# Εργασία για το μάθημα : ΠΡΟΗΓΜΕΝΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ: ΤΕΧΝΙΚΕΣ ΒΕΛΤΙΣΤΟΠΟΙΗΣΗΣ ΚΩΔΙΚΑ ΓΙΑ ΠΟΛΥΕΠΕΞΕΡΓΑΣΤΙΚΕΣ ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ

Alexis Apostolakis ECE-NTUA PhD candidate AM: 03003076

## Εφαρμογή παραλληλοποίησης για την μετατροπή tabular dataset σε 3-D

## ΕΙΣΑΓΩΓΗ

Η ανάγκη προεπεξεργασίας/μετατροπής dataset είναι ένα σημαντικό μέρος της διαδικασίας ανάπτυξης ενός μοντέλου μηχανικής μάθησης (ML) και οι διεργασίες που χρειάζεται να αναπτυχθούν είναι πολλές φορές ιδιαίτερες για κάθε dataset [1], οπότε είναι συχνά αναγκαίο να γίνει βελτιστοποίηση κώδικα για να υπάρχει αποτελεσματική εκμετάλλευση των πόρων του συστήματος. Στην συγκεκριμένη εργασία το θέμα είναι η βελτιστοποίηση κώδικα για την μετατροπή ενός tabular dataset σε 3-D για να χρησιμοποιηθεί ως είσοδος σε convolutional NNs.

Συγκεκριμένα αναφερόμαστε σε ένα tabular dataset το οποίο χρησιμοποιήθηκε για έρευνα σχετική με πρόβλεψη κινδύνου δασικής πυρκαγιάς της επόμενης μέρας στην επικράτεια της Ελλάδας [2]. Το dataset κατασκευάστηκε αρχικά με αυτό τον τρόπο για να χρησιμοποιηθεί για την εκπαίδευση μοντέλων μηχανικής μάθησης (ML) που δέχονται είσοδο δεδομένα 1-D, όπως τα ensemble trees και τα νευρωνικά δίκτυα NN/DNN (χωρίς δλδ. convolutional layers). Όμως για την συνέχιση της έρευνας και των πειραμάτων με προηγμένα μοντέλα που αναγνωρίζουν σχέσεις γειτονικών pixel (όπως π.χ. τα CNNs) γεννήθηκε η ανάγκη μετατροπής του dataset σε 3-D όπου οι δύο διαστάσεις αναφέρονται στην θέση του pixel και η τρίτη στα features.

Το μεγάλο μέγεθος του dataset που ξεπερνά το 1TB, και οι αριθμητικές πράξεις που απαιτούνται για την μετατροπή του dataset μας καθοδήγησαν να εξετάσουμε τις μεθόδους εκείνες που θα εκμεταλλευτούν καλύτερα τους πόρους του συστήματος για την ταχύτερη επεξεργασία. Με το σκεπτικό αυτό προχωρήσαμε στην σύγκριση μεθόδων παραλληλοποίησης του κώδικα με python multiprocessing [3] που παρακάμπτει το Global Interpreter Lock (GIL) [4] της python και με cython parallelism [5] που επιτυγχάνει την παραλληλοποίηση μέσω του cython pre-compiler σε C++ και του OpenMP [6] και επίσης παρακάμπτει το GIL.

Επιπλέον η συγκεκριμένη διαδικασία μετατροπής του dataset κατά την πρακτική εφαρμογή της, κάνει εκτεταμένη χρήση του I/O για ανάγνωση και εγγραφή των αρχείων. Είναι προφανές ότι η συνολική “end-to-end” διαδικασία ανάγνωσης - μετατροπής - εγγραφής δημιουργεί αρκετές “συγκρούσεις” με πολύπλοκους κανόνες στα αιτήματα χρήσης πόρων του συστήματος από τις CPU. Αναζητήσαμε λοιπόν, για αριθμό αρχείων ίσο με τις διαθέσιμες CPU, με πειραματικό τρόπο, την βέλτιστη αναλογία αριθμού παράλληλων threads για την διαδικασία της μετατροπής, που διενεργείται στην μνήμη μόνο και του αριθμού παράλληλων threads για την συνολική διαδικασία της ανάγνωσης-μετατροπής-εγγραφής που κάνει χρήση και των πόρων του storage I/O. Ο σκοπός είναι η βέλτιστη παραμετροποίηση του αριθμού των threads για όλες τις διαδικασίες για την ολοκλήρωση της μετατροπής του dataset στον ταχύτερο δυνατό χρόνο.

## Dataset και Ορισμός προβλήματος

Το tabular dataset περιέχει από μια γραμμή για κάθε ημερολογιακή ημέρα και pixel (κελί) περιοχής ενδιαφέροντος, ενός κανάβου της επικράτειας. Στις στήλες περιέχονται, το ID του pixel και η ημερομηνία που αποτελούν στην πραγματικότητα το indexing των γραμμών και τα 90 χαρακτηριστικά (features) του dataset δλδ παράμετροι μετεωρολογίας, διαμόρφωσης εδάφους δορυφορικοί δείκτες βλάστησης, υγρασίας και θερμοκρασίας εδάφους. Αποτελείται από περίπου 372000 γραμμές (κελιά κανάβου) ανά ημέρα x 1200 ημέρες = 446,4x$10^6$ συνολικά γραμμές. Επειδή η σχετική έρευνα εστιάζει σε δασικές πυρκαγιές, δεν υπάρχουν στο dataset pixels που η θέση τους να είναι σε θάλασσα, νερό αστικές περιοχές κλπ. Επίσης πρέπει να σημειώσουμε ότι το ID του pixel έχει προκύψει από μια διαδικασία επιπεδοποίησης (“flattening”) ενός αρχικού δισδιάστατου κανάβου.

| id | firedate | max_temp | min_temp | mean_temp | res_max | dom_vel | rain_7days | dem | slope | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| 333237 | 20150730 | 0.849671 | 0.804729 | 0.843411 | 0.092412 | 0.138356 | 0.000088 | 0.032146 | 0.030961 | ... |
| 335462 | 20150730 | 0.836843 | 0.806189 | 0.837291 | 0.092412 | 0.127498 | 0.000092 | 0.029680 | 0.029864 | ... |
| 335463 | 20150730 | 0.849671 | 0.804729 | 0.843411 | 0.092412 | 0.138356 | 0.000088 | 0.034410 | 0.027004 | ... |
| 335464 | 20150730 | 0.849671 | 0.804729 | 0.843411 | 0.092412 | 0.138356 | 0.000088 | 0.032146 | 0.030961 | ... |
| 337688 | 20150730 | 0.836843 | 0.806189 | 0.837291 | 0.092412 | 0.127498 | 0.000092 | 0.037847 | 0.035692 | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 2996868 | 20150730 | 0.645476 | 0.903426 | 0.760677 | 0.344471 | 0.349348 | 0.000016 | 0.002148 | 0.015507 | ... |
| 2999095 | 20150730 | 0.645476 | 0.903426 | 0.760677 | 0.344471 | 0.349348 | 0.000016 | 0.003355 | 0.006261 | ... |
| 3001322 | 20150730 | 0.645476 | 0.903426 | 0.760677 | 0.344471 | 0.349348 | 0.000016 | 0.003355 | 0.006261 | ... |
| 3235050 | 20150730 | 0.657105 | 0.882089 | 0.756951 | 0.344471 | 0.217974 | 0.000012 | 0.002214 | 0.000187 | ... |

Εικόνα 1: Δείγμα Tabular dataset μεσα από jupyter notebook. Η στήλη id αναφέρεται στον αριθμό του κελιού (pixel) και η στήλη firedate στην ημερομηνία. Όλες οι άλλες στήλες αποτελούν τα features του dataset

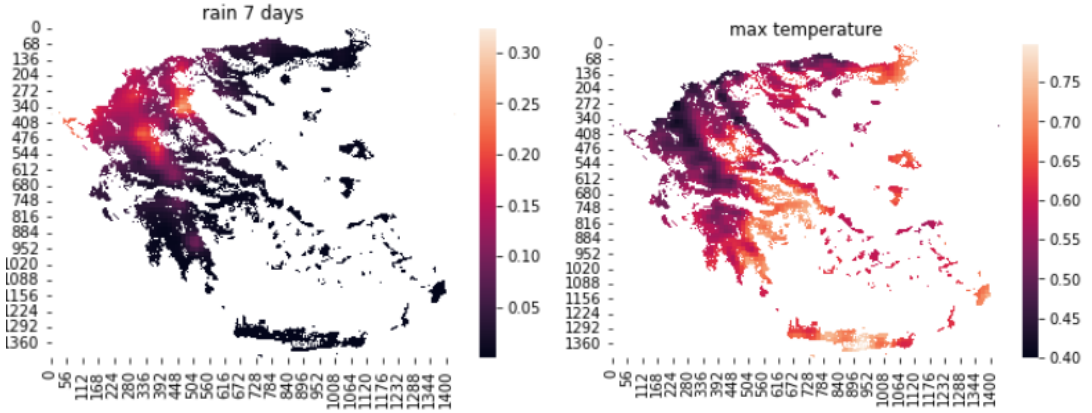
Ο στόχος είναι να μετατραπεί το tabular dataset (Εικόνα 1) σε τρισδιάστατη μορφή (3-D) για να είναι συμβατό ως δεδομένο εισόδου για εκπαίδευση/πρόβλεψη σε convolutional NN μοντέλα μηχανικής μάθησης. Οπότε, το αποτέλεσμα της μετατροπής του, θα είναι ένας 3-D πίνακας (κύβος) για κάθε ημερομηνία όπου οι $x, y$ διαστάσεις θα αντιστοιχούν σε κάθε κελί του κανάβου και η τρίτη διάσταση θα περιέχει όλα τα χαρακτηριστικά (features) του dataset. Επειδή είναι γνωστές οι διαστάσεις (πλάτος, ύψος) του αρχικού κανάβου, και το ID έχει προκύψει από διαδικασία “επιπεδοποίησης” μπορούμε με δύο απλές αριθμητικές πράξεις να υπολογίσουμε την θέση του κελιού $x_i, y_i$ με ID $i$ στον 3-D πίνακα:

$$y_i = int((i - min(i))/w) \ , \quad x_i = i - min(i) - wy_i \quad (1)$$

Όπου $min(i)$ είναι το μικρότερο υπάρχων ID στο tabular dataset που επίσης βρίσκεται στην πρώτη γραμμή του κανάβου και $w$ το πλάτος του κανάβου σε pixel. Τέλος για να ολοκληρωθεί η μετατροπή πρέπει οι τιμές κάθε γραμμής με ID $i$ του tabular dataset να αντιγραφούν (ώς διάνυσμα) στην θέση $x_i, y_i$ του 3-D πίνακα:

$$\boldsymbol{A}_{x_i y_i} = \boldsymbol{z} \ , \ \boldsymbol{z} \in \mathbb{R}^{90} \quad (2)$$

Όπου $\boldsymbol{A}$ ο 3-D πίνακας και $\boldsymbol{z}$ το διάνυσμα τιμών της γραμμής του tabular dataset με ID $i$. Στην Εικόνα 2 προβάλλονται 2 φέτες (slice) του z άξονα του ημερησίου τελικά παραγόμενου 3-D dataset για 2 από τα χαρακτηριστικά του (features)




Εικόνα 2: Δείγμα 3-D dataset μετά την μετατροπή όπου φαίνονται δύο 2-D “slice” συγκεκριμένων feature. Η αριστερή εικόνα δείχνει τις τιμές του feature “rain_7 days” με την μορφή heatmap και η δεξιά το feature “max temperature”. Στον 3-D πίνακα περιέχονται όλα τα features στην 3η διάσταση.

## ΜΕΘΟΔΟΛΟΓΙΑ

### Μετατροπή μεμονωμένου ημερήσιου dataset στην μνήμη

Από τις σχέσεις (1) συμπεραίνουμε ότι η εύρεση των συντεταγμένων του pixel μπορεί να γίνεται ανεξάρτητα για κάθε γραμμή, οπότε αυτό παραπέμπει σε ένα data parallel πρόβλημα [7]. Έτσι υπάρχει η δυνατότητα, διαιρώντας το dataset σε κομμάτια που να περιέχουν έναν κατά προτίμηση ίσο αριθμό γραμμών, να εκτελείται παράλληλα η ίδια διαδικασία, για όσα από τα κομμάτια μπορούν να εξυπηρετήσουν ταυτόχρονα οι CPU του συστήματος. Επίσης, από την σχέση (2) φαίνεται ότι για την τελική απόδοση των τιμών πρέπει να υπάρχει μια διαμοιραζόμενη μνήμη που θα αντιστοιχεί στον τελικο 3-D πίνακα. Αυτή η απόδοση τιμών αναμένεται βέβαια να προκαλέσει συγκρούσεις στα αιτήματα πρόσβασης στην μνήμη.

Με σκοπό την σύγκριση και την επιλογή της καλύτερης μεθόδου παραλληλοποίησης δοκιμάζονται μερικές από τις αρκετές διαφορετικές τεχνικές που μπορούν να αναπτυχθούν στο περιβάλλον python. Οι μέθοδοι παραλληλοποίησης μέσω python για τις οποίες υπάρχει δυνατότητα και ενδιαφέρον να χρησιμοποιηθούν για αυτή την εφαρμογή είναι η βιβλιοθήκη multiprocessing της python [3] που παρακάμπτει το Global Interpreter Lock (GIL) [4] και οι δυνατότητες παραλληλοποίησης μέσω cython [5] που είναι ένας pre-compiler που παράγει κώδικα C++ και υποστηρίζει OpenMP [6]. Ο κώδικας cython μετατρέπεται σε C++ μέσω του αντίστοιχου pre-compiler και στην συνέχεια μέσω του compiler C++ γίνεται βιβλιοθήκη κώδικα μηχανής της οποίας οι συναρτήσεις μπορούν να εκτελεστούν μέσα από την python με την ταχύτητα προφανώς compiled κώδικα (και όχι interpreter). Άλλες γνωστές βιβλιοθήκες παραλληλοποίησης είναι η ενσωματωμένη στην python threading [8] που όμως δεν παρακάμπτει το GIL με αποτέλεσμα αρκετά χαμηλή απόδοση, η βιβλιοθήκη dask [9] που ειδικεύεται μεν στην παράλληλη επεξεργασία πινάκων, όμως δεν μπορεί να υποστηρίξει περίπλοκη μετατροπή των πινάκων όπως την περιγράφουμε για την συγκεκριμένη εφαρμογή και τέλος η Numba [10] που έχει αντίστοιχες δυνατότητες με την cython καθώς παράγει compiled κώδικα, αλλά οι διαδικασίες του pre-compiling / compiling γίνονται με

αδιάφανο τρόπο για τον χρήστη στον οποίο δίνονται περιορισμένες δυνατότητες παραμετροποίησης.
Ο γενικός αλγόριθμος που εκτελούμε στο κάθε πείραμα είναι :

1. DS ← assign tabular dataset of a certain date
2. w ← width of grid, min_id ← min pixel ID //known values
3. A ← empty 3-D array //in shared memory for parallel experiments
4. For each line z in DS with ID i { //parallel execution in parallel experiments
    5. Row ← int(i-min_id)/w
    6. Col ← i - min_id -w*Row
    7. A(Col, Row) ← z }

Το block των γραμμών 5-7 εκτελείται προφανώς παράλληλα στα πειράματα όπου χρησιμοποιούνται περισσότερες cpu μοιράζοντας κομμάτια (σύνολα γραμμών) του dataset σε διαφορετικά threads.
Ο κώδικας για όλα τα πειράματα είναι διαθέσιμος στο github link: https://github.com/AlexApostolakis/par_dataset_conv. Τα βασικά αρχεία είναι το jupyter notebook par_conv_tab_3D.ipynb όπου εκτελούνται τα περισσότερα πειράματα και δημιουργούνται τα γραφήματα απόδοσης, το create2Dpar.pyx όπου υλοποιείται ο κώδικας cython και το tab_2_3D.py που εκτελούνται τα πειράματα για την end-to-end διαδικασία με ανάγνωση και εγγραφή αρχείων. Παρακάτω παρουσιάζονται σύντομα οι μέθοδοι υλοποίησης που συγκρίνονται για την απόδοσή τους μέσα από διαγράμματα speed-up και wall time [7].

α) single threaded python loop
Το πείραμα αυτό είναι πολύ απλό καθώς υλοποιείται με απλό σειριακό loop που διατρέχει το tabular dataset και γεμίζει τον 3-D πίνακα. Η υλοποίηση του είναι κάτω από το markdown κελί “Simple python run” στο jupyter notebook par_conv_tab_3D.ipynb

β) python multiprocessing με shared memory
Για το πείραμα αυτό επιστρατεύεται η βιβλιοθήκη της python multiprocessing που δίνει την δυνατότητα παράλληλης εκτέλεσης διαδικασιών με την εντολή Process και την δυνατότητα χρήσης διαμοιραζόμενης μνήμης. Η υλοποίηση είναι επίσης στο jupyter notebook par_conv_tab_3D.ipynb κάτω από markdown κελί “multiprocessing with shared memory”

γ) cython prange through OpenMP
Για το πείραμα αυτό υλοποιούμε την συνάρτηση της μετατροπής του dataset σε cython. Το πρόγραμμα cython γίνεται pre-compiled σε C++ και στην συνέχεια γίνεται κανονικά compiled από τον C++ compiler σε shared library της οποίας οι συναρτήσεις μπορούν να εκτελεστούν μέσα από κώδικα python ως εκτελέσιμος κώδικας παρακάμπτοντας δλδ. τον python interpreter. Η cython υποστηρίζει την χρήση της OpenMP μέσα από διάφορες εντολές. Σε αυτή την συνάρτηση χρησιμοποιούμε την ισχυρή εντολή `prange` [5] (parallel range δλδ) που εκτελεί παράλληλα τις εντολές του loop που βρίσκονται στο block του κώδικα που ανοίγει η εντολή. Η `prange to` παίρνει ως όρισμα το schedule (του OpenMP) που είναι ο τρόπος που θα διαμοιραστούν οι επαναλήψεις στα threads και το chunk size που είναι ο αριθμός των επαναλήψεων του loop που θα εκτελέσει σειριακά σε κάθε thread (έχει νόημα για ορισμένα schedules π.χ. static). Η συνάρτηση cython βρίσκεται στο αρχείο create2Dpar.pyx :

```
def fillcube(int nt, cnp.ndarray[FLOAT32_t, ndim=2] tab, long firstid, long rdiff,
gW, gH, sched=None, chunks='auto', intensive=0)
```

Η παράμετρος `nt` είναι ο αριθμός των threads, tab το tabular dataset, τα `firstid, rdiff, gW,gH` είναι χαρακτηριστικά του κανάβου για την μετατροπη, `sched` το schedule, `chunks` το chunk size, και `intensive` είναι μια παράμετρος για αύξηση της δυσκολίας των υπολογισμών και της αντίστοιχης απασχόλησης της CPU που χρησιμοποιούμε σε ένα πείραμα για να δείξουμε τις δυνατότητες παραλληλοποίησης. Η εκτέλεση του πειράματος με την χρήση της συνάρτησης αυτής βρίσκεται στο jupyter notebook par_conv_tab_3D.ipynb κάτω από το markdown κελί “cython with shared memory”. Πρέπει να σημειωθεί εδώ ότι έγιναν πειράματα με διάφορα schedules και chunk size, χωρίς να παρατηρηθεί ιδιαίτερη διαφορά εκτός βέβαια από την περίπτωση που δινόταν chunk size όχι μοιρασμένο σύμφωνα με τον αριθμό των threads, όπως και αναμενόταν. Τελικά τα πειράματα διενεργήθηκαν με schedule ‘static’ και chunk size μοιρασμένο στον αριθμό των threads.

### Βελτιστοποίηση ολοκληρωμένης διαδικασίας μετατροπής dataset (read-convert-write)

Στην πράξη αυτό που τελικά μας ενδιαφέρει είναι η end-to-end απόδοση της όλης διαδικασίας. Δηλαδή όχι απλά ο χρόνος μετατροπής ενός ημερήσιου dataset στην μνήμη, αλλά η συνολική απόδοση της ανάγνωσης-μετατροπής-εγγραφής, που περιλαμβάνει και μεγάλο μέρος εντολών storage I/O. Περιμένουμε στην ολοκληρωμένη διαδικασία να έχουμε πολλές συγκρούσεις και αναμονές λόγω των αιτημάτων/διακοπών του λειτουργικού για storage I/O. Για να βελτιστοποιήσουμε λοιπόν την διαδικασία αυτή, σχεδιάζουμε ένα πείραμα, το οποίο περιλαμβάνει την μετατροπή nCPU αρχείων, όσες δλδ και οι cpu που διαθέτει το σύστημα. Στο πείραμα αυτό θα μετρήσουμε την απόδοση εκτελώντας την “end-to-end” διαδικασία με παραλληλοποίηση για μεταβλητό αριθμό threads αντίστοιχο με τα ημερήσια αρχεία (1-nCPU) και επίσης η μετατροπή του dataset στην μνήμη που περιγράψαμε προηγουμένως θα εκτελείται επίσης με παραλληλοποίηση με μεταβλητό αριθμό threads (1-nCPU). Ο στόχος του πειράματος είναι να βρούμε τον καλύτερο συνδυασμό του αριθμού threads που θα αποδίδεται στην end-to-end διαδικασία και του αριθμού threads που θα αποδίδεται στην διαδικασία μετατροπής μόνο. Ο αλγόριθμος για αυτό το πείραμα είναι ο παρακάτω:

1. For each daily dataset (in n files): { //execute parallel n threads (n:1-nCPU)
    2. DS ← Read tabular dataset from storage
    3. w ← width of grid, min_id ← min pixel ID
    4. A ← empty 3-D array //in shared memory
    5. For each line z in DS with ID i { //execute parallel m threads (m:1-nCPU)
        6. Row ← int(i-min_id)/w
        7. Col ← i - min_id -w*Row
        8. A(Col, Row) ← z }
    9. Write 3-D dataset to storage}

Ο αλγόριθμος αυτός εκτελείται στον κώδικα μέσω των συναρτήσεων:

```
creategrid_xs_small(rdiff, firstid, gridwidth, gridheight, dayfile, pcpus, ccpus, queue)
create_xs_files(creategrid, days, pthreads, cthreads)
```

```
dataset_conv_opt(maxcpus, dayfiles, nfiles)
```

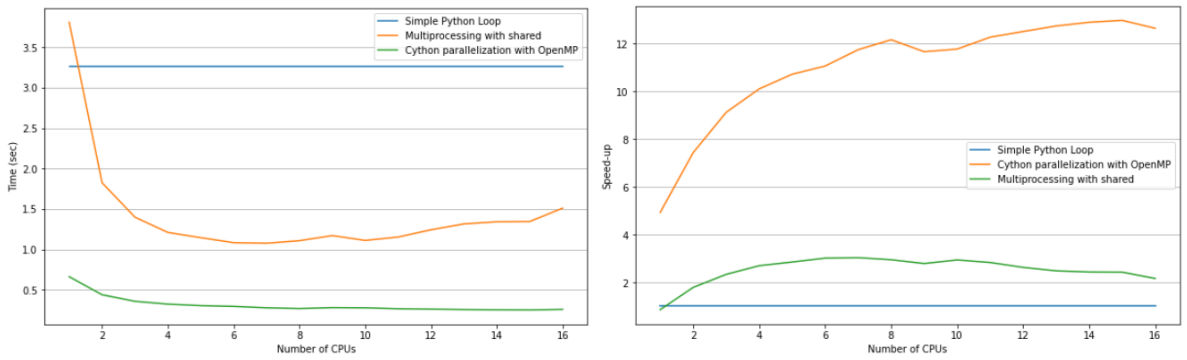
και βρίσκεται στο αρχείο python tab_2_3D.py. Για την παράλληλη εκτέλεση της “end-to-end” διαδικασίας επιστρατεύεται η παραλληλοποίηση μέσω python multiprocessing και για την παραλληλοποίηση της μετατροπής του dataset χρησιμοποιείται η παραλληλοποίηση μέσω της συνάρτησης cython που έχει και την καλύτερη απόδοση στα αντίστοιχα πειράματα της μετατροπής του dataset στην μνήμη.

## ΑΠΟΤΕΛΕΣΜΑΤΑ

Όλα τα πειράματα που αφορούν τα αποτελέσματα έχουν εκτελεστεί σε σύστημα virtual machine με 16 CPU και επαρκή μνήμη ώστε να εκτελεί την μετατροπή ενός ημερήσιου dataset στην μνήμη.

### Σύγκριση μεθόδων single threaded/parallel python και cython

Για το πείραμα α) το απλό single threaded python loop δλδ ο χρόνος επεξεργασίας είναι περίπου στα 3,3 second, για το πείραμα β) python multiprocessing με shared memory το αποτέλεσμα ξεκινά από τα 3,8 sec περίπου με μία cpu, φτάνει την καλύτερη απόδοση με 6 cpu (1,14 sec) και μετά αυξάνεται σταδιακά μέχρι περίπου 1,5 sec με 16 cpu. Τέλος, για το πείραμα γ) cython prange με OpenMP έχουμε σαφέστατα την καλύτερη απόδοση καθώς ο χρόνος με μια cpu είναι κοντά στα 0,6 sec σταθεροποιείται στα 0,3 περίπου sec μετά από 5 cpu και πάνω (Εικόνα 3 αριστερά).



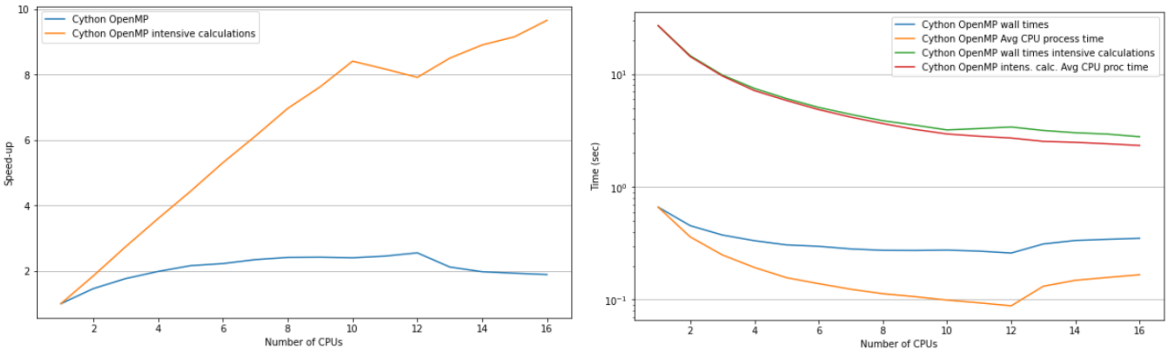

Εικόνα 3: Αριστερά: Ο χρόνος επεξεργασίας (wall time) για κάθε πείραμα. Δεξιά: το Speed-up των συγκρινόμενων μεθόδων. Για να υπολογιστεί το speed-up έχει οριστεί ως $T_{seq}$ ο χρόνος του σειριακού πειράματος (α)

Για το speed-up, αν θεωρήσουμε ως $T_{seq}$ (δλδ. χρόνο του σειριακού αλγορίθμου [7]) τον χρόνο του πειράματος α) βλέπουμε στην Εικόνα 3 δεξιά ότι στο πείραμα β) το speed-up φτάνει κοντά στο 3 με 6 cpu και αυξάνοντας τον αριθμό των cpu υποχωρεί σταδιακά μέχρι το 2, ενώ για το πείραμα γ) ξεκινάει από 5 με μια μόνο cpu καθώς επωφελείται προφανώς από τον compiled κώδικα C++ και σταθεροποιείται κοντά στο 12-13 από 8 cpu και πάνω. Βέβαια αν σαν $T_{seq}$ στο πείραμα γ) ορίσουμε τον χρόνο επεξεργασίας με μια cpu στο ίδιο πείραμα, προφανώς τότε το μέγιστο speed-up θα είναι περίπου ίσο με 2,5 από τις 8 cpu και πάνω. Ο βασικός λόγος που το speed-up δεν ανεβαίνει ιδιαίτερα με την αύξηση των cpu είναι από την μία ότι οι αριθμητικές πράξεις που εκτελούνται (βλ. σχέσεις (1), βήματα πρώτου αλγορίθμου 5,6) είναι απλές και από την άλλη ότι οι εντολές ανάθεσης τιμών (βλ. σχέση (2), βήμα πρώτου αλγορίθμου 7) είναι αρκετά απαιτητικές σε όγκο, οπότε δημιουργούνται

συγκρούσεις στα αιτήματα εγγραφής στην μνήμη που προκαλούν και σημαντικούς χρόνους αναμονής (idle time) για τις cpu.

### Πείραμα με εντατικοποίηση υπολογισμών

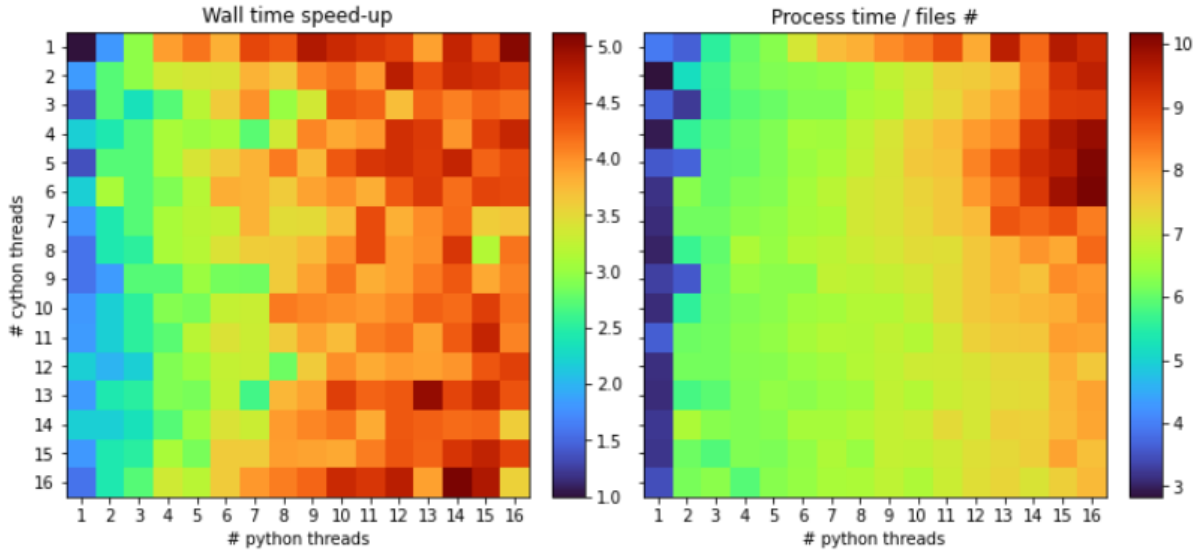
Για να δείξουμε ότι ο ισχυρισμός για τους χρόνους αναμονής των cpu λόγω αναθέσεων στην μνήμη ισχύει, εντατικοποιούμε τεχνητά τα βήματα αριθμητικών υπολογισμών προσθέτοντας πράξεις πολύπλοκων συναρτήσεων και εκτελούμε πάλι το πείραμα γ) με την cython. Η συγκριτική απόδοση τώρα του πειράματος με τα “intensive calculations” μας δείχνει ότι το speed-up (θέτοντας ως $T_{seq}$ την εκτέλεση με μια cpu) ανεβαίνει έως και 10 ενώ το πείραμα με τους απλούς υπολογισμούς δεν ξεπερνά το 2,5 (Εικόνα 4 αριστερά) όπως είχαμε εντοπίσει και στην προηγούμενη παράγραφο. Επίσης στο λογαριθμικό γράφημα της Εικόνας 4 δεξιά βλέπουμε ότι για το πείραμα με τα “intensive calculations” ο συνολικός χρόνος επεξεργασίας (wall time) συνεχίζεται να μειώνεται μέχρι και την χρήση των 16 cpu. Επίσης στο ίδιο γράφημα παρατηρούμε ότι ο καθαρός μέσος χρόνος επεξεργασίας ανά cpu (κόκκινη γραμμή) ακολουθεί πολύ κοντά την καμπύλη του wall time (πράσινη γραμμή), πράγμα που σημαίνει ότι υπάρχει πολύ καλύτερη εκμετάλλευση των cpu σε αντίθεση με το αρχικό πείραμα, όπου οι καμπύλες wall time και καθαρού χρόνου επεξεργασίας (μπλέ και πορτοκαλί αντίστοιχα) ανά cpu απομακρύνονται αρκετά.




Εικόνα 4: Αριστερά: τα συγκριτικά speed-up για το πείραμα γ) με τους απλούς υπολογισμούς και εντατικοποιημένους. Δεξιά: Γράφημα λογαριθμικής κλίμακας με τον χρόνο επεξεργασίας (wall time) για απλούς και εντατικοποιημένους υπολογισμούς και των αντίστοιχων μέσων καθαρών χρόνων επεξεργασίας ανά CPU.

### Πείραμα για την βελτιστοποίηση της “end-to-end” διαδικασίας

Στο πείραμα αυτό μας ενδιαφέρει κυρίως, όπως αναφέρθηκε στο κεφάλαιο για την μεθοδολογία, να βρεθεί ο καλύτερος συνδυασμός του αριθμού των threads για την παράλληλη εκτέλεση της end-to-end διαδικασίας (read-convert-write) και του αριθμού threads για την διαδικασία convert μόνο. Για τον σκοπό αυτό δημιουργούμε heatmap διαγράμματα που παρουσιάζουν με χρωματική κλίμακα τις τιμές wall time speed-up και process time από τους συνδυασμούς των threads για τις δύο διαδικασίες, αφού προηγουμένως έχουμε σώσει τις αντίστοιχες τιμές σε κατάλληλο πίνακα κατά την διάρκεια του πειράματος. Για λόγους συντομίας αλλά και προέλευσης του κώδικα ονομάζουμε ως cython threads τα threads που αναφέρονται μόνο στην μετατροπή του ημερήσιου dataset στην μνήμη και python threads τα threads που αναφέρονται στην end-to-end διαδικασία για κάθε αρχείο που περιλαμβάνει και το read-write




Εικόνα 5: Αριστερά: wall time speed up για κάθε συνδυασμό cython, python threads. Δεξιά: Καθαρό CPU process time ανά αρχείο για κάθε συνδυασμό cython, python threads.

Στην Εικόνα 5 αριστερά παρατηρούμε ότι το speed-up είναι μεγαλύτερο όσο αυξάνονται τα python threads χωρίς να επηρεάζεται η διαδικασία σε μεγάλο βαθμό από τον αριθμό των cython threads. Αυτό είναι αναμενόμενο καθώς τα cython threads αναλαμβάνουν την μετατροπή στην μνήμη που διαρκεί λιγότερο από ένα second και έχουμε δει ότι το μέγιστο speed-up που επιτυγχάνεται είναι περίπου 2,5. Από την άλλη τα python threads παραλληλοποιούν και τις διαδικασίες read write οι οποίες ξεπερνούν το 90% του χρόνου της συνολικής επεξεργασίας. Δεξιά στην Εικόνα 5 όπου αποτυπώνεται το καθαρό μέσο CPU process time ανά αριθμό αρχείων, παρατηρούμε ότι υπάρχει μια περιοχή πάνω δεξιά όπου έχουμε την μεγαλύτερη απασχόληση των cpu ανά αρχείο. Μπορούμε να υποθέσουμε ότι όσο μεγαλύτερη είναι η τιμή αυτή τόσο μικρότερο είναι cpu wait time και ενδεχομένως σε αυτούς τους συνδυασμούς να γίνεται τέτοιος συγχρονισμός ώστε να υπάρχουν λιγότερες “συγκρούσεις” αιτημάτων χρήσης I/O storage και μνήμης.
Για να καταλήξουμε λοιπόν σε έναν βέλτιστο συνδυασμό, εφαρμόζουμε την εξής λογική: Παίρνουμε τον συνδυασμό cython / python threads με το μέγιστο Process Time / file από τους συνδυασμούς όμως που ξεπερνούν το κατώφλι 90% του μέγιστου speed-up. Δλδ.

$$\boldsymbol{B} = \left[b_k\right] = \{\overline{[i,j]} : \; i,j \; \forall \; S_{ij} \; > \; 0.9max(\boldsymbol{S})\}$$

$$\boldsymbol{N}_{ct}, \boldsymbol{N}_{pt} = \; argmax(p_{i,j}), \; \overline{[i,\; j]} \; \epsilon \; \boldsymbol{B}$$

όπου S ο πινακας των speed-up, B πίνακας με τους συνδυασμούς cython, python threads με speed-up μεγαλύτερο από το 90% του μέγιστου speed-up, [$p_{ij}$] ο πίνακας των cpu process times / file και $\boldsymbol{N}_{ct}$, $\boldsymbol{N}_{pt}$ ο καλύτερος συνδυασμός cython, python threads. Συγκεκριμένα το αποτέλεσμα των παραπάνω πράξεων πάνω στα αποτελέσματα του πειράματος δίνει $\boldsymbol{N}_{ct} = 4$ και $\boldsymbol{N}_{pt} = 16$. Ο συνδυασμός αυτός είναι ένα τετράγωνο από την πιο σκούρα κόκκινη περιοχή στο heatmap δεξιά στην Εικόνα 5 με τα process time και ταυτόχρονα δίνει speed-up κοντά στο 4,7. Ο κώδικας για την εκτέλεση των πειραμάτων και η καταγραφή της απόδοσης είναι στο αρχειο tab_2_3D.py και ο κώδικας για την προβολή των γραφημάτων heatmap και της επιλογής του καλύτερου συνδυασμού είναι στο jupyter notebook par_conv_tab_3D.ipynb κάτω από το markdown κελί “end-to-end dataset conversion performance”

## ΣΥΜΠΕΡΑΣΜΑ

Η προεπεξεργασία των συνόλων δεδομένων αποτελεί μεγάλο μέρος του data science και τα υπάρχοντα εργαλεία, βιβλιοθήκες και συναρτήσεις, δεν αρκούν πάντα για να υλοποιηθεί εύκολα μια λύση που να παρέχει βέλτιστη απόδοση μέσω παραλληλοποίησης. Είναι πολλές φορές αναγκαίο να αναπτύσσεται κώδικας σε χαμηλότερο επίπεδο από έτοιμες συναρτήσεις και βιβλιοθήκες έχοντας καλή κατανόηση των τρόπων εκμετάλλευσης των πόρων του συστήματος μέσω παραλληλοποίησης. Με την ανάπτυξη συγκεκριμένης μεθοδολογίας και αντίστοιχου κώδικα στην παρούσα εργασία δείξαμε ότι υπάρχει δυνατότητα επιτάχυνσης στην παραγωγή του dataset κατά τουλάχιστον 5 φορές με την εφαρμογή παραλληλοποίησης. Μελλοντικά, θα μπορούσε η περίπτωση αυτή μετατροπής, να γενικευτεί και να ενσωματωθεί σε κάποια εργαλεία παραλληλοποίησης όπως π.χ. η βιβλιοθήκη dask που εστιάζει στην επεξεργασία πινάκων με παραλληλοποίηση. Παρόλα αυτά επειδή η προεπεξεργασία έχει συνήθως να αντιμετωπίσει πολυποίκιλες δομές και μορφές δεδομένων, φαίνεται ότι η ανάγκη να αναπτύσσονται κατά περίπτωση βέλτιστες διαδικασίες και λογισμικό για για την προεπεξεργασία μεγάλων δεδομένων που δεν αντιμετωπίζεται άμεσα από έτοιμα εργαλεία θα παραμείνει αρκετό καιρό ακόμα στο προσκήνιο.

### Βιβλιογραφία


[1] A. Paleyes, R.-G. Urma, and N. D. Lawrence, “Challenges in Deploying Machine Learning: a Survey of Case Studies,” *ACM Comput. Surv.*, vol. 55, no. 6, pp. 1–29, Jul. 2023, doi: 10.1145/3533378.

[2] A. Apostolakis, S. Girtsou, G. Giannopoulos, N. S. Bartsotas, and C. Kontoes, “Estimating Next Day’s Forest Fire Risk via a Complete Machine Learning Methodology,” *Remote Sens.*, vol. 14, no. 5, p. 1222, Mar. 2022, doi: 10.3390/rs14051222.

[3] “multiprocessing — Process-based parallelism — Python 3.8.3 documentation.” [Online]. Available: https://docs.python.org/3/library/multiprocessing.html

[4] “GlobalInterpreterLock - Python Wiki.” https://wiki.python.org/moin/GlobalInterpreterLock (accessed Mar. 06, 2023).

[5] “Using Parallelism — Cython 3.0.0b1 documentation.” https://cython.readthedocs.io/en/latest/src/userguide/parallelism.html (accessed Mar. 06, 2023).

[6] tim.lewis, “Home,” *OpenMP*. https://www.openmp.org/ (accessed Mar. 06, 2023).

[7] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, “A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures,” *Commun. Comput. Phys.*, vol. 15, no. 2, pp. 285–329, Feb. 2014, doi: 10.4208/cicp.110113.010813a.

[8] “threading — Thread-based parallelism — Python 3.8.2 documentation.” [Online]. Available: https://docs.python.org/3/library/threading.html

[9] “Dask — Dask documentation.” https://docs.dask.org/en/stable/ (accessed Mar. 08, 2023).

[10] “Numba: A High Performance Python Compiler.” https://numba.pydata.org/ (accessed Mar. 08, 2023).