑΝΝΗΣ Α.Μ.: 2014030133

Σκοπός εργαστηριακής άσκησης

Σκοπός της εργαστηριακής άσκησης είναι η κατανόηση και η χρίση του OpenMP Application Protocol Interface (API) καθώς και ένα υποσύνολο των συναρτήσεων του POSIX threads standard. Ο αλγόριθμος τον οποίο καλούμαστε να υλοποιήσουμε καθώς και να επιταχύνουμε είναι ο Smith-Waterman ο οποίος χρησιμοποιείται για την τοπική ευθυγράμμιση ακολουθιών.

Περιγραφή

Ο αλγόριθμος μας έχει πολυπλοκότητα O(m*n) όπου m,n είναι τα μήκη των ακολουθιών. Αρχικά δημιουργήσαμε 5 κώδικες εκ τον οποίον ο ένας είναι ο σειριακός αλγόριθμος, οι δυο επόμενες είναι υλοποιήσεις fine-grain με χρήση OpenMP & pthreads και τέλος οι άλλες 2 είναι coarse-grain με χρήση OpenMP & pthreads.Το διαφορετικό granularity έχει κατά συνέπεια διαφορετικό computation to communication ratio. Πρέπει να διευκρινίσουμε ότι οι υλοποιείς fine-grain με OpenMP & pthreads περιέχουν ακριβός τα ίδια tasks και αντίστοιχα για τις coarse-grain τα tasks είναι τα ίδια. Παρακάτω παρουσιάζονται τα αποτελέσματα των πειραμάτων στα 10 datasets και με τις 5 υλοποίησης

Σειριακή υλοποίηση:

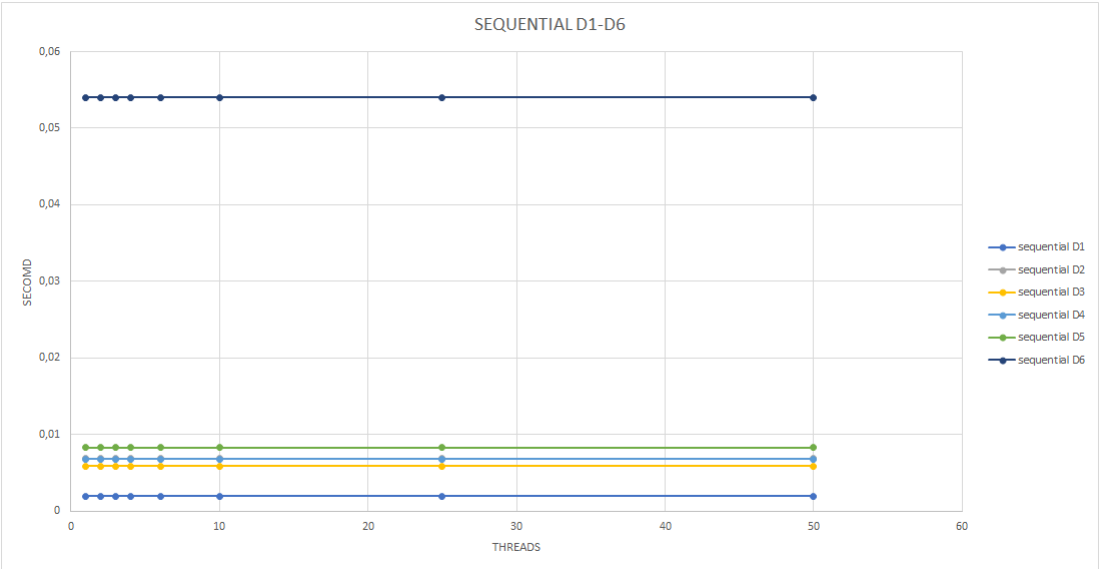
Το γενικό μοτίβο του προγράμματος μας στην σειριακή υλοποίηση (καθώς και στις άλλες υλοποιήσεις με μικρές αλλαγές) είναι ότι διαβάζουμε ένα ζευγάρι ακολουθιών από το αρχείο και δυναμικά δεσμεύουμε την κατάλληλη μνήμη για αυτές. Στην συνέχεια αρχικοποιούμε τον πίνακα ο οποίος θα δεχτεί τα scores τα οποία θα εξάγουμε από την σύγκριση ανάλογα με τα MATCH MISSMATCH και GAP που θα μας δώσει ο χρήστης. Έπειτα υπολογίζουμε και γράφουμε σε αυτόν τα score (εδώ είναι και η συνάρτηση στην οποία εφαρμόσουμε τις fine-grain παραλληλοποίησης μας) και τέλος κάνουμε το traceback ώστε να δούμε το αποτέλεσμα τις ευθυγράμμισης.

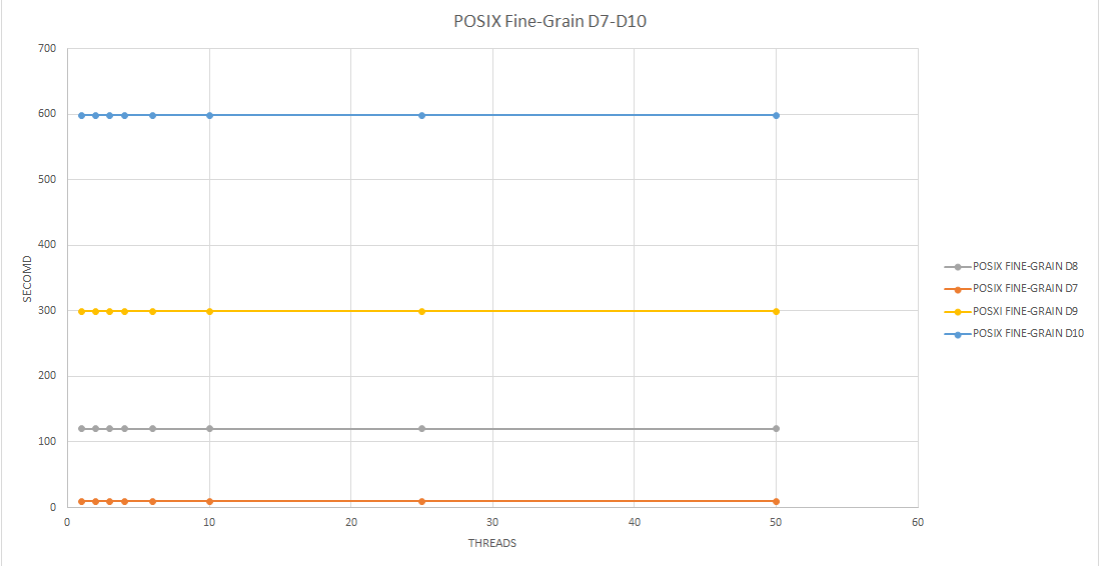
Σε όλο το πρόγραμμα μας και για όλες τις υλοποιήσεις η δέσμευση μνήμης γίνετε δυναμικά.

Παρακάτω φαίνεται η συνάρτηση η οποία υπολογίζει τα scores του πίνακα:

```
void ScoreTable(char seq1[],char seq2 []){
    int seq1len = strlen(seq1);
    int seq2len = strlen(seq2);
    double s =gettime();
    for (int i = 1; i <= seq1len ; i++)
    {
        for (int j = 1; j <= seq2len ; j++)
        {
            int scoreDiag = 0;
            int scoreLeft=0;
            int scoreUp=0;
            int maxScore=0;
            count_cells++;
            if (seq1[i - 1] == seq2[j - 1]){//equal
                scoreDiag = M[i - 1][j - 1] + MATCH;
            }
            else{
                scoreDiag = M[i - 1][j - 1] + MISSMATCH;//missmatch
            }
            scoreLeft = M[i][j - 1] + GAP;
            scoreUp =  M[i - 1][j] + GAP;
            maxScore = MAX(MAX(scoreDiag, scoreLeft), scoreUp);//find max score
            if(maxScore <= 0){
                M[i][j] = 0;
            }
            else{
                M[i][j] = maxScore;//cell max score
            }
        }
    }
                    double st =gettime();
                    cells_calc_time=cells_calc_time+(st-s);
}
```

Τέλος τρέξαμε όλα τα dataset με τον σειριακό μας κώδικα και πήραμε τις παρακάτω μετρήσεις:





Όπως εύκολα παρατηρούμε όσο ποιο μεγάλο είναι το αρχείο τόσο ποιο πολύ καθυστερεί να το διατρέξει όλο.

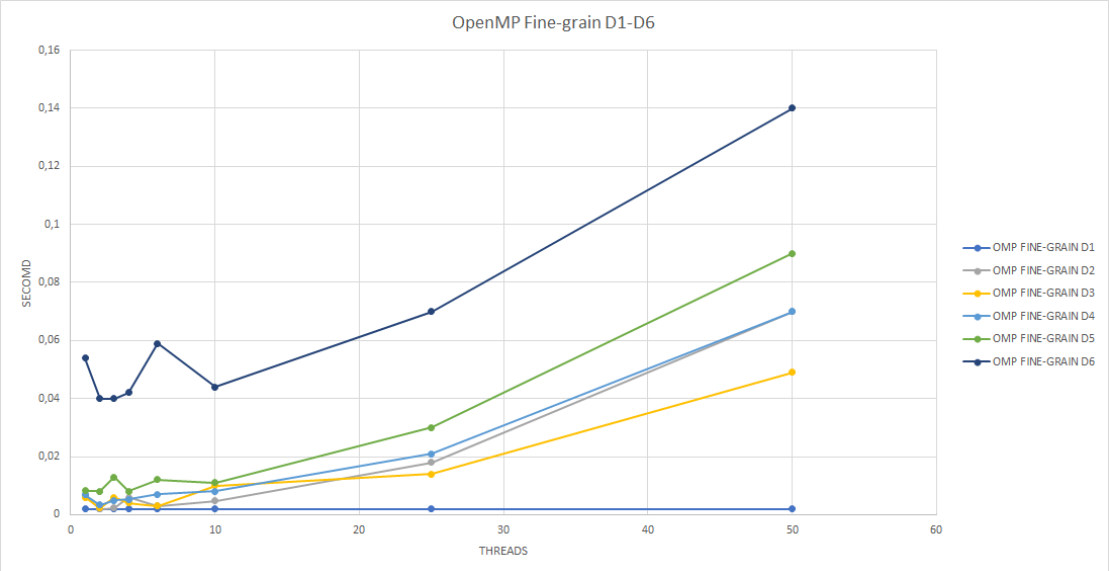
OpenMP fine-grain υλοποίηση:

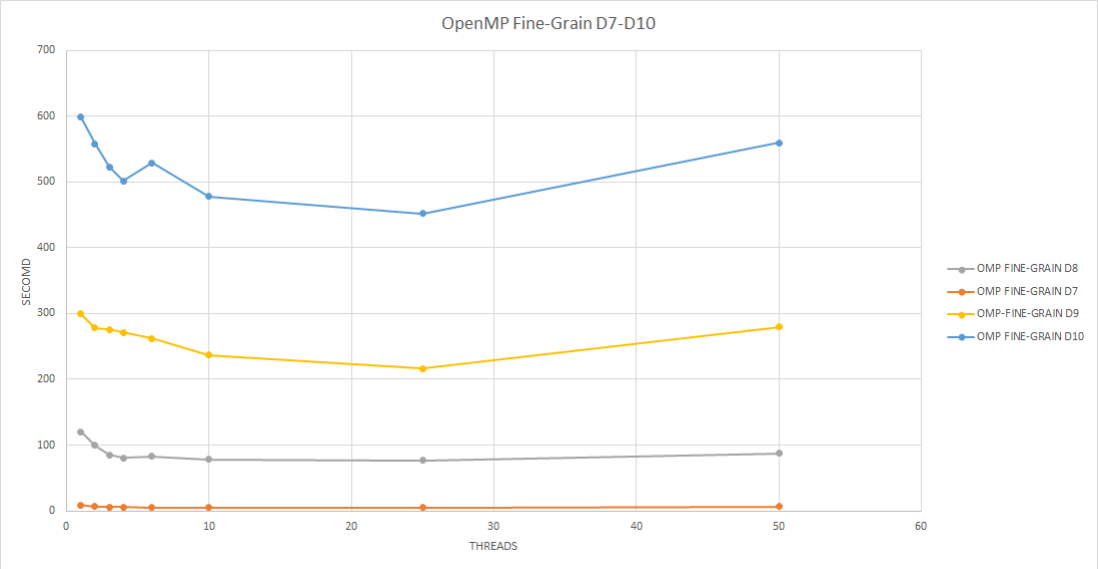
Στον υπόλοιπο κώδικα εκτός τις scoretable συνάρτησης δεν αλλάξαμε τίποτα. Στην συγκεκριμένη συνάρτηση παραλληλοποιήσαμε την διπλή for loop με την οποία υπολογίζετε κελί-κελί ο πίνακας. Το κύριο πρόβλημα ήταν ότι χρειαζόμασταν πάντα το πάνω κελί από αυτό που υπολογίζαμε κάθε στιγμή καθώς και το διαγώνιο του και το αριστερά του. Αν κάναμε μια κανονική παραλληλοποίηση με το omp parallel for ουσιαστικά δεν θα είχαμε αποτέλεσμα λόγο κακού load balance (με την προϋπόθεση να γράφονται σωστά τα scores στον πίνακα). Ο τρόπος που κάναμε την παραλληλοποίηση είναι ουσιαστικά ότι σπάμε την στήλη σε ίσα κομμάτια ανάλογα με το πόσα threads έχουμε. Υπολογίζουμε αρχικά το $1^{ο}$ κομμάτι τις $1^{ης}$ στήλης με το $1^{ο}$ thread. Στην συνέχεια αφού το $1^{ο}$ κομμάτι είναι έτυμο αυτό το thread (με ID=0) θα πάει να φτιάξει το $1^{ο}$ κομμάτι τις $2^{ης}$ στήλης. Ταυτόχρονα ξεκινάμε το $2^{ο}$ thread να συνεχίσει για άλλο ένα κομμάτι την $1^{η}$ στήλη (από εκεί που σταμάτησε το $1^{ο}$). Αφού φτιαχτούν και αυτά τα κομμάτια τότε συνεχίζουμε , το $1^{ο}$ thread θα φτιάξει το $1^{ο}$ κομμάτι τις $3^{ης}$ στήλης, το $2^{ο}$ θα φτιάξει των το $2^{ο}$ κομμάτι τις $2^{ης}$ στήλης και θα ξεκινήσει και το $3^{ο}$ thread να φτιάχνει το $3^{ο}$ κομμάτι τις $1^{ης}$ στήλης και συνεχίζετε έτσι μέχρι να τελειώσουν όλες οι στήλες. Αν τα threads δεν φτάνουν για να γεμίσουν με την μια τον πίνακα τότε μόλις το $1^{ο}$ thread τελειώσει το κομμάτι τις τελευταίας στήλης τότε πηγαίνει να δημιουργήσει το κομμάτι το οποίο είναι κάτω από το κομμάτι που δημιούργησε το τελευταίο thread. Παρομοίως και τα υπόλοιπα thread κάνουν αντίστοιχη δουλειά με αποτέλεσμα να έχουμε μια αρκετά καλή κατανομή εργασιών τα αποτελέσματα τις οποίας φαίνονται και στην απόδοση καθώς βλέπουμε μεγάλες διάφορες στους χρόνους εκτέλεσης και ειδικότερα των μεγάλων ακολουθιών.

Παρακάτω φαίνεται ο κώδικας για την υλοποίηση της παραλληλοποίησης που αναλύθηκε παραπάνω:

```
#pragma omp parallel shared(seq1,seq2,M)  private(i,j,wave,tStart,tEnd,ID,cells,scoreDiag,scoreLeft,scoreUp,maxScore) num_threads(threads)
    {
        nothreads=omp_get_num_threads();
        ID=omp_get_thread_num();
        tStart= (((seq2len/nothreads)*ID)+1);  // start thread by ID
        tEnd= tStart+ceil(seq2len/nothreads); // end by thread
    for (   wave = 1; wave <= seqllen+nothreads-1 ; wave++)
    {
        i=wave-ID;// delay for threads
       if(i>=1 && i<=seqllen)
       {
                    for (  j = tStart; j <= tEnd    ; j++)
                   {
                        scoreDiag = 0;
                        scoreLeft=0;
                        scoreUp=0;
                        maxScore=0;
                      if (seq1[i - 1] == seq2[j - 1]){
                          scoreDiag = M[i - 1][j - 1] + MATCH;
                      }
                      else{
                          scoreDiag = M[i - 1][j - 1] + MISSMATCH;
                      }
                      scoreLeft = M[i][j - 1] + GAP;
                      scoreUp =  M[i - 1][j] + GAP;
                      maxScore = MAX(MAX(scoreDiag, scoreLeft), scoreUp);
                      if(maxScore <= 0){
                          M[i][j] = 0;
                      }
                      else{
                          M[i][j] = maxScore;
                      }
                    }
                   }

    #pragma omp barrier
        }
}
```

Τέλος τρέξαμε όλα τα dataset με τον παράλληλο μας κώδικα και πήραμε τις παρακάτω μετρήσεις





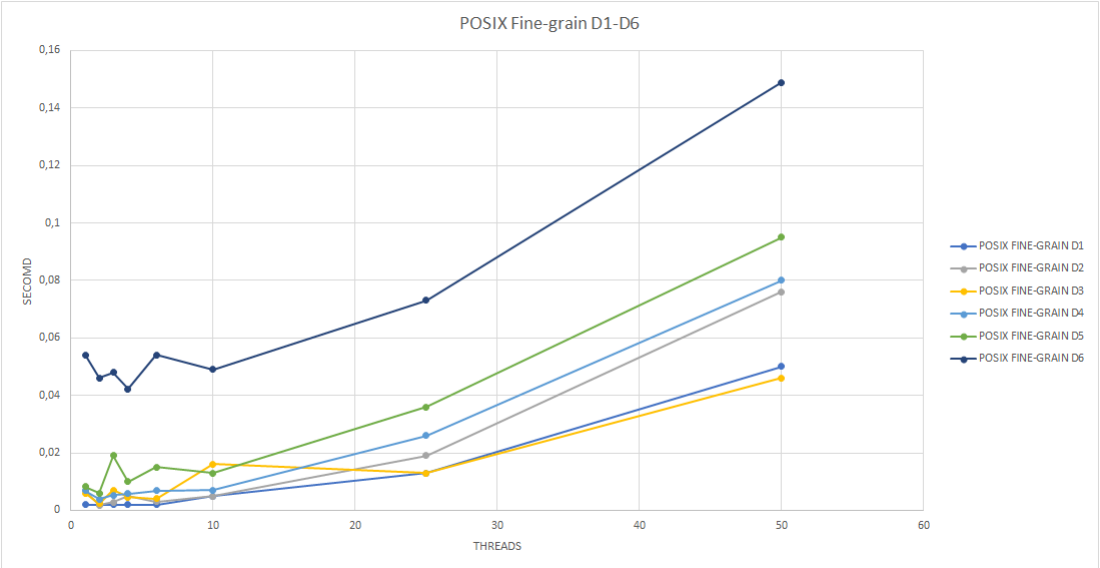
Παρατηρούμε ότι η βελτίωση της ταχύτητας γίνετε για περιορισμένο αριθμό από threads.Έτσι καταλαβαίνουμε ότι κυρίως στα μικρά αρχεία είναι ανώφελο να έχουμε πολλά thread διότι σπαταλάμε περισσότερο χρόνο να τα δημιουργήσουμε πάρα να εκτελέσουν. Επίσης υπάρχει καθυστέρηση στην επικοινωνία των πυρήνων ώστε να συγχρονίσουν τα threads η οποία επάγετε σε καθυστέρηση του συνολικού χρόνου εκτελέσεις(communication to computation). Τέλος έχουμε μεγαλύτερη βελτίωση στα αρχεία με μεγάλες ακολουθίες(όχι πολλές) όσο αυξάνουμε τα thread διότι δεν υπάρχουν idle threads (και αυτό βεβαία έως ένα ορισμένο σημείο).

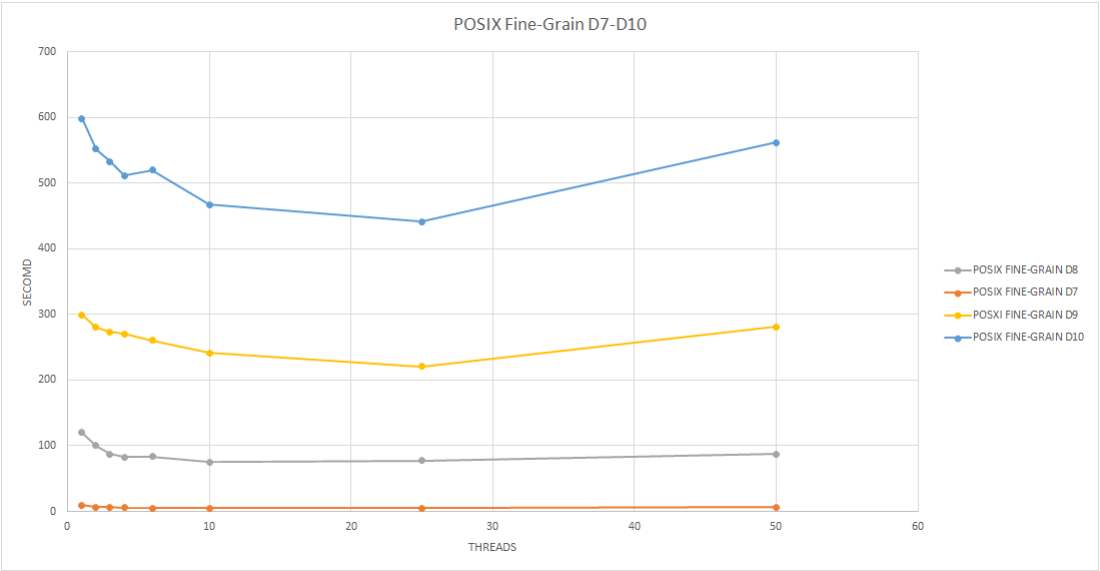
POSIX fine-grain υλοποίηση:

Η λογική του load balance και στα pthreads είναι ακριβός η ίδια με το OpenMP. Για να την υλοποιήσουμε αρχικά δημιουργήσαμε μια struct για να περάσουμε ορίσματα στο task του κάθε thread το οποίο είναι η διπλή for loop η οποία γεμίζει τον πίνακα. Αυτή την κάναμε ξεχωριστή συνάρτηση ώστε να την δίνουμε σαν task μέσω της create thread. Τα threads είναι μέσα σε έναν πίνακα pthread_t και αναγνωρίζονται με το ID το οποίο το έχουμε δώσει εμείς μέσα στην struct μέσω της setup_thread_data.

```
    int nothreads=threads;


        thread_data_t *td;//srtuct with data
        td=setup_thread_data(nothreads, seq1len, seq2len);//setup data
        pthread_t thread[nothreads]; //threads
        pthread_barrier_init(&bar,NULL,nothreads);
        for(int i=0;i<nothreads;i++){
                pthread_create(&thread[i], NULL, p_SmithWaterman, (void*)&td[i]);// create thread and exec task
        }
       for(int j=0;j<nothreads;j++){
                pthread_join(thread[j],NULL); //join for exit and continue
        }
        pthread_barrier_destroy(&bar);

                    double st =gettime();
                    count_cells=count_cells+(seq1len*seq2len);
                    cells_calc_time=cells_calc_time+(st-s);

                    return;
}
```

Τέλος τρέξαμε όλα τα dataset με τον παράλληλο μας κώδικα και πήραμε τις παρακάτω μετρήσεις:





Παρατηρούμε οι μετρήσεις είναι παρόμοιες διότι το granularity είναι ακριβός το ίδιο στις 2 υλοποιήσεις. Οι μικρές διαφορές οφείλονται στο λειτουργικό. Τα συμπεράσματα είναι τα ίδια όπως και στην OpenMP υλοποίηση.

OpenMP coarse-grain υλοποίηση:

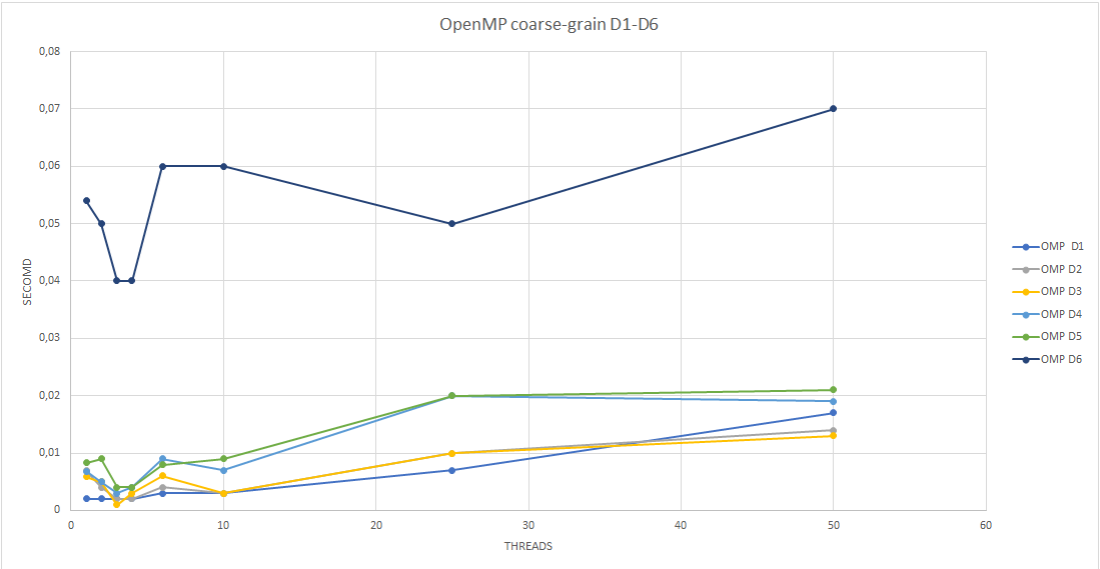
Για την coarse-grain υλοποίηση μας με OpenMP κάναμε ορισμένες αλλαγές στον κώδικα μας. Προηγουμένως διαβάζαμε ζευγάρι-ζευγάρι τις ακολουθίες από το αρχείο. Τώρα διαβάζουμε όλες τις ακολουθίες από το αρχείο και το κάθε thread θα πάρει μια ακολουθία ολόκληρη να εκτελέσει και όχι κομμάτι ακολουθίας όπως ήταν στις fine-grain προηγουμένως. Ουσιαστικά το task εδώ είναι η πλήρης εκτέλεση του αλγορίθμου για ένα ζευγάρι ακολουθιών. Είναι φανερά πολύ μεγαλύτερο το task στην coarse-grain υλοποίηση μας από ότι στην fine-grain η οποία έπαιρνε ως task μια διπλή for loop μόνο. Το πρόβλημα εδώ είναι ότι στο μεγάλο αυτό task υπάρχει και εγράφη στο αρχείο η οποία δεν μπορεί να γίνει παράλληλα διότι υπάρχουν περισσότερα από 1 match για ορισμένες ακολουθίες τα οποία πρέπει να γραφούν με την σειρά. Αυτό κάνει επιτακτική την ανάγκη για να την χρίση του critical σε ένα section κώδικα στο οποίο γράφουμε στο αρχείο μας για την αποφυγή race condition. Αυτό το κάνει η traceback συνάρτηση μας την οποία βάζουμε σε critical. Ο παραλληλισμός της for η οποία τρέχει για όλες τις ακολουθίες οι οποίες είναι σηκωμένες στην μνήμη γίνετε με χρίση του omp parallel for το οποίο κάνει αυτό πολύ καλό load balance στα threads.

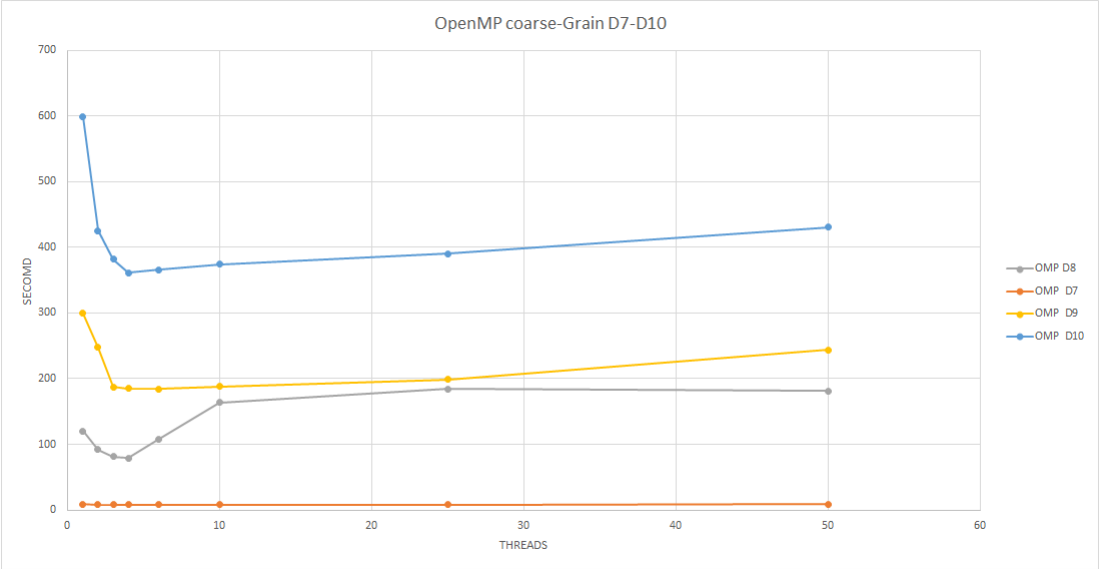
Παρακάτω φαίνεται η παράλληλη υλοποίηση μας με coarse-grain granularity με χρίση OpenMP:

```
#pragma omp parallel for private(i,m,s) num_threads(threads)
        for( i = 0; i <k ; i++){
            m=initialize(Q[i],D[i]);
             s= ScoreTable(Q[i],D[i],m);
            #pragma omp critical
            {
             Traceback(Q[i],D[i],s);
            }
        }
```

Q[i] είναι ο πίνακας που περιέχει τις ακολουθίες Q, αντίστοιχα ο D[] τις D. Υπάρχει πλήρης αντιστοιχία μεταξύ την ζευγαριών στις θέσεις του πίνακα. Το κ είναι ο αριθμός των ζευγαριών. Τα m,s είναι δείκτες στον πίνακα ο οποίος περιέχει τα scores. Όπως βλέπουμε είναι private άρα κάθε thread έχει το δικό του instance για αποφυγή race condition.

Τέλος τρέξαμε όλα τα dataset με τον παράλληλο μας κώδικα και πήραμε τις παρακάτω μετρήσεις:





Στην coarse-grain υλοποίηση τα πράγματα είναι λίγο διαφορετικά. Εδώ αρχεία με μικρό αριθμό από ακολουθίες δεν παρουσιάζουν ουσιαστική διαφορά στους χρόνους από τον σειριακό. Αυτό οφείλετε στο ότι ως task εδώ είναι όλος ο αλγόριθμος, οπότε 1 thread θα έχει μια ακολουθία. Άρα όπως φαίνεται και στο πάνω διάγραμμά καθώς και στην D7 όσο αυξάνουν τα threads δεν βλέπουμε βελτίωση στον χρόνο. Αντιθέτως στα μικρά αρχεία όπου ο χρόνος εκτέλεσης τους είναι μικρός παρατηρούμε αύξηση στον χρόνο όσο αυξάνουν τα threads διότι κάνουν περισσότερο χρόνο να δημιουργηθούν πάρα να εκτελέσει ο αλγόριθμος (για μικρές και λίγες ακολουθίες). Επίσης πολλά threads μένουν άεργα.

Τέλος η βελτίωση φαίνεται ποιο πολύ στα αρχεία με μεγάλο αριθμό από ακολουθίες στα οποία γίνετε και ο παραλληλισμός σε επίπεδο coarse-grain. Εδώ κάθε thread παίρνει από μια ακολουθία και την εκτελεί. Οπότε με πολλά threads θα εκτελούνται πολλές ακολουθίες ταυτόχρονα. Το overhead στην coarse-grain είναι σχετικά μικρό σε σχέση με την fine-grain υλοποίηση.

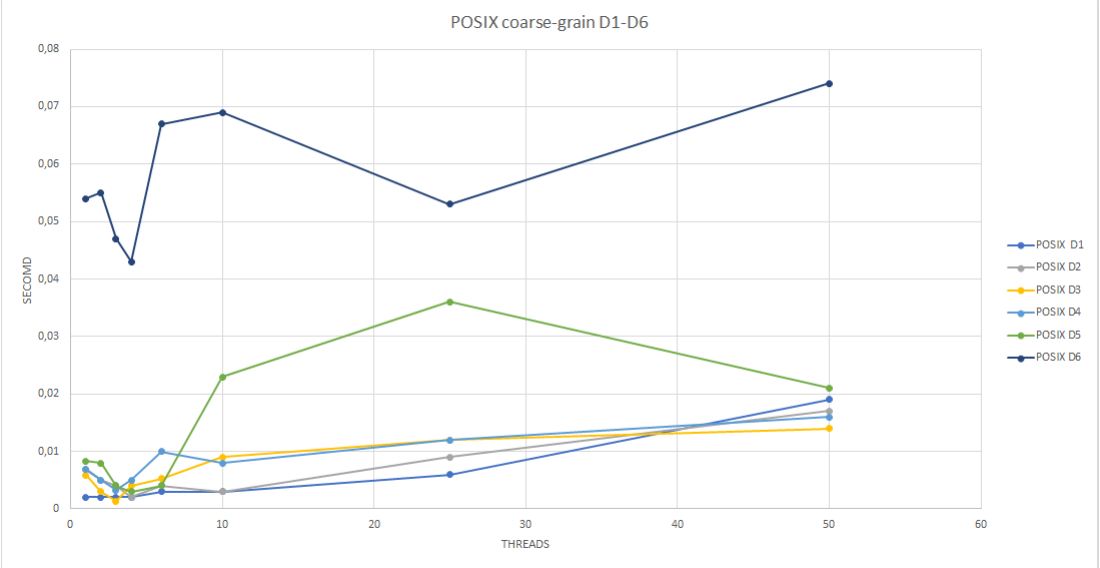
POSIX coarse-grain υλοποίηση:

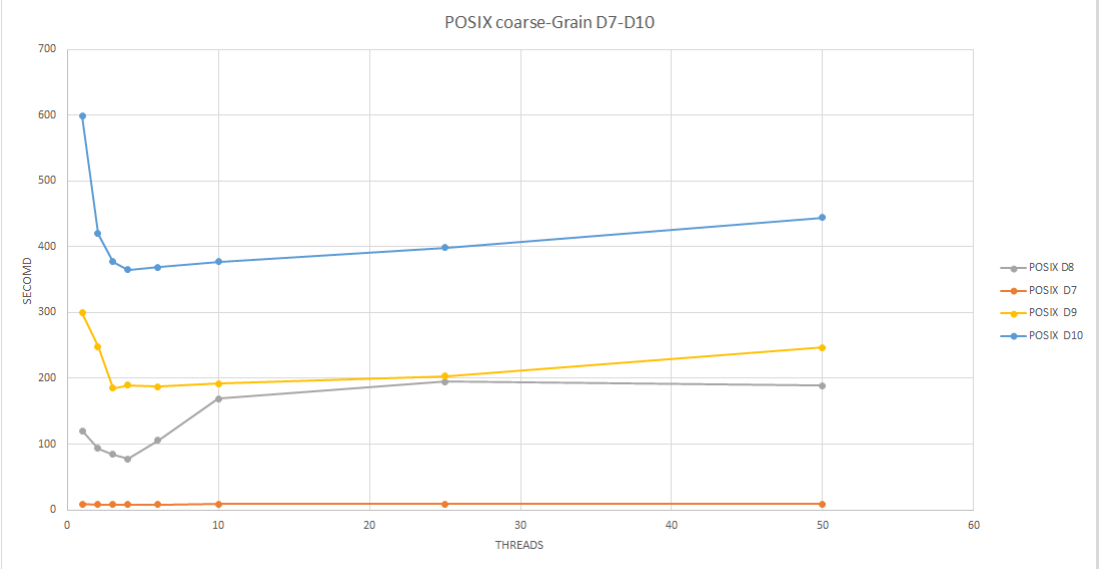
Ακριβός με την ίδια λογική όπως προηγουμένως ,η coarse-grain υλοποίηση με pthreads αφορά την εκτέλεση μεγάλου task και αντίστοιχα την εκτέλεση του αλγορίθμου για ένα ζευγάρι ακολουθιών. Η αλλαγή του κώδικα είναι παρόμοια με του OpenMP. Ποιο συγκεκριμένα δημιουργούμε τα threads όπως και στην fine-grain υλοποίηση μας και ως task έχουμε την εκτέλεση του αλγορίθμου ο οποίος εκτελείτε στην συνάρτηση algor. Η διαφορά από το OpenMP είναι ότι των δεν υπάρχει ''έτυμη'' η κατανομή του load και έτσι καλούμαστε να την κάνουμε εμείς. Η λογική που ακολουθήσαμε είναι η εξής: κάθε thread ξεκινάει από το id του και έχει βήμα τον αριθμό των threads, δηλαδή αν έχουμε 3 threads το thread με id=0 θα πάρει την 1η ακολουθία μετά την 4η , την 7η κτλ. Αντίστοιχα το 2ο thread θα πάρει την 2η, την 5η , την 8η κτλ. Από τα αποτελέσματα των μετρήσεων φαίνεται να είναι μια αρκετά καλή κατανομή φόρτου στα threads μιας και οι χρόνοι είναι πολύ κοντά στην προηγουμένη coarse-grain υλοποίηση με OpenMP. Τέλος όπως και στην OpenMP υλοποίηση πρέπει να αποφύγουμε το race condition της εγγραφής και έτσι ''κλειδώνουμε'' την πρόσβαση στην συνάρτηση traceback στην οποία γίνετε η εγγραφή με την χρίση mutex.

```
    pthread_mutex_init(&count_mutex, NULL);
    thread_data_t *td;
    td=setup_thread_data(nothreads, k);
    pthread_t thread[nothreads];
    pthread_barrier_init(&bar,NULL,nothreads);

    for(int i=0;i<nothreads;i++){
        pthread_create(&thread[i], NULL, algor, (void*)&td[i]);
    }

   for(int j=0;j<nothreads;j++){
            pthread_join(thread[j],NULL);
    }
    pthread_barrier_destroy(&bar);

pthread_mutex_destroy(&count_mutex);
```

```
void*algor(void * ptr_to_tdata){
    int**m;
    int**s;
thread_data_t * td = (thread_data_t*)ptr_to_tdata;

    for(int i = td->thread_id; i <td->k ; i+=td->numThreads){
                m=initialize(Q[i],D[i]);

                 s= ScoreTable(Q[i],D[i],m);

               pthread_mutex_lock(&count_mutex);
                 Traceback(Q[i],D[i],s);
               pthread_mutex_unlock(&count_mutex) ;

            }

pthread_exit((void*)ptr_to_tdata);

}
```

Τέλος τρέξαμε όλα τα dataset με τον παράλληλο μας κώδικα και πήραμε τις παρακάτω μετρήσεις:





Ήταν αναμενόμενο να έχουμε παρόμοια αποτελέσματα μιας και το επίπεδο του granularity είναι το ίδιο. Υπάρχουν βεβαίως διαφορές καθώς αλλάζει και η υλοποίηση όπως αναφέρθηκε ποιο πάνω αλλά και προφανώς το σύστημα το οποίο μπορεί να προκαλέσει μικροκαθυστερήσεις στην εκτέλεση του και ειδικότερα στους πολύ μικρούς χρόνους που φαίνονται και ποιο έντονα. Όλα τα συμπεράσματα είναι ίδια με του OpenMP coarse-grain. Τέλος παρατηρούμε ότι η υλοποίηση που κάναμε στα pthreads για το load balance είναι πολύ καλή αφού παρατηρούνται μικροδιαφορές από τον αυτοματοποιημένο παραλληλισμό του OpenMP.

Συμπεράσματα:

Κατανοήσαμε πλήρως την έννοια της παραλληλοποίησης και ήρθαμε σε μια καλή επαφή με τοOpenMP και τα pthreads. Επίσης καταλάβαμε πως το granularity επηρεάζει την απόδοση του αλγορίθμου μας. Επιπροσθέτως μέσω των παρατηρήσεων των διαγραμμάτων έγινε ποιο εμφανές το σημείο το ποιο ανάλογα με τον αριθμό των threads έχουμε τους βέλτιστους χρόνους. Ακόμα ενστερνιστήκαμε πλήρως την έννοια του computation to communication.

Παρατηρήσεις στην εκτέλεση και λειτουργία του προγράμματος:

Μετά το πέρας της εκτέλεσης του αλγορίθμου σε κάθε πρόγραμμα εκτυπώνονται τα στοιχεία που ζητούνται από την εκφώνηση. Ο συνολικός αριθμός κελιών που πήραν τιμή αφορά όλους του πίνακες και όχι μόνο τον scoretable (πίνακες για align, ακολουθίες κτλ). Ο συνολικός αριθμός από traceback steps μετράει κάθε βήμα ευθυγράμμισης ώστε να βρεθεί η μια ακολουθία. Ο χρόνος υπολογισμού κελιών επίσης αφορά όλα τα κελιά των πινάκων προγράμματος. Οπότε και τα CUPS λειτουργούν με βάση αυτούς τους υπολογισμούς.

Script:

Το runscript το οποίο δημιουργήσαμε δέχεται ως εισόδους :

-name “ID του report” : Όπως αναφέρετε στην εκφώνηση το ID είναι το ένα διακριτικό το οποίο εμφανίζετε στο report εξόδου.

-input ‘’το path στο οποίο βρίσκονται τα αρχεία ακολουθιών’’ : Το input είναι η παράμετρος στην οποία βάζουμε το που βρίσκονται τα αρχεία ,χωρίς το όνομα του αρχείου για να μπορεί να πάρει τα αρχεία D1 ,D2 ,D3 ,D4 ,D5 και D9 αυτόματα. Τα αρχεία εξόδου θα δημιουργηθούν στον ίδιο φάκελο με αυτόν του input.

-match ”int”: ένας ακέραιας ο οποίος είναι η τιμή του match.

-missmatch ”int”: ένας ακέραιας ο οποίος είναι η τιμή του missmatch.

-gap ”int”: ένας ακέραιας ο οποίος είναι η τιμή του gap.

-threads ”int”: ένας ακέραιας ο οποίος δηλώνει με ποσά threads θα εκτελέσει το πρόγραμμα μας. Προφανώς στην σειριακή υλοποίηση δεν χρησιμοποιούνται τα threads και μπορούν να παραληφθούν.

Όλα τα ορίσματα μπορούν να μπουν με τυχαία σειρά όπως ζητείτε και σύμφωνα με την τελευταία ανακοίνωση το πρόγραμμα τρέχει αυτόματα στην ακολουθίες που αναφέρθηκαν παραπάνω για όλα τα προγράμματα σειριακά καθώς και με fine και coarse granularity με είσοδο των παραπάνω παραμέτρων. Στην περίπτωση λιγότερων ορισμάτων το πρόγραμμα δεν λειτουργεί σωστά. Τέλος να παρατηρηθεί πως για να μην υπάρξει ασάφεια σχετικά με το που θα πάνε τα αρχεία εξόδου και ποσά θα είναι αυτά στον αυτοματοποιημένο κώδικα (και στο να μην δημιουργηθούν πάρα πολλά αρχεία εξόδου) τα αρχεία εξόδου απογράφονται.

Παράδειγμα εκτέλεσης :

“bash runscript.sh -name “A” -input “/home/john/” -match 3 -missmatch -1 -gap -1 -threads 4”

Συνεργασία:

Σας αναφέρουμε ότι συνεργαστήκαμε με την ομάδα των συναδέλφων Μερταράκη Ανδρέα και Μάνο Ρομπογιαννάκη και ποιο συγκεκριμένα στις πηγές ορισμένων κωδίκων από το internet (παραθέτονται στην βιβλιογραφία) καθώς και coarse υλοποιήσεις των παράλληλων προγραμμάτων.

Βιβλιογραφία:

Align: http://www.quretec.com/u/vilo/edu/2002-03/Tekstialgoritmid_I/Loengud/Loeng3_Edit_Distance/bcorum_copy/seq_align4.htm?fbclid=IwAR0FvQfxQA8con2GftEaOyrFa_Fji4DoTlq6w37sNNzD_-ylSzOwgJHm7AU

Score table: https://github.com/PSNAppz/Smith-Waterman/blob/master/waterman.c?fbclid=IwAR3cjP8jzYSF8do3b-exk3mVDZq-Kq6TqtXBvhMDd_d4jT46kafWLhFZMWk

Omp & pthreads fine: https://github.com/Leezekun/Smith-Waterman-1?fbclid=IwAR1iOMwIY6QAtY_aTI3u7c52HJ1XtMMlzBbLZt2HgG22e9VNc1wkfKJToMw