ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Παράλληλα και Κατανεμημένα Συστήματα Υπολογιστών

Άσκηση 2

(Μέρος 1ο)

ΧΑΤΖΗΘΩΜΑ ΑΝΤΡΕΑΣ

ΑΕΜ: 8026

antreasc@ece.auth.gr

Υπεύθυνος καθηγητής: κ. Πιτσιάνης Νίκος
Ημερομηνία Παράδοσης: 06/01/2016

ΕΙΣΑΓΩΓΗ

Το ζητούμενο της εργασίας αυτής ήταν η υλοποίηση ενός παράλληλου προγράμματος σε MPI για την υλοποίηση του κατανεμημένου αλγόριθμου αναζήτησης και εύρεσης του πιο κοντινού γείτονα (knn) για κάθε σημείο $q \in Q$, ανάμεσα σε σημεία $c \in C$. Όπως αναφέρεται και στην εκφώνηση της άσκησης, τα σύνολα των σημείων Q και C προέρχονται από ομοιόμορφες κατανομές (uniform distributions), εντός του μοναδιαίου κύβου $[0, 1)^3$ στον τρισδιάστατο χώρο.

Για την υλοποίηση του αλγόριθμου, αναπτύχθηκε πρώτα ένας σειριακός αλγόριθμος ο οποίος ακολουθεί τις οδηγίες και τις προϋποθέσεις της εκφώνησης με εξαίρεση τις οδηγίες που αφορούν το παράλληλο κομμάτι. (αρχείο $knn\_serial.c$)

Ακολούθως, στο ίδιο αρχείο, αναπτύχθηκε υπό τη μορφή συνάρτησης, ένας brute force αλγόριθμος αναζήτησης ούτως ώστε να επαληθευτεί η ορθότητα των αποτελεσμάτων.

Και τέλος, το σειριακό πρόγραμμα μετατράπηκε σε παράλληλο με τη χρήση MPI. (αρχείο $knn\_mpi\_v2.c$)

Το compile για την σειριακή υλοποίηση έγινε με την εντολή:

$$gcc\ knn\_serial.c - lm - O3 - o\ serial$$

και run με την εντολή:

$$./serial\ N\ n\ m\ k$$

Αντίστοιχα, για την υλοποίηση με MPI το compile έγινε με την εντολή:

$$mpicc\ knn\_mpi.c - lm - O3 - o\ mpi$$

και run με την εντολή:

$$mpirun - np\ P\ ./mpi\ N\ n\ m\ k$$

Όπου και στις 2 περιπτώσεις τα N, n, m, k ισοδυναμούν με $2^N, 2^n, 2^m, 2^k$ αντίστοιχα και στην υλοποίηση mpi, P είναι ο αριθμός των διεργασιών.

**Σημειώνεται ότι τα διαγράμματα, οι παρατηρήσεις αλλά και τα συμπεράσματα από τους χρόνους εκτέλεσης του σειριακού αλλά και του παράλληλου προγράμματος, θα παραδοθούν σε μεταγενέστερη αναφορά λόγο αδυναμίας πρόσβασης στο Hellasgrid*.

Σειριακός Αλγόριθμος

Για περισσότερη άνεση και ευκολία κατά την διαδικασία της υλοποίησης του ζητούμενου προγράμματος σειριακά, δημιουργήθηκαν 4 δομές (structs) και 12 συναρτήσεις (συμπεριλαμβανομένου και της main).

Η υλοποίηση του σειριακού αλγόριθμου βρίσκεται στο αρχείο $knn\_serial.c$

Οι δομές που δημιουργήθηκαν είναι οι ακόλουθες:

- **minD**: χρησιμοποιείται για την αποθήκευση των αποτελεσμάτων της σειριακής knn υλοποίησης καθώς αλλά και της brute force υλοποίησης, ούτως ώστε στο τέλος να γίνει μια σύγκριση των αποτελεσμάτων και να ελεγχθεί η ορθότητα του αλγόριθμου.

- **cube**: χρησιμοποιείται για την αποθήκευση στοιχείων που αφορούν τον κύβο $(0, 1]^3$.

  Στην δομή αυτή αποθηκεύονται: ο αριθμός των $n \times m \times k$, ο αριθμός των υποκύβων που θα δημιουργηθούν ($subCubesNum$) καθώς και τα διαστήματα που θα δημιουργηθούν για τον κάθε υποκύβο ($nStep, mStep, kStep$) στους αντίστοιχους άξονες $x,\ y,\ z$.

- **points**: χρησιμοποιείται για την αποθήκευση στοιχείων που αφορούν τα σημεία Q και C που δημιουργούνται.
  Στη δομή αυτή αποθηκεύονται: ο αριθμός των στοιχείων που θα δημιουργηθούν (N), καθώς και όλα τα σημεία Q και C που δημιουργούνται σε δισδιάστατους πίνακες ($** pointsQ\ και\ ** pointsC$)

- **subCubes**: χρησιμοποιείται για την αποθήκευση στοιχείων που αφορούν κάθε υποκύβο που δημιουργείται. Κάθε υποκύβος προσομοιώνεται δημιουργώντας ένα αντικείμενο της δομής αυτής, το οποίο αντικείμενο περιέχει ένα τριπλό pointer ούτως ώστε κάθε υποκύβος να αντιστοιχεί σε ένα σετ τριών συντεταγμένων.

  Στη δομή αυτή αποθηκεύονται: ο αριθμός των σημείων C και Q που ανήκουν στον εκάστοτε υποκύβο ($cntQ\ και\ cntC$), καθώς και η θέση κάθε σημείου στον αντίστοιχο πίνακα της δομής points ($* pntAddrQ\ και\ * pntAddrC$).

Οι συναρτήσεις που δημιουργήθηκαν είναι οι παρακάτω:

- **randZeroToOne()**: επιστρέφει έναν πραγματικό αριθμό μεταξύ $(0, 1]$.

- **initPoints()**: δεσμεύει την απαραίτητη μνήμη για τους πίνακες αποθήκευσης των σημείων Q και C. Τέλος, δίνει τιμές στους πίνακες χρησιμοποιώντας την συνάρτηση randZeroToOne().

- **initSubCubes()**: δεσμεύει την απαραίτητη μνήμη για όλους τους υποκύβους (αντικείμενο της δομής subCubes).

- **sortPointsInSubCubes()**: ταξινομεί όλα τα σημεία στους υποκύβους τους οποίους ανήκουν. Κατά την ταξινόμηση, αποθηκεύεται σε πίνακα η θέση κάθε σημείου. Η θέση αυτή, δείχνει την θέση του σημείου στον πίνακα της δομής points. Κάθε υποκύβος, έχει τον αντίστοιχο πίνακα αποθήκευσης θέσεων σημείων.

- **limits(int x, int y, int z)**: ελέγχει κατά πόσο ένα σετ τριών συντεταγμένων ενός υποκύβου, είναι εντός ή εκτός του μοναδιαίου κύβου $(0, 1]^3$ και ανάλογα επιστρέφει 1 ή 0 αντίστοιχα.

- **distance(float a[3], float b[3])**: υπολογίζει και επιστρέφει την απόσταση μεταξύ 2 σημείων.

- **knnSearch()**: υλοποιεί τον αλγόριθμο αναζήτησης knn χρησιμοποιώντας όλες τις παραπάνω συναρτήσεις. Ο αλγόριθμος, παίρνει κάθε ένα υποκύβο ξεχωριστά, κοιτάει πόσα σημεία Q βρίσκονται στον υποκύβο αυτό και ακολούθως παίρνει τους παράπλευρους* υποκύβους και ελέγχει τις αποστάσεις μεταξύ των σημείων C και κρατάει την μικρότερη. **Ως παράπλευροι υποκύβοι ορίζονται οι υποκύβοι που έχουν τουλάχιστον μία κοινή ακμή, πλευρά ή κορυφή. Δηλαδή, ένας υποκύβος μπορεί να έχει μέχρι και 26 παράπλευρους υποκύβους.*

- **print_Q_C()**: εκτυπώνει όλα τα σημεία του συνόλου Q αλλά και του C. Χρησιμοποιήθηκε μόνο για μικρό αριθμό στοιχείων για λόγους debugging αλλά και ελέγχου ορθότητας του αλγόριθμου.

- **printCubepointsNum()**: εκτυπώνει πόσα στοιχεία έχει κάθε υποκύβος. Η υλοποίηση της έγινε καθαρά για σκοπούς debugging και ελέγχου ορθότητας.

- **bruteForceMethod()**: Brute force υλοποίηση για την αναζήτηση του κοντινότερου σημείου. Ελέγχει κάθε Q σημείο με όλα τα C και αποθηκεύει την μικρότερη απόσταση.

- **test()**: ελέγχει κατά πόσο κάθε σημείο, έχει για κοντινότερο σημείο το ίδιο σημείο είτε με την brute force υλοποίηση είτε με την σειριακή knn υλοποίηση.

- **main()**: καλεί όλες τις απαραίτητες συναρτήσεις για την υλοποίηση όλου του αλγόριθμου. Επίσης κρατάει τον συνολικό χρόνο εκτέλεσης που χρειάζεται κάθε υλοποίηση. Στον χρόνο αυτό, δεν συμπεριλαμβάνεται ο χρόνος δέσμευσης μνήμης καθώς και ο χρόνος παραγωγής τον αρχικών σημείων. Συμπεριλαμβάνεται όμως, ο χρόνος ταξινόμησης των στοιχείων σε υποκύβους καθώς και ο χρόνος περάτωσης της knn αναζήτησης μέχρι και το τελευταίο σημείο.

Το πρόγραμμα έχει εκτελεστεί σε προσωπικό Η/Υ και έχει ελεγχθεί η ορθότητα του, όπως αναφέρθηκε και παραπάνω.

Παράλληλος Αλγόριθμος (MPI)

Στην παράλληλη υλοποίηση χρησιμοποιήθηκαν οι δομές και οι συναρτήσεις οι οποίες υλοποιήθηκαν στην σειριακή έκδοση του αλγορίθμου, με όλες τις απαραίτητες αλλαγές.

Η υλοποίηση του παράλληλου αλγόριθμου βρίσκεται στο αρχείο $knn\_mpi\_v2.c$. Το αρχείο με το όνομα $knn\_mpi.c$ να αγνοηθεί καθώς έγινε μια μικρή διόρθωση.

Για την παράλληλη υλοποίηση προστέθηκε η συνάρτηση **mpiImplement()**, στην οποία γίνονται όλες οι απαραίτητες επικοινωνίες μεταξύ των διεργασιών.

Επίσης, τα σημεία τώρα δεν αποθηκεύονται σε δισδιάστατο πίνακα αλλά σε μονοδιάστατους πίνακες (ένας για κάθε x, y, z του συνόλου Q και ένας για κάθε x, y, z του συνόλου C). Ο λόγος είναι για περισσότερη ευκολία στη μεταφορά δεδομένων από την μια διεργασία στην άλλη.

Η υλοποίηση του προγράμματος

Ως γνωστό, στην αρχή του προγράμματος πρέπει να γίνει η αρχικοποίηση του περιβάλλοντος εκτέλεσης MPI. Αυτό γίνεται στην αρχή της συνάρτησης main χρησιμοποιώντας την ρουτίνα MPI_Init και ακολούθως με τις αντίστοιχες ρουτίνες, προσδιορίζεται το αναγνωριστικό της καλούμενης διεργασίας (pID) μέσα στον communicator καθώς ο αριθμός των διεργασιών ($nProc$). Έπειτα, γίνονται οι απαραίτητες δεσμεύσεις μνήμης, όπως και στην σειριακή υλοποίηση. Για τον διαμοιρασμό των δεδομένων στις διάφορες διεργασίες, που θα χρειαστεί να γίνει αργότερα, δημιουργείται ανά διεργασία, ένα περισσότερο αντικείμενο στη δομή $subCubes$ και ένα στη δομή $points$, τα οποία φέρουν το αρχικό “proc” από την λέξη process. Αυτό γίνεται ούτως ώστε κάθε διεργασία, να μπορεί να συλλέξει όλα τα απαραίτητα δεδομένα που χρειάζεται για να εκτελέσει την αναζήτηση του κοντινότερου γείτονα στο χώρο τον οποίο θα της ανατεθεί. Αυτό θα γίνει καλύτερα κατανοητό και παρακάτω.

Αφού γίνουν όλες οι απαραίτητες δεσμεύσεις μνήμης, κάθε διεργασία παράγει $N/nProc$ σημεία, όπου $N = ο\ αριθμός\ των\ συνολικών\ σημείων$ και $nProc = ο\ αριθμός\ των\ διεργασιών$. Έτσι, με αυτόν τον τρόπο, αν προσθέσουμε όλα τα σημεία των διεργασιών, δημιουργούνται N συνολικά σημεία Q και N σημεία C. Για την δημιουργία των σημείων, χρησιμοποιήθηκε η συνάρτηση $randZeroToOne$ καθώς και η $srand(seed)$, όπου $\text{seed} = (\text{time} * (\text{pID} + 1)$ ούτως ώστε να αποφευχθεί η δημιουργία ίδιων σημείων.

Έπειτα, γίνεται από κάθε διεργασία ταξινόμηση των σημείων που τις ανήκουν σε υποκύβους και αποθηκεύεται σε κάθε υποκύβο, η αντίστοιχη θέση του σημείου αυτού. Η θέση αυτή, συνδυάζει το χαρακτηριστικό της διεργασίας pID καθώς και την πραγματική θέση του σημείου στον πίνακα που ήδη ανήκει το σημείο, ούτως ώστε αργότερα στην συγκέντρωση όλων τον σημείων να μην γίνεται επανεγγραφή (overwrite) κάποιου σημείου από κάποιο άλλο.

Αφού γίνει η ταξινόμηση των σημείων, καλείται από κάθε διεργασία η συνάρτηση $mpiImpliment$. Στη συνάρτηση αυτή, όπως αναφέρθηκε και παραπάνω, γίνονται όλες οι απαραίτητες επικοινωνίες μεταξύ των διεργασιών.

Στην αρχή, η κάθε διεργασία στέλνει σε όλες τις υπόλοιπες τα ταξινομημένα της σημεία (τις θέσεις τους στον αρχικό πίνακα παραγωγής τους). Για να γίνει αυτό, κάθε διεργασία πρέπει να στείλει στις υπόλοιπες, τα σημεία (θέσεις) ανά υποκύβο τα οποία έχει ταξινομήσει. Έτσι, για κάθε υποκύβο, καλείται η ρουτίνα $MPI\_Allgather$ ούτως ώστε κάθε διεργασία να ενημερωθεί από όλες τις υπόλοιπες διεργασίες το μέγεθος των δεδομένων που θα συλλεχθούν και έπειτα υπολογίζονται οι μετατοπίσεις (displacements) διότι κάθε υποκύβος δεν περιέχει σταθερό αριθμό σημείων. Αφού υπολογίστηκαν οι μετατοπίσεις, γίνεται χρήση της ρουτίνας $MPI\_Allreduce$ ούτως ώστε να υπολογιστεί ο συνολικός αριθμός των σημείων ανά υποκύβο, από όλες τις διεργασίες και έτσι να γίνει η κατάλληλη δέσμευση μνήμης για να μπορέσει κάθε διεργασία να υποδεχθεί τον όγκο δεδομένων (όλα τα σημεία ταξινομημένα σε υποκύβους). Με την χρήση τώρα της ρουτίνας $MPI\_Allgatherv$, συλλέγονται από όλες τις διεργασίες όλες οι θέσεις των σημείων ανά υποκύβο.

Τέλος, με παρόμοια διαδικασία και με τη χρήση της ρουτίνας $MPI\_Allgatherv$ διαμοιράζονται σε όλες τις διεργασίες τα αρχικά σημεία που παράχθηκαν από κάθε διεργασία και συλλέγονται σε πίνακες ούτως ώστε από κάθε διεργασία να μπορεί να γίνεται η αντιστοίχηση της θέσης του σημείου με το ίδιο το σημείο.

Μετά το πέρας της συνάρτησης $mpiImpliment$, κάθε διεργασία έχει τις θέσεις των σημείων αποθηκεμένες στους πίνακες $*pntAddrQ$ και $*pntAddrC$ του αντικειμένου $***procSC$ της δομής $subCube$ όπως επίσης και όλα τα σημεία τα οποία είναι αποθηκευμένα σε μονοδιάστατους πίνακες στο αντικείμενο $procPnt$ της δομής $points$.

Εφόσον τελείωσε η εκτέλεση της $mpiImpliment$ και όλες οι επικοινωνίες έγιναν με επιτυχία, καλείται η συνάρτηση $knnSearch$. Στη συνάρτηση $knnSearch$ κάθε διεργασία εκτελεί τον αλγόριθμο που αναπτύχθηκε στη σειριακή υλοποίηση, με τη διαφορά ότι κάθε διεργασία πραγματοποιεί αναζήτηση σε διαφορετικό χώρο του κύβου. Ο χώρος αναζήτησης που πραγματοποιεί κάθε διεργασία είναι ενιαίος και όλες οι διεργασίες έχουν τον ίδιο αριθμό υποκύβων. Τα όρια αναζήτησης είναι

δυναμικά, συνδυάζοντας μέσα το χαρακτηριστικό pID της κάθε διεργασίας αλλά και τον συνολικό αριθμός διεργασιών $nProc$, ούτως ώστε ο χώρος αναζήτησης να κατακερματίζεται ανάλογα με τον συνολικό αριθμό διεργασιών. Τέλος, όταν γίνει η αναζήτηση και η εύρεση του κοντινότερου γείτονα για κάθε σημείο, το αποτέλεσμα αποθηκεύεται σε ένα πίνακα στη δομή $minD$ ούτως ώστε αργότερα να επαληθευτεί η ορθότητα των αποτελεσμάτων συγκρίνοντας τα με την μέθοδο brute force.

Εφόσον ολοκληρώθηκε η αναζήτησης από όλες τις διεργασίες, μέσω της διεργασίας $pID = 0$ λαμβάνεται ο συνολικός χρόνος εκτέλεσης του αλγόριθμου και εκτυπώνεται στην κονσόλα.

Τέλος, με τη χρήση της ρουτίνας $MPI\_Finalize$ τερματίζεται το περιβάλλον εκτέλεσης του MPI.

Έλεγχος ορθότητας

Όπως αναφέρθηκε και σε διάφορα σημεία παραπάνω, για τον έλεγχο των αποτελεσμάτων χρησιμοποιήθηκε η brute force μέθοδος κατά την οποία γινότανε έλεγχος κάθε σημείου με όλα τα υπόλοιπα και έτσι με αυτό τον τρόπο δεν υπήρχε περίπτωση λάθους. Παρόλα αυτά, η χρήση της brute force απαιτούσε μεγάλο χρόνο εκτέλεσης και για τον λόγο αυτό εφαρμόστηκε με επιτυχία για αριθμό σημείων εώς και $N = 2^{15}$. Εφόσον για $N = 2^{15}$ ο αλγόριθμος δουλεύει σωστά, τότε επαγωγικά, βγαίνει το συμπέρασμα ότι θα δουλεύει σωστά για οποιοδήποτε αριθμό N.

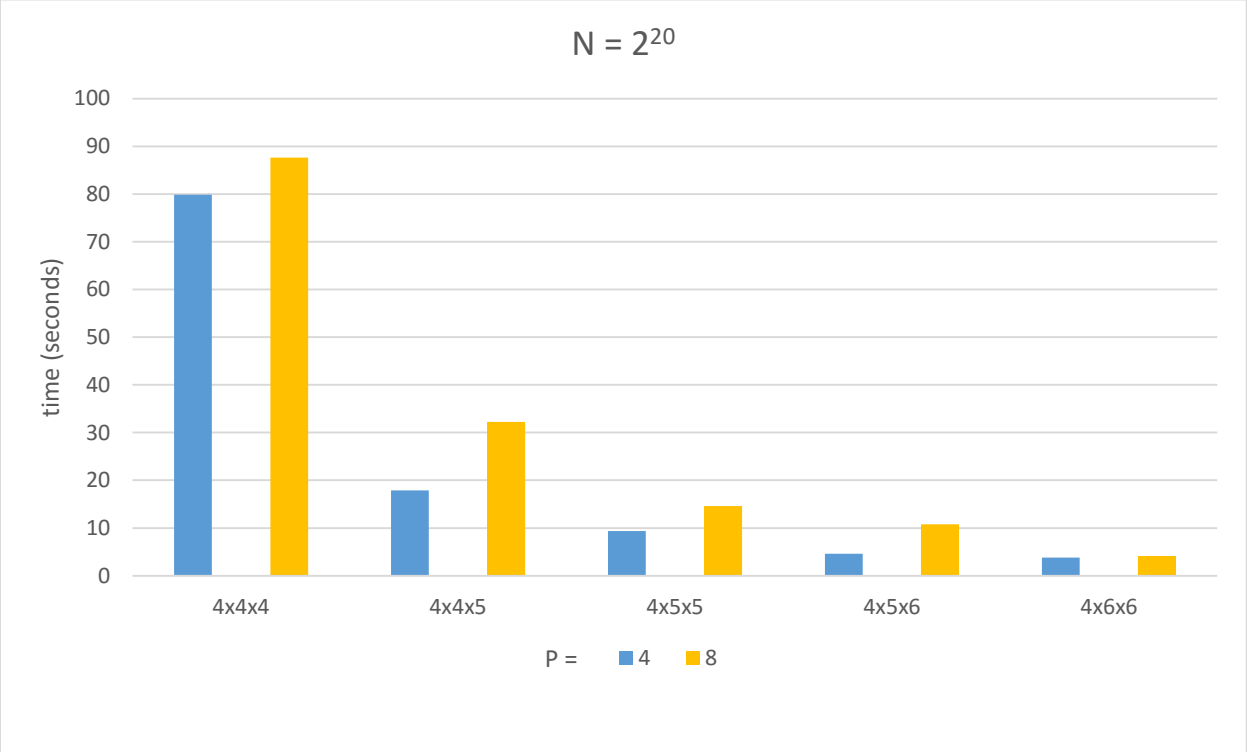
Επίσης, για το debugging του κώδικα αλλά και για τον έλεγχο της σωστής μετάδοσης δεδομένων μεταξύ των διεργασιών, χρησιμοποιήθηκαν εντολές $printf$ σε αρκετά σημεία του κώδικα, όπως επίσης και η χρήση της συνάρτησης $pint\_Q\_C$, η οποία αναπτύχθηκε για την εκτύπωση όλων των σημείων (για μικρό N). Εκτυπώνοντας όλα τα σημεία, έγινε εκτέλεση του αλγόριθμου χειροκίνητα και ταυτοποίηση των αποτελεσμάτων.

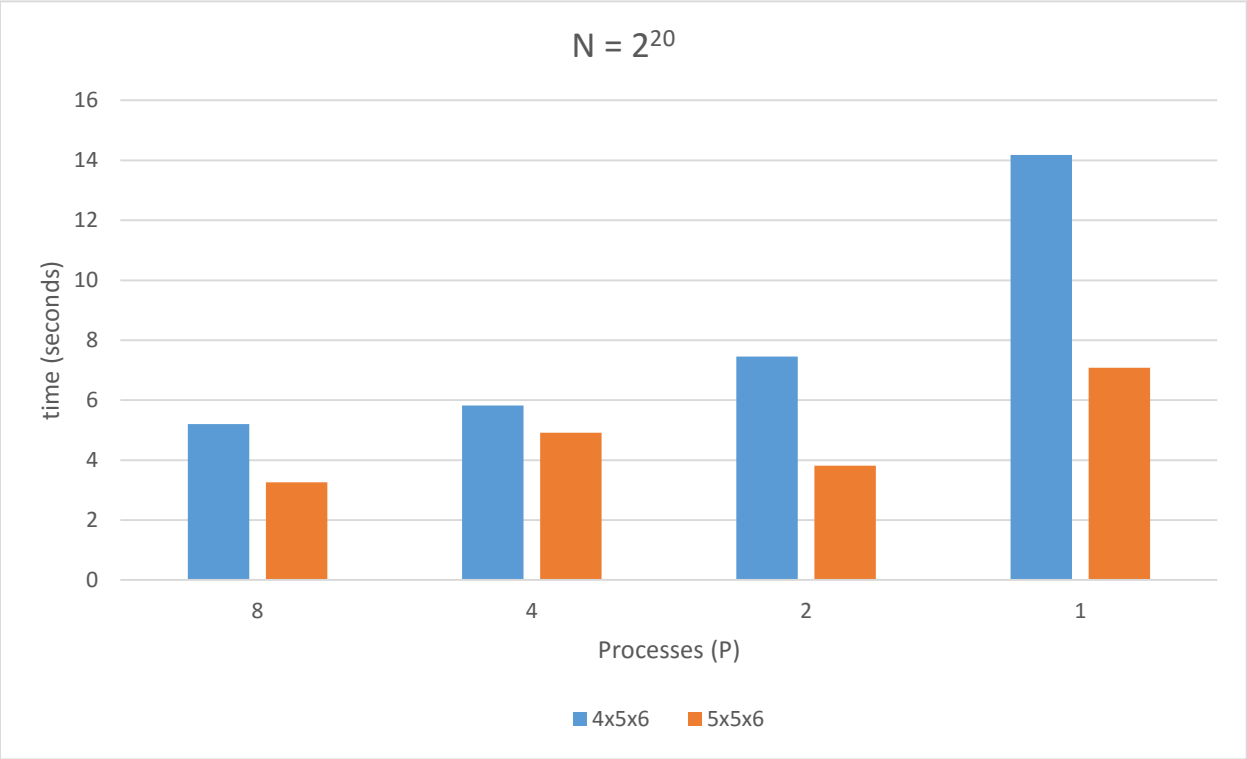
Αποτελέσματα – Παρατηρήσεις – Συμπεράσματα

Όπως αναφέρθηκε και στην εισαγωγή, λόγο τεχνικών προβλημάτων με το Hellasgrid, δεν κατέστει δυνατό να παρθούν τα αποτελέσματα εντός του χρόνου παράδοσης της εργασίας και για αυτόν τον λόγο θα παραδοθούν εκπρόθεσμα στο Μέρος 2ο.

Από μερικές μετρήσεις που έγιναν σε προσωπικό Η/Υ, η βελτίωση στο χρόνο εκτέλεσης με τη χρήση του παράλληλου προγράμματος έναντι του σειριακού προγράμματος, είχε γίνει ιδιαίτερα αισθητή. Επίσης η χρήση της brute force μεθόδου για μεγάλο αριθμό σημείων, απαιτούσε αρκετά λεπτά εώς και ώρες για την περάτωση της αναζήτησης.

Παρακάτω παρατίθενται μερικές μετρήσεις, παρόλα αυτά, ο αριθμός των σημείων που δημιουργούνται είναι μικρός ($N = 2^{20}$), οπότε τα αποτελέσματα και τα συμπεράσματα θα είναι με κάθε επιφύλαξη εώς ότου γίνουν οι μετρήσεις στο Hellasgrid.





Από το 1ο διάγραμμα (στην προηγούμενη σελίδα) βγαίνει το συμπέρασμα ότι ο χρόνος εκτέλεσης του αλγόριθμου βελτιώνεται καθώς αυξάνεται αριθμός $n\ x\ m\ x\ k = 2^{[12:16]}$. Επίσης φαίνεται ότι σε μερικές περιπτώσεις όπου το $n\ x\ m\ x\ k$ είναι σχετικά μικρό, ο χρόνος εκτέλεσης ήταν καλύτερος με 4 διεργασίες παρά με 8. Αυτό ίσως να οφείλεται στο χρόνο που σπαταλούν οι διεργασίες για να επικοινωνήσουν μεταξύ τους.

Από το 2ο διαγράμματα φαίνεται η σειριακή υλοποίηση με P = 1 και κάποιες παράλληλες υλοποιήσεις με P = 2, 4 και 8. Εδώ βγαίνει το συμπέρασμα ότι ο χρόνος εκτέλεσης του αλγορίθμου βελτιώνεται καθώς αυξάνεται ο αριθμός των διεργασιών.

Τα συμπεράσματα όμως αυτά, ίσως να μην είναι πολύ ακριβές για τον λόγο ότι οι αλγόριθμοι έτρεξαν για μικρό αριθμό σημείων (N) αλλά και μικρό αριθμό διεργασιών. Καλύτερα και πιο εύστοχα συμπεράσματα, θα φανούν μετά την ολοκλήρωση της εκτέλεσης των αλγόριθμων στην υπολογιστική μονάδα Hellasgrid τα οποία θα αναλυθούν εκπρόθεσμα στο 2ο Μέρος της αναφοράς.