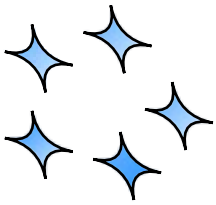


Αρχιτεκτονική Παράλληλων και Κατανεμημένων Υπολογιστών

Ασκηση 1

Σπυριδάκης Χρήστος

1 Εισαγωγή

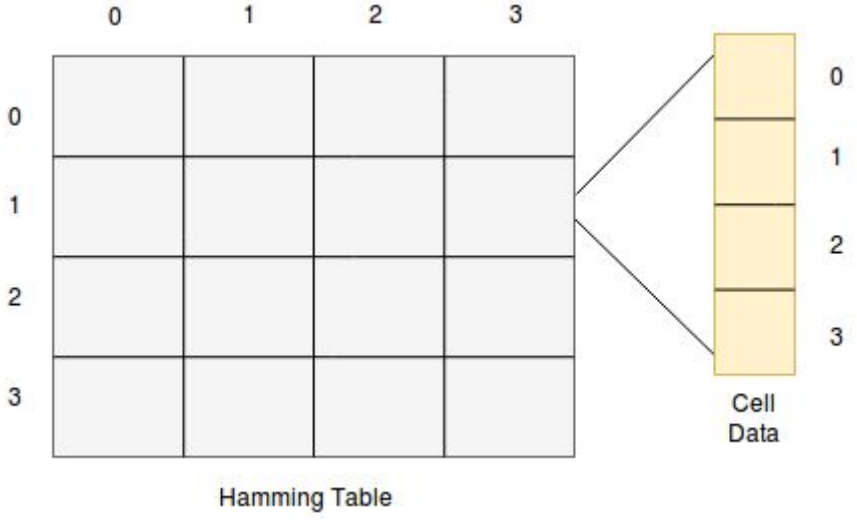
Η υλοποίηση της άσκησης χωρίζεται σε 4 βασικά μέρη. Στα αρχεία τα οποία συσχετίζονται με την σειριακή υλοποίηση (serialHamming.c και serialHamming.h), στα αρχεία τα οποία συσχετίζονται με τις παράλληλες υλοποιήσεις χρησιμοποιώντας OpenMp (openMpHamming.c και openMpHamming), σε αυτά που πραγματοποιούνται οι παράλληλες υλοποιήσεις με Pthreads (pthreadsHamming.c και pthreadsHamming.h) και τέλος στα συνδετικά αρχεία και αρχεία πληροφοριών (main.c, Makefile, κλπ.).
Για να περιγραφτεί όσο πιο αναλυτικά γίνεται η υλοποίηση της άσκησης, η αναφορά έχει χωριστεί σε 3 ενότητες:

1. **Περιγραφή**
2. **Εκτέλεση**
3. **Συμπεράσματα**

---

2 Περιγραφή

Σε αυτήν την ενότητα θα περιγραφεί ο τρόπος με τον οποίο υλοποιήθηκε κάθε υποσύνολο της άσκησης. Στην παρακάτω εικόνα παρουσιάζεται ο πίνακας Hamming για ένα απλοποιημένο παράδειγμα συνόλου A με αριθμό συμβολοσειρών (αριθμός m) ίσον με 4, ενός σύνολου B με αριθμό συμβολοσειρών (αριθμός n) ίσος με 4. Ενώ το μέγεθος συμβολοσειράς (αριθμός l) έχει και αυτό μήκος 4.



Εικόνα 2.1 : Hamming table για σύνολα A και B με μέγεθος A ίσον του B ίσον με 4 και μέγεθος συμβολοσειράς ίση με 4

ΣΗΜΕΙΩΣΗ: Αριστερά εμφανίζεται ο πίνακας στον οποίο θα αποθηκευτούν οι αποστάσεις hamming για κάθε συνδυασμό συμβολοσειρών μεταξύ των συνόλων A και B. Ενώ δεξιά εμφανίζεται για μία μεμονωμένη περίπτωση η συμβολοσειρά που θα ελεγχθεί. **ΣΗΜΑΝΤΙΚΟ** υπάρχει μόνο ένας πίνακας δεξιά (και όχι 2, ένας για κάθε συμβολοσειρά ενός συνόλου), καθώς επειδή ο έλεγχος είναι σε πλήρη αντιστοιχία (π.χ. αν φανταστούμε τα σύνολα ως δύο 2d arrays όπου είναι το A [n] [l] και το B [m] [l]. Τότε αν θέλουμε να βρούμε την απόσταση hamming στις λέξεις A [3] και B [1] θα ελέγξουμε τις θέσεις της συμβολοσειράς μεταξύ των δύο λέξεων με την ίδια σειρά) οπότε παραβλέπετε ο δεύτερος πίνακας για την απλοποίηση που έγινε.

2.1 **Serial** (Σειριακός Υπολογισμός)

Αφού είχαν δημιουργηθεί και αρχικοποιηθεί τα σύνολα A και B με τυχαίους χαρακτήρες η ευκολότερη αλλά και πιο χρονοβόρα υλοποίηση που έγινε ήταν να βρεθούν σειριακά οι αποστάσεις hamming, αυτό υλοποιήθηκε με τον εξής τρόπο:

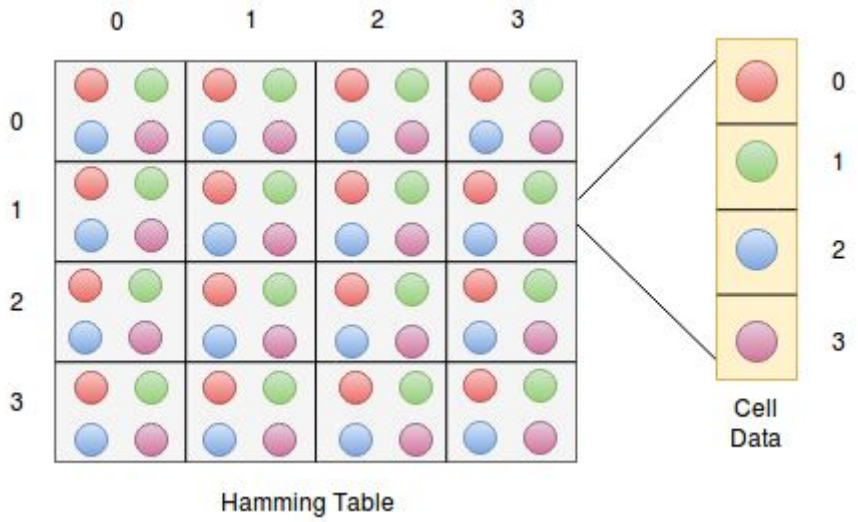
serialHamming.c

```
...
for (i = 0; i < src->Alen; i++) {
        for (j = 0; j < src->Blen; j++) {
                for (k = 0; k < src->Strlen; k++) {
                        if (src->A[i][k] != src->B[j][k]) {
                                serialhammingValues[i][j]++;
                                (*serialSum)++;
                        }
                }
        }
}
...
```

Στον παραπάνω κώδικα αυτό που γίνεται είναι να ελέγχουμε μία μία τις θέσεις των συμβολοσειρών του κάθε συνόλου και να αυξάνουμε δύο μετρητές όταν δούμε ότι διαφέρουν οι χαρακτήρες. Ο ένας μετρητής συσχετίζεται με τον πίνακα hamming, ενώ ο άλλος για το ολικό sum των αποστάσεων hamming.
Αφού ολοκληρωθεί η σειριακή υλοποίηση η συνάρτηση “επιστρέφει” το **serialSum** το οποίο χρησιμοποιούμε ως validate variable στις παρακάτω υλοποιήσεις για να σιγουρευτούμε ότι δεν υπήρξε κάποιο Data Race και οι υπολογισμοί μας ήταν σωστοί.

2.2 Task A (Παραλληλοποίηση σε μέρος των String)

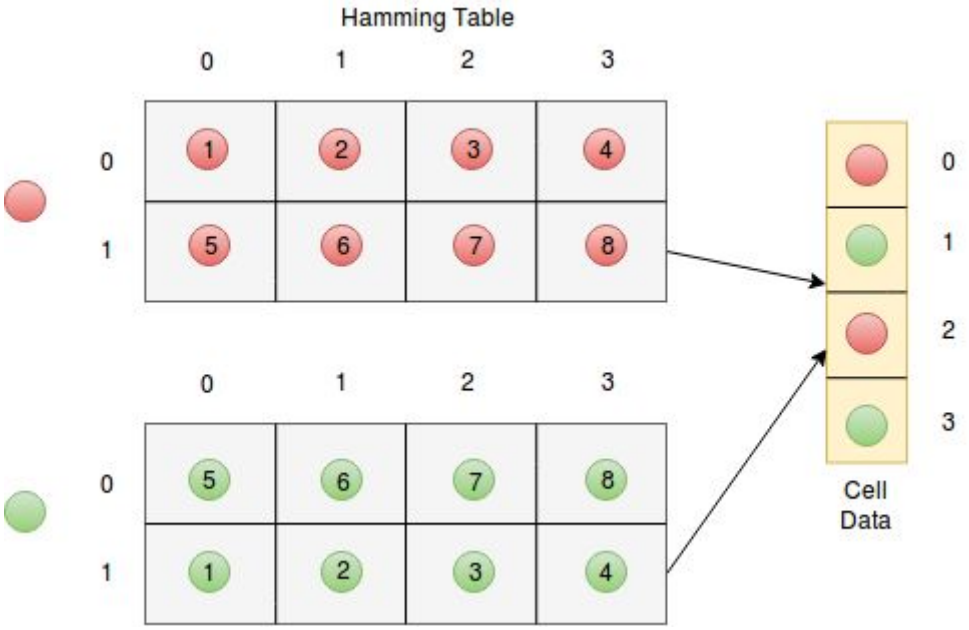
Αφού ολοκληρώθηκε η σειριακή υλοποίηση, υπολογίσαμε τις αποστάσεις hamming με παραλληλοποίηση σε μέρος των συμβολοσειρών. Η γενική ιδέα που υλοποιήθηκε περιγράφεται στην παρακάτω εικόνα:



Εικόνα 2.2: Παραλληλοποίηση σε μέρος των συμβολοσειρών

Έστω ότι θέλουμε να χρησιμοποιήσουμε 4 Thread για τον υπολογισμό, στον οποίο θέλουμε να βρούμε τις αποστάσεις hamming για το παράδειγμα που παρουσιάστηκε παραπάνω. Ο κάθε κύκλος αναπαριστά και ένα Thread και ανάλογα με το χρώμα του κύκλου μπορούμε να ξεχωρίσουμε το ένα Thread από ένα άλλο. Στην παραπάνω εικόνα παρουσιάζεται ποιες θέσεις θα προσπελαστούν από ποιο Thread ώστε να υλοποιηθεί η παραλληλοποίηση που ζητήθηκε. Συγκεκριμένα κάθε Thread θα προσπελάσει κάθε θέση του πίνακα όμως μόνο ένα Thread θα ελέγξει την απόσταση Hamming μεταξύ μίας θέσης της λέξης του συνόλου A και B.

Αυτή η υλοποίηση είναι η πιο ανεκτική σε περιπτώσεις Data Races λόγου το γεγονότος ότι πολλαπλά Thread χρειάζεται να γράψουν στην ίδια θέση του HammingTable. Η απλή λύση είναι ανεξαρτήτου υλοποίησης να κάναμε κάποια μορφή Data Synchronization (π.χ. #pragma omp critical για OpenMp, mutex για Pthreads, κλπ.) πράγμα το οποίο γίνεται αλλά από μόνου του δεν αρκεί. Ο λόγος είναι ότι είναι τόσο συχνές οι περιπτώσεις στις οποίες ένα Thread περιμένει ένα άλλο, που για μεγαλύτερες τιμές των m,n,l η υλοποίηση είναι αρκετά πιο αργή και από την σειριακή. Για να ελαχιστοποιηθεί η πιθανότητα σε κοντινές στιγμές να χρειαστεί να γράψουν διαφορετικά thread στο ίδιο Cell του Hamming Table, έχει ακολουθηθεί και μία άλλη τεχνική. Ανάλογα με το Thread_id τα Thread παρόλο που θα προσπελάσουν όλο τον πίνακα Hamming αυτό θα γίνει με τελείως διαφορετικό τρόπο για το καθένα, με αποτέλεσμα να είναι αρκετά μικρότερες οι πιθανότητες πολλαπλά Thread να χρειαστεί να γράψουν ταυτόχρονα στο ίδιο Cell. Στην παρακάτω εικόνα παρουσιάζεται ένα παράδειγμα αυτής της τεχνικής.



Εικόνα 2.3: Σειρά προσπέλασης δύο Thread στο ίδιο Hamming Table ανάλογα με το Thread ID

Ο κώδικας ο οποίος δημιουργήθηκε για το OpenMp έχοντας υπόψιν τα παραπάνω είναι ο εξής:

openMpHamming.c

```
…
      for(t=0;t<NUM_THREADS;t++){
          int start=(t+THREAD_ID)%NUM_THREADS;
          for(i=start;i<src->Alen;i=i+NUM_THREADS){
              for (j = 0; j < src->Blen; j++) {
                  #pragma omp for nowait
                  for (k = 0; k < src->Strlen; k++) {
                      if (src->A[i][k] != src->B[j][k]) {
                          psum++;

                          #pragma omp atomic
                          hammingValues[i][j]++;
                      }
                  }
              }
          }
      }

      #pragma omp atomic
      sum += psum;
...
```

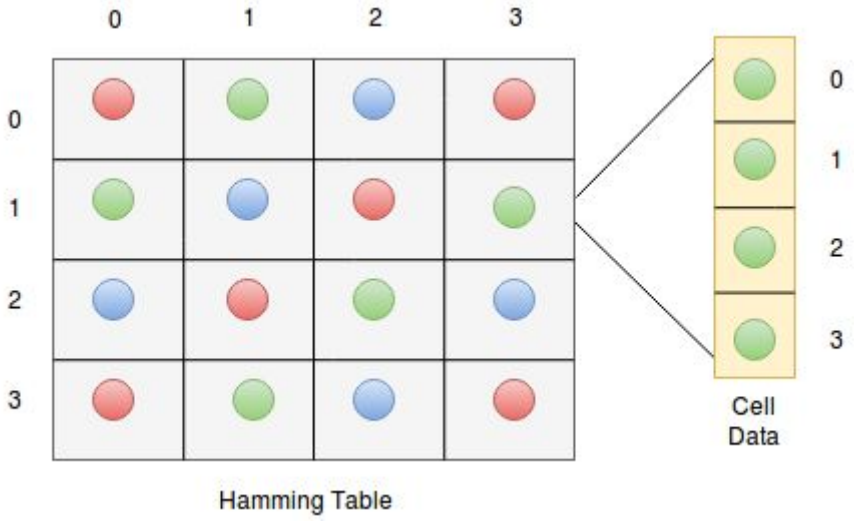
Ενώ για την υλοποίηση με Pthreads είναι ο εξής:

pthreadsHamming.c

```
…
  for (t = 0; t < NUM_THREADS; t++) {
      int start = (t + id) % NUM_THREADS;
      for (i = start; i < src_GLOBAL->Alen; i = i + NUM_THREADS) {
          for (j = 0; j < src_GLOBAL->Blen; j++) {
              isum = 0;
              for (k = id; k < src_GLOBAL->Strlen; k = k + NUM_THREADS) {
                  if (src_GLOBAL->A[i][k] != src_GLOBAL->B[j][k]) {
                      psum++;
                      isum++;
                  }
              }
              pthread_mutex_lock(&hammlock[i]);
              hammingValues_GLOBAL[i][j] += isum;
              pthread_mutex_unlock(&hammlock[i]);
          }
      }
  }

  pthread_mutex_lock(&lock);
  sum_GLOBAL += psum;
  pthread_mutex_unlock(&lock);

...
```

2.3 **Task B** (Παραλληλοποίηση στα cell)

Σε αυτήν την υλοποίηση έγινε παραλληλοποίηση στα Cell, αυτό σημαίνει ότι το κάθε Thread θα υπολογίσει διαφορετικές θέσεις του Hamming Table. Η παρακάτω εικόνα περιγράφει την υλοποίηση που πραγματοποιήθηκε (η επιλογή αυτή έγινε στα Pthreads, **δεν είναι ίδια** με αυτή των OpenMp καθώς αφού είναι πιο high level υλοποίηση επιλέχθηκε το default scheduling το οποίο μπορεί να διαφέρει).



Εικόνα 2.4: Παράλληλη υλοποίηση στα Cell

Παρόλα αυτά μόνο ένα Thread θα ασχοληθεί με την σύγκριση των δύο λέξεων, του συνόλου A και B, με αποτέλεσμα να μην έχουμε τέτοιου είδους Data Races (όπως του Task A) σε αυτήν την περίπτωση. Για το OpenMp ο κώδικας που δημιουργήθηκε είναι ο εξής:

openMpHamming.c

```
…
      for(k=0;k<src->Strlen;k++){
          #pragma omp for collapse(2) nowait
          for(i=0;i<src->Alen;i++){
              for(j=0;j<src->Blen;j++){
                  if (src->A[i][k] != src->B[j][k]) {
                      hammingValues[i][j]++;
                      psum++;
                  }
              }
          }
       }

       #pragma omp atomic
       sum += psum;
...
```

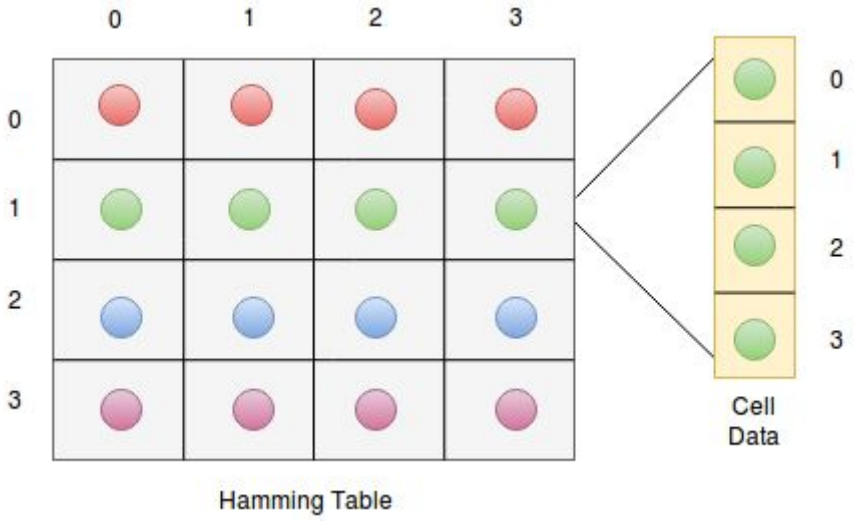
Για να γίνει η υλοποίηση με Pthreads έγινε με παρόμοιο τρόπο με αυτό που κάνει το collapse στο OpenMp, δημιουργήθηκε ένα “μεγάλο” loop στο οποίο μετατρέπει τον δισδιάστατο πίνακα σε μονοδιάστατο και μετά για το κάθε Thread βρίσκει τα Cell τα οποία θα υπολογίσει. Η υλοποίηση είναι η εξής:

pthreadsHamming.c

```
…
   for (index = id; index < src_GLOBAL->Alen * src_GLOBAL->Blen; index = index +
NUM_THREADS) {

        //Calculate i and j
        if (src_GLOBAL->Alen == 1) {
            i = 0;
            j = index;
        } else if (src_GLOBAL->Blen == 1) {
            j = 0;
            i = index;
        } else {
            i = index % src_GLOBAL->Alen;
            j = index / src_GLOBAL->Alen;
        }

        for (k = 0; k < src_GLOBAL->Strlen; k++) {
            if (src_GLOBAL->A[i][k] != src_GLOBAL->B[j][k]) {
                hammingValues_GLOBAL[i][j]++;
                psum++;
            }
        }
    }

    pthread_mutex_lock(&lock);
    sum_GLOBAL += psum;
    pthread_mutex_unlock(&lock);

...
```

2.4 **Task C** (Παραλληλοποίηση στα row)

Για να υλοποιηθεί αυτό το task αρκεί να γινόταν η παρακάτω προσπέλαση:



Εικόνα 2.5: Παράλληλη υλοποίηση στα Row

Σε αυτήν την περίπτωση κάθε Thread υπολογίζει ένα word του ενός συνόλου με όλο το άλλο σύνολο. Ούτε σε αυτήν εμφανίζονται Data Races τύπου όπως το Task A διότι κάθε Thread υπολογίζει διαφορετικό Cell.

Η υλοποίηση σε OpenMp:

openMpHamming.c

```
...
      for (i = 0; i < src->Alen; i++) {
          for (k = 0; k < src->Strlen; k++) {
              #pragma omp for nowait
              for (j = 0; j < src->Blen; j++) {
                  if (src->A[i][k] != src->B[j][k]) {
                      hammingValues[i][j]++;
                      psum++;
                  }
              }
          }
       }
       #pragma omp atomic
       sum += psum;
...
```

Η υλοποίηση σε Prthread:

pthreadsHamming.c

```
…
   for (i = 0; i < src_GLOBAL->Alen; i++) {
       for (k = 0; k < src_GLOBAL->Strlen; k++) {
           for (j = id; j < src_GLOBAL->Blen; j = j + NUM_THREADS) {
               if (src_GLOBAL->A[i][k] != src_GLOBAL->B[j][k]) {
                   hammingValues_GLOBAL[i][j]++;
                   psum++;
               }
           }
       }
   }

   pthread_mutex_lock(&lock);
   sum_GLOBAL += psum;
   pthread_mutex_unlock(&lock);
...
```

---

3 Εκτέλεση

Σε περίπτωση που θέλουμε να τρέξουμε τον κώδικα πρέπει να είμαστε σε Linux-Based σύστημα (χρειάζεται επίσης να έχουμε gcc και make). Μεταβαίνουμε με ένα terminal στο directory του project και αρκεί να κάνουμε τα εξής, ώστε να κάνουμε compile και run το εκτελέσιμο:

1. make
2. ./run

Σημείωση:
α) Σε αυτήν την περίπτωση το πρόγραμμα θα προσαρμοστεί στο σύστημα και θα επιλέξει τον μεγαλύτερο δυνατό αριθμό από Threads για τις παράλληλες υλοποιήσεις. Αν θέλετε να τρέξετε το πρόγραμμα για συγκεκριμένο αριθμό από Threads τότε πρέπει να αλλάξετε στο αρχείο **mystructs.h** την μεταβλητή **THREADS** από **BEST** στον αριθμό των Thread που επιθυμείτε.

β)Αν θέλετε να τρέξετε το πρόγραμμα για συγκεκριμένες τιμές (μέγεθος συνόλου A και B, μέγεθος συμβολοσειράς κλπ.) χρειάζεται να αλλάξετε στο αρχείο **mystructs.h** τις μεταβλητές **ARRAY_A, ARRAY_B, STRINGS_L, aLength, bLength, stringLength** όπως περιγράφετε στο in-line documentation που υπάρχει.
γ) Αν επιθυμείτε να αλλάξετε από random σε pseudo-random το **seed** αρκεί να αλλάξετε την τιμή του SEED στο αρχείο **mystructs.h.**
δ) Αν θέλουμε να διαγράψουμε τα *.o και το run αρχεία, μπορούμε να το κάνουμε τρέχοντας:

- make clean

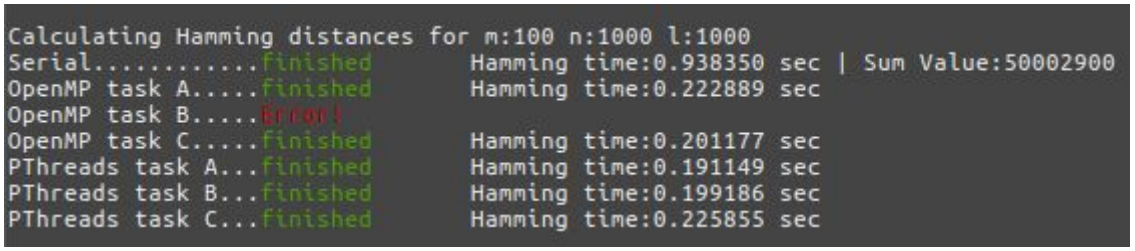
Αν όλα έχουν κυλίσει ομαλά τότε στο terminal εμφανίζεται το εξής:

```
Calculating Hamming distances for m:1 n:1 l:10
Serial............finished       Hamming time:0.000001 sec | Sum Value:4
OpenMP task A.....finished       Hamming time:0.013929 sec
OpenMP task B.....finished       Hamming time:0.004498 sec
OpenMP task C.....finished       Hamming time:0.000027 sec
PThreads task A...finished       Hamming time:0.013778 sec
PThreads task B...finished       Hamming time:0.003293 sec
PThreads task C...finished       Hamming time:0.000296 sec

Calculating Hamming distances for m:1 n:1 l:100
Serial............finished       Hamming time:0.000001 sec | Sum Value:36
OpenMP task A.....finished       Hamming time:0.000064 sec
OpenMP task B.....finished       Hamming time:0.000024 sec
OpenMP task C.....finished       Hamming time:0.000020 sec
PThreads task A...finished       Hamming time:0.013833 sec
PThreads task B...finished       Hamming time:0.000116 sec
PThreads task C...finished       Hamming time:0.000105 sec

Calculating Hamming distances for m:1 n:1 l:1000
Serial............finished       Hamming time:0.000011 sec | Sum Value:533
OpenMP task A.....finished       Hamming time:0.000016 sec
OpenMP task B.....finished       Hamming time:0.000038 sec
OpenMP task C.....finished       Hamming time:0.000029 sec
PThreads task A...finished       Hamming time:0.000113 sec
PThreads task B...finished       Hamming time:0.000114 sec
PThreads task C...finished       Hamming time:0.000128 sec

Calculating Hamming distances for m:1 n:100 l:10
Serial............finished       Hamming time:0.000012 sec | Sum Value:497
OpenMP task A.....finished       Hamming time:0.000064 sec
OpenMP task B.....finished       Hamming time:0.000006 sec
OpenMP task C.....finished       Hamming time:0.000011 sec
PThreads task A...finished       Hamming time:0.000133 sec
PThreads task B...finished       Hamming time:0.000112 sec
PThreads task C...finished       Hamming time:0.014284 sec

Calculating Hamming distances for m:1 n:100 l:100
Serial............finished       Hamming time:0.000168 sec | Sum Value:4975
OpenMP task A.....finished       Hamming time:0.013503 sec
OpenMP task B.....finished       Hamming time:0.000059 sec
OpenMP task C.....finished       Hamming time:0.000042 sec
PThreads task A...finished       Hamming time:0.009320 sec
PThreads task B...finished       Hamming time:0.005236 sec
PThreads task C...finished       Hamming time:0.000242 sec

Calculating Hamming distances for m:1 n:100 l:1000
Serial............finished       Hamming time:0.001092 sec | Sum Value:49975
OpenMP task A.....finished       Hamming time:0.010702 sec
OpenMP task B.....finished       Hamming time:0.000299 sec
OpenMP task C.....finished       Hamming time:0.000307 sec
PThreads task A...finished       Hamming time:0.001211 sec
PThreads task B...finished       Hamming time:0.001164 sec
PThreads task C...finished       Hamming time:0.001283 sec

Calculating Hamming distances for m:1 n:1000 l:10
Serial............finished       Hamming time:0.000117 sec | Sum Value:5023
OpenMP task A.....finished       Hamming time:0.000057 sec
OpenMP task B.....finished       Hamming time:0.000021 sec
OpenMP task C.....finished       Hamming time:0.000021 sec
PThreads task A...finished       Hamming time:0.000385 sec
PThreads task B...finished       Hamming time:0.000247 sec
PThreads task C...finished       Hamming time:0.000236 sec
```

Εικόνα 3.1: Program output

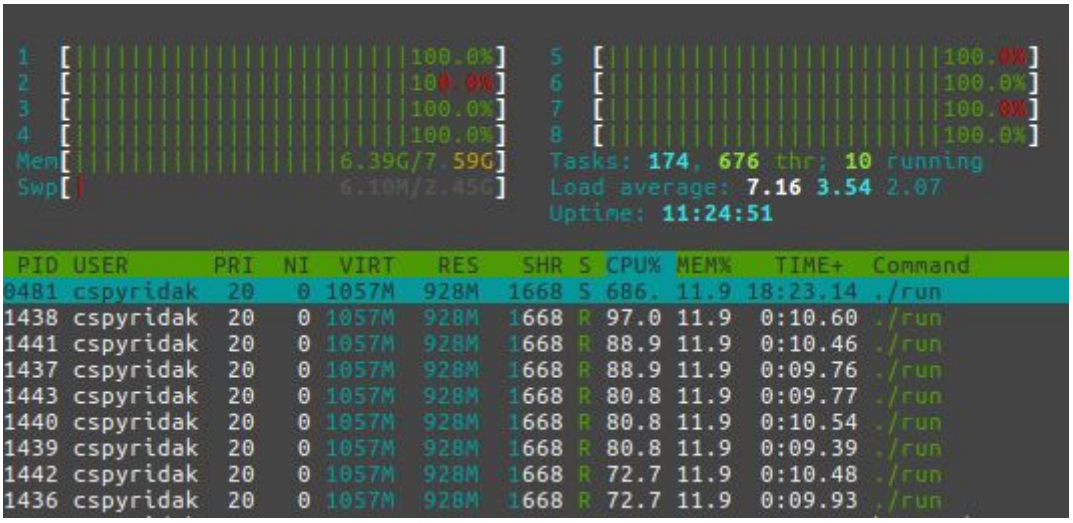
Στην περίπτωση που εμφανίζεται με πράσινο η λέξη finished δίπλα από μία υλοποίηση και μετά ο χρόνος που χρειάστηκε για αυτή, αυτό σημαίνει ότι το ολικό sum (που αναγράφεται μόνο στην σειριακή υλοποίηση) της συγκεκριμένης περίπτωσης είναι το ίδιο με αυτό της σειριακής άρα δεν υπάρχει λάθος υπολογισμός. Σε περίπτωση λανθασμένου sum τότε

εμφανίζεται το εξής (το συγκεκριμένο χρησιμοποιήθηκε στα πρώτα στάδια του project ώστε να λυθούν προβλήματα τα οποία υπήρξαν):



Εικόνα 3.2: Λανθασμένος υπολογισμός

Κατά την εκτέλεση των παράλληλων υλοποιήσεων με χρήση 8 Threads μπορούμε να παρατηρήσουμε στο htop τα εξής. Στο οποίο μπορούμε να παρατηρήσουμε ότι το πρόγραμμα μας αξιοποιεί όλα τα Threads που του ζητήθηκαν.

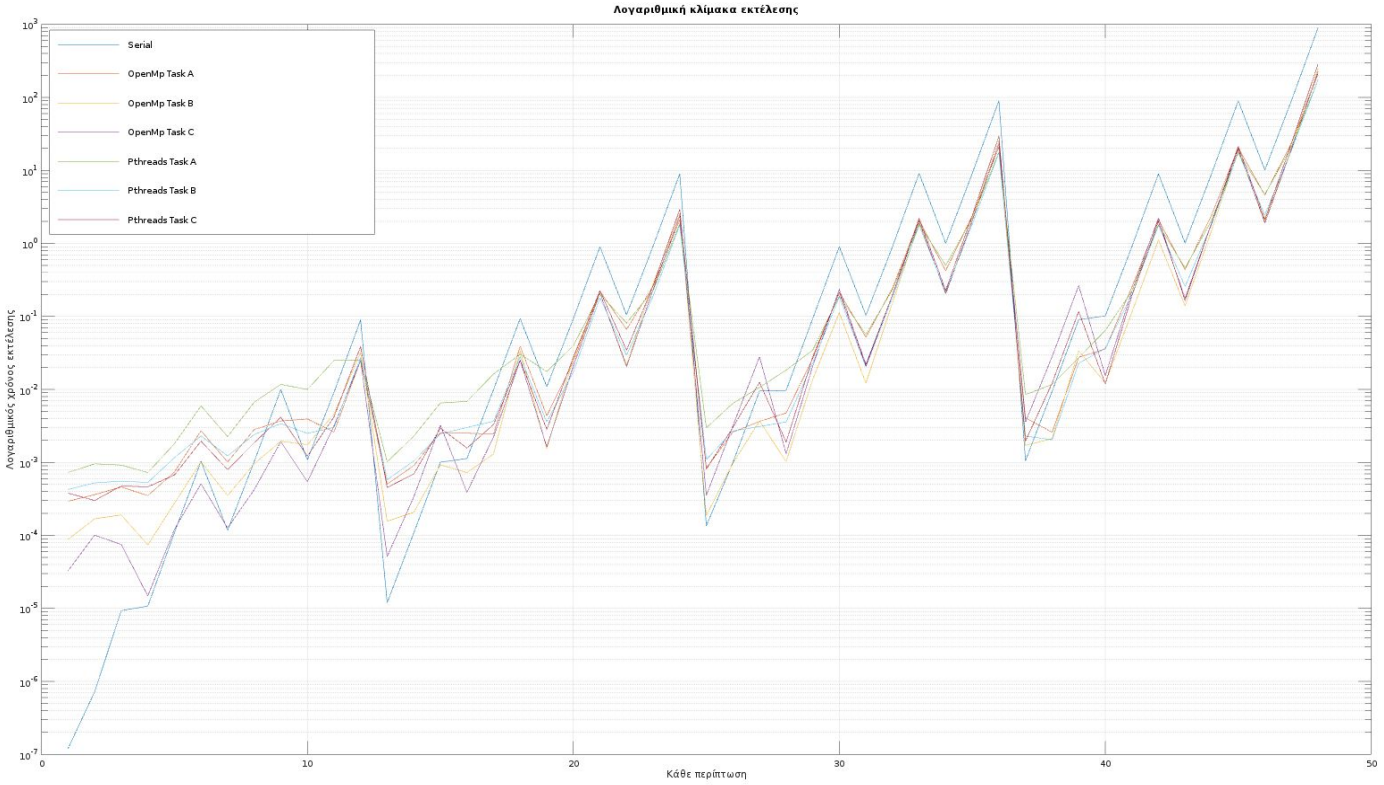


Εικόνα 3.3: Παράλληλή εκτέλεση

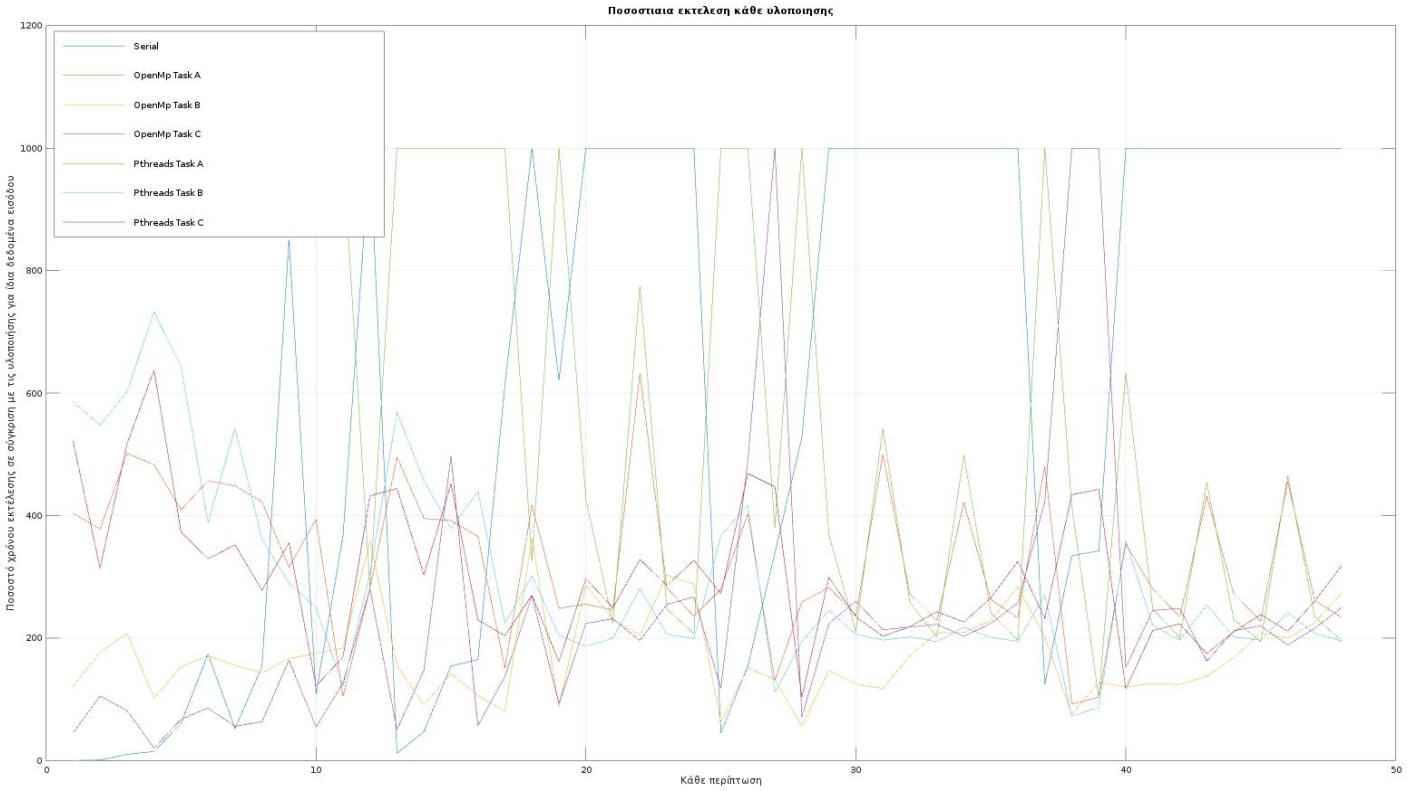
Σημείωση: Επειδή σε laptop με τετραπύρηνο επεξεργαστή για τον υπολογισμό όλων των περιπτώσεων που ζητήθηκαν (από μία φορά όχι για την πιθανοτική ανάλυση) χρειάστηκε περίπου 50-60 λεπτά να ολοκληρωθεί το πρόγραμμα, στο directory του project υπάρχει το sample_out.txt στο οποίο εμφανίζονται ακριβώς αυτά που εμφανίστηκαν στο terminal για μία τυχαία εκτέλεση χρησιμοποιώντας 8 threads.

4 Συμπεράσματα

Αφού εκτελέστηκε 1000 φορές το πρόγραμμα (μόνο για τις μικρές επαναλήψεις ώστε να δημιουργηθεί μία πιο ακριβής άποψη στα τελικά αποτελέσματα). Δημιουργήθηκαν τα παρακάτω διαγράμματα. Στο ένα παρουσιάζεται για κάθε είσοδο δεδομένων η ποσοστιαία διαφορά των χρόνων εκτέλεσης της μίας υλοποίησης από την άλλη, ενώ στο άλλο σε λογαριθμική κλίμακα οι χρόνοι όλων των υλοποιήσεων.
Σημείωση: Τα παρακάτω διαγράμματα αναφέρονται για παράλληλη εκτέλεση σε 8 threads.



Εικόνα 4.1: Χρόνοι για κάθε εκτέλεση σε λογαριθμική κλίμακα



Εικόνα 4.2: Ποσοστό χρόνου εκτέλεσης σε σύγκριση με τις άλλες υλοποιήσεις για κοινές εισόδους

Από τα παραπάνω διαγράμματα μπορούμε να συμπεράνουμε τα εξής:

1. Για **πολύ μικρές** επαναλήψεις ο σειριακός τρόπος εκτέλεσης παρουσιάζεται να είναι αρκετά πιο γρήγορος **από κάθε μορφή** παραλληλοποίησης που υλοποιήθηκε.
2. Σε περιπτώσεις όπου μας συμφέρει να υπολογίσουμε παράλληλα υπολογισμούς με το συγκεκριμένο σύστημα καταφέραμε να επιταχύνουμε κάποιες υλοποιήσεις εώς και πάνω από 4 φορές ταχύτερο χρόνο εκτέλεσης.
3. Το Task A και στα Pthreads και στο OpenMp είναι η πιο αργή υλοποίηση σε σύγκριση με τις άλλες.
4. Το Task C παρουσιάστηκε και στους δύο τρόπους παραλληλοποίησης να είναι ελάχιστα πιο η γρήγορη εκδοχή.

Τέλος ελέγχθηκε πόσο επηρεάζει ο αριθμός των threads για την εκτέλεση ενός προγράμματος. Συγκεκριμένα για τις ίδιες εισόδους (m=1000, n=1000, l=1000 και ίδιες αρχικοποίησης των πινάκων με την χρήση του seed=100) υπολογίσαμε τις αποστάσεις hamming με διαφορετικό πλήθος από threads, σε όλες η απόσταση βγήκε ίδια (ίση με 500020056) όμως όπως ήταν αναμενόμενο ήταν διαφορετικοί οι χρόνοι υλοποίησης.

	2 Threads	4 Threads	6 Threads	8 Threads
Serial	9.655848	9.680156	9.429436	9.796154
OpenMp A	5.810829	3.106437	2.639129	2.231162
OpenMp B	3.287979	1.736472	1.896400	2.196999
OpenMp C	6.018426	2.844537	2.423421	2.044075
Pthreads A	4.902999	2.468502	2.187701	1.746219
Pthreads B	4.854484	2.599501	2.169397	1.817826
Pthreads C	5.955975	3.208010	2.596825	2.273619

Πίνακας 4.3: Για ίδια δεδομένα εισόδου και διαφορετικό πλήθος Thread χρόνοι υλοποίησης

Από τον παραπάνω πίνακα μπορούμε να συμπεράνουμε τα εξής:

- Όσο αυξάνεται το πλήθος των thread μέχρι να φτάσουμε στο μέγιστο αποδεκτό από το σύστημα ο υπολογισμός γίνεται ταχύτερος.
- Παρόλα αυτά επειδή δεν είναι πραγματικοί οι πυρήνες δεν περιμένουμε να βγάλουν την ίδια δουλειά που θα έβγαζαν 8 πυρήνες, έτσι το hyperthreading προσπαθεί απλά να βελτιώσει “κακό” προγραμματισμό.