ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΗΣ ΕΛΛΑΔΟΣ

ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ

“ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ”

Θέμα: «Ανάπτυξη εργαστηριακού οδηγού Παράλληλου Προγραμματισμού με τη γλώσσα προγραμματισμού Python».

Υπεύθυνος καθηγητής: ΒΑΡΣΑΜΗΣ ΔΗΜΗΤΡΗΣ

Διδάσκων καθηγητής: ΒΑΡΣΑΜΗΣ ΔΗΜΗΤΡΗΣ

Φοιτητής: Γεωργιάδης Μεθόδιος - Χρήστος (221)

ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ 2023

Πίνακας περιεχομένων

Εισαγωγή

Στην εποχή της ψηφιακής επανάστασης, η ανάγκη για αποτελεσματική εκτέλεση υπολογιστικών εργασιών καθίσταται επιτακτική. Η εκμετάλλευση της υπολογιστικής ισχύος απαιτεί συνεχή εξέλιξη τεχνικών και προσεγγίσεων. Στο πλαίσιο αυτό, ο παράλληλος προγραμματισμός εμφανίζεται ως κρίσιμο κομμάτι για την αξιοποίηση της δυναμικής των υπολογιστικών συστημάτων.

Αναλυτικότερα στον κόσμο των υπολογιστών, οι διεργασίες και τα νήματα αποτελούν τα βασικά στοιχεία του παράλληλου προγραμματισμού. Μια διεργασία αναπαριστά ένα ανεξάρτητο πρόγραμμα που εκτελείται σε δικό της χώρο διευθύνσεων μνήμης, ενώ τα νήματα μοιράζονται τον ίδιο χώρο διευθύνσεων. Η εκτέλεση των εργασιών στην CPU μπορεί να γίνει με διάφορους τρόπους. Η διαδικασία αυτή μπορεί να διακριθεί σε τρεις βασικές κατηγορίες εκτέλεσης: σειριακή, συντρέχουσα, και παράλληλη. Καθένας από αυτούς τους τρόπους παρουσιάζει διαφορετικά πλεονεκτήματα και μειονεκτήματα, που εξαρτώνται από τη φύση των εργασιών και τις απαιτήσεις της εφαρμογής. Η παραλληλοποίηση μπορεί να επιτευχθεί με τη χρήση νημάτων και διεργασιών. Πιο συγκεκριμένα τα νήματα βελτιστοποιούν τον χρόνο για εργασίες εισόδου – εξόδου, ενώ οι διεργασίες για εργασίες μεγάλων υπολογιστικών πράξεων. Συνολικά, ο παράλληλος προγραμματισμός ανοίγει νέους ορίζοντες για την απόδοση των εφαρμογών, ενισχύοντας την ικανότητα εκμετάλλευσης των πόρων της CPU και επιτυγχάνοντας την παράλληλη εκτέλεση εργασιών για βέλτιστα αποτελέσματα.

ΚΕΦΑΛΑΙΟ 1

Python

Η Python αποτελεί μια εξαιρετικά προσβάσιμη γλώσσα προγραμματισμού, καθώς επιτρέπει στους χρήστες να αναπτύξουν εφαρμογές χωρίς την ανάγκη να εμβαθύνουν σε υπερβολικό βαθμό. Απολαμβάνει ευρεία αποδοχή στον χώρο της ανάπτυξης εμπορικών εφαρμογών, καθώς γίνεται ολοένα περισσότερο αναγνωρίσιμη και εκτιμημένη.

Κύριος στόχος της Python είναι η αναγνωσιμότητα του κώδικα και η απλότητα χρήσης, χαρακτηριστικά που συντελούν στην ευκολία και την ταχύτητα εκμάθησής της. Η γλώσσα αυτή διαθέτει πληθώρα βιβλιοθηκών που διευκολύνουν τις καθημερινές εργασίες των προγραμματιστών.

Αρχικά σχεδιασμένη ως γλώσσα σεναρίων για το λειτουργικό σύστημα Amoeba, η Python έχει εξελιχθεί διαρκώς. Η έκδοση 3.0, που κυκλοφόρησε το 2008, σηματοδότησε μια σημαντική αλλαγή μεταξύ των εκδόσεών της, διακόπτοντας την προς τα πίσω συμβατότητα με προηγούμενες εκδόσεις.

Ένα αξιοσημείωτο χαρακτηριστικό της Python είναι η ανοικτή της φύση, ως γλώσσα ανοικτού κώδικα. Αυτό σημαίνει ότι ο κώδικάς της είναι διαθέσιμος δωρεάν σε όλους, ενθαρρύνοντας τη συνεισφορά και την προσαρμογή από την κοινότητα προγραμματιστών. Η Python αποτελεί σημαντικό πυρήνα στον κόσμο του ανοικτού λογισμικού, με συνεχή ανάπτυξη και αποδοτική συνεργασία στο διαδίκτυο.

Τέλος, η Python επωφελείται από τον μεγάλο αριθμό βιβλιοθηκών που είναι διαθέσιμες, συνιστώντας έναν εύκολα προσβάσιμο οικοσύστημα για τους προγραμματιστές. Αυτό συμβάλλει στην ευελιξία και την αποτελεσματικότητα της γλώσσας, καθιστώντας την επιλογή προτίμησης σε πολλούς τομείς και κοινότητες προγραμματιστών.

ΚΕΦΑΛΑΙΟ 2

Διεργασία

Διεργασία είναι ένα στιγμιότυπο ενός προγράμματος που εκτελείται σε έναν υπολογιστή. Πρόγραμμα ορίζεται ένα στατικό σύνολο εντολών. Η διεργασία συνιστά την εκτέλεση αυτών των εντολών και μπορεί να δημιουργηθεί από άλλη διεργασία κατά την εκτέλεσή της. Κάθε διεργασία έχει το δικό της χώρο μνήμης και περιλαμβάνει:

- Ένα τμήμα εντολών προγράμματος (κώδικα).
- Ένα ή τμήμα δεδομένων.
- Ένα σετ καταχωρητών της CPU.
- Μία στοίβα.
- Διαφόρους πόρους του λειτουργικού συστήματος (π.χ. αρχεία, συσκευές I/O κ.α.).

Έστω ότι το λειτουργικό σύστημα δημιουργεί δυο διεργασίες κάθε διεργασία έχει ένα τμήμα κώδικα και δεδομένα. Οι διεργασίες βλέπουν μόνο το χώρο που τους έδωσε το λειτουργικό σύστημα, τίποτα άλλο. Αν θέλει η μία διεργασία να επικοινωνήσει με την άλλη, δεν μπορεί να το κάνει άμεσα. Θα πρέπει να γίνει μέσω του λειτουργικού συστήματος, οπότε το λειτουργικό σύστημα δημιουργεί ένα ψευτοκοινόχρηστο χώρο στη μνήμη. Έτσι ώστε όταν μια διεργασία θέλει να στείλει κάτι στην άλλη, το ζητάει από το λειτουργικό σύστημα. Τότε το στέλνει στην ψευτοκοινόχρηστη μνήμη και από εκεί διαβάζει η άλλη διεργασία τις πληροφορίες

που θέλει. Οπότε δεν υπάρχει περίπτωση να γράψετε πρόγραμμα που θα τρέχει σε παράλληλες διεργασίες και να ορίζετε μια μεταβλητή που θα γράφουν κατευθείαν και οι δυο διεργασίες. Γι' αυτό λεμέ ότι έχουμε κατανεμημένη μνήμη.

Η κατανεμημένη μνήμη (Distributed Memory) αναφέρεται σε ένα σύστημα όπου η μνήμη δεν είναι κοινόχρηστη ανάμεσα σε διάφορες διεργασίες. Κάθε διεργασία έχει τη δική της μνήμη και δεν μπορεί να αποκτήσει άμεση πρόσβαση στη μνήμη άλλων διεργασιών. Η επικοινωνία μεταξύ των διεργασιών γίνεται μέσω μηχανισμών επικοινωνίας όπως η ουρά.

Νήματα

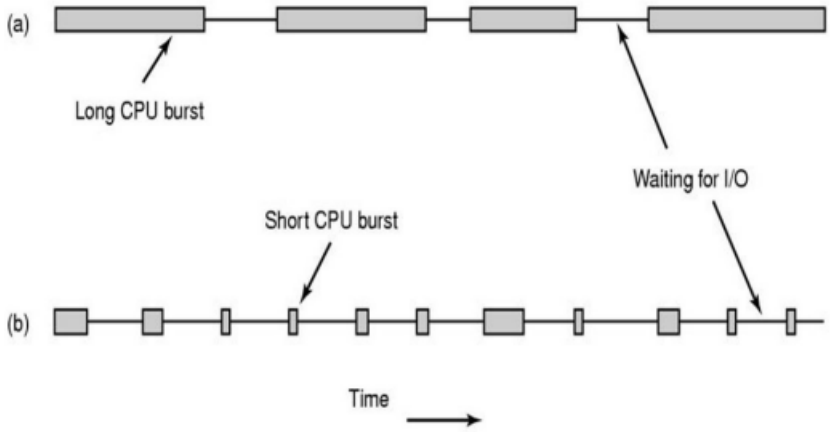
Τα νήματα δημιουργούνται από τις διεργασίες και είναι ακολουθίες εντολών τις οποίες διαχειρίζεται ανεξάρτητα το λειτουργικό σύστημα. Κάθε διεργασία περιλαμβάνει ένα τουλάχιστον νήμα το οποίο εκτελείται κατά την έναρξή της. Τα νήματα που ανήκουν στην ίδια διεργασία μοιράζονται τα τμήματα κώδικα και δεδομένων καθώς και τους λοιπούς πόρους που έχουν εκχωρηθεί στην διεργασία και λαμβάνουν ξεχωριστά:

- Ένα μοναδικό αναγνωριστικό (thread id).
- Έναν μετρητή προγράμματος (δείκτη στην επόμενη εντολή που θα εκτελεστεί).
- Ένα σύνολο καταχωρητών της CPU.
- Μία στοίβα (stack).

Όταν θα σπάσουμε τον κώδικα σε νήματα, κάθε νήμα θα πάρει κάποιους δικούς του καταχωρητές και στοίβα για να διαχειριστεί την εκτέλεση του. Επίσης θα έχει το ίδιο τμήμα κώδικα, δεδομένων και πόρων με την αρχική διεργασία. Όλα τα νήματα βλέπουν το τμήμα δεδομένων (data) για αυτό μπορούμε μια μεταβλητή να την προσπελάσουμε είτε από το ένα νήμα είτε από το άλλο γι' αυτό λεμέ ότι έχουμε κοινόχρηστη μνήμη. Εδώ όμως έχουμε το πρόβλημα του συγχρονισμού.

Ο συγχρονισμός αφορά τον τρόπο με τον οποίο τα νήματα συντονίζονται μεταξύ τους όταν προσπελάζουν και τροποποιούν κοινόχρηστες μεταβλητές. Υπάρχουν διάφοροι μηχανισμοί συγχρονισμού που μπορείτε να χρησιμοποιήσετε για να αντιμετωπίσετε το πρόβλημα του συγχρονισμού σε πολυνηματικές εφαρμογές στη γλώσσα προγραμματισμού Python, όπως τα κλειδώματα (Locks). Τα κλειδώματα είναι μηχανισμοί που επιτρέπουν σε ένα νήμα να αποκλείσει άλλα νήματα από την πρόσβαση σε ένα κοινόχρηστο τμήμα κώδικα.

Συνοψίζοντας τον κώδικα το χωρίζουμε σε νήματα. Τα νήματα μπορούν να εκτελεστούν είτε με ξεχωριστές διεργασίες, είτε στην ίδια. Αν εκτελεστούν σε ξεχωριστές διεργασίες, δεν μπορούμε να έχουμε κοινόχρηστη μνήμη. Πρέπει να γίνει επικοινωνία με έναν μηχανισμό ανταλλαγής μηνυμάτων .Αν εκτελεστούν σε νήματα μπορεί πάλι να εκτελεστούν σε διαφορετικούς επεξεργαστές, αλλά δεν θα δημιουργηθούν ξεχωριστές διεργασίες. Σε αυτή την περίπτωση τα νήματα μπορούν να δουν το ίδιο τμήμα δεδομένων και κώδικα.



(a) CPU-bound task (b) I/O-bound task

CPU-bound

Το CPU-bound είναι ένα πρόγραμμα όπου ο επεξεργαστής χρειάζεται συνεχώς για την εκτέλεση υπολογισμών και δεν έχει πολλές περιόδους αδράνειας. Αυτό σημαίνει ότι η παράλληλη εκτέλεση αυτών των προγραμμάτων με νήματα μπορεί να είναι δύσκολη ή ακόμη και αναποτελεσματική λόγω του GIL. Τα CPU-bound προγράμματα εκτελούν πολύπλοκους υπολογισμούς, επεξεργάζονται μεγάλους όγκους δεδομένων, επιστημονικούς υπολογισμούς και άλλες εργασίες με εντατική χρήση του επεξεργαστή.

Για CPU-bound εργασίες σε Python, μια εναλλακτική λύση μπορεί να είναι η χρήση διεργασιών αντί για νήματα, καθώς οι διεργασίες στην Python δεν δεσμευονται απο το GIL και μπορούν να εκτελούνται παράλληλα. Αυτό μπορεί να βοηθήσει στην εκμετάλλευση των πολλαπλών πυρήνων του επεξεργαστή. Ωστόσο, η επικοινωνία μεταξύ διεργασιών μπορεί να είναι πιο δύσκολη από την επικοινωνία μεταξύ νημάτων, καθώς πρέπει να χρησιμοποιηθούν δίαυλοι επικοινωνίας όπως η ουρά για τη μεταφορά δεδομένων μεταξύ των διεργασιών.

Ορισμένα από τα πλεονεκτήματα της χρήσης διεργασιών για CPU-bound εργασίες ειναι:

- Οι διεργασίες μπορούν να εκτελούνται παράλληλα σε πολλούς πυρήνες του επεξεργαστή, εκμεταλλευόμενες πλήρως τη δυνατότητα παραλληλισμού των υπολογισμών.
- Το GIL αφορά μόνο τα νήματα στην Python σε αντίθεση με τις διεργασίες. Δηλαδή μπορούμε να εκτελούμε CPU-bound εργασίες παράλληλα με διεργασίες χωρίς τον περιορισμό που επιβάλλει το GIL.
- Η διεργασίες είναι αυτόνομες επειδή δεν έχουν κοινόχρηστη μνήμη. Αυτό σημαίνει ότι μια διεργασία δεν μπορεί να παρεμβαίνει στη μνήμη άλλων διεργασιών. Έτσι προσφέρεται απομόνωση και ασφάλεια.

Στις CPU-bound διεργασίες η σειριακή εκτέλεση υπερτερεί της αντίστοιχης με νήματα λόγω του GIL και του επιπλέον φόρτου διαχείρισης των νημάτων.

I/O-bound

Ένα πρόγραμμα είναι I/O-bound όταν για την εκτέλεση μιας εργασίας περιορίζεται από τα αιτήματα εισόδου/εξόδου και όχι από την υπολογιστική ισχύ του επεξεργαστή. Τα I/O-Bound προγράμματα περιλαμβάνουν το διάβασμα μεγάλων αρχείων του δίσκου, το κατέβασμα αρχείων από το διαδίκτυο, η αποστολή και λήψη σε βάσεις δεδομένων και άλλες εργασίες που απαιτούν περισσότερο χρόνο για εισόδους /εξόδους απ' ότι για υπολογισμούς.

Η χρήση νημάτων μπορεί να είναι αποδοτικότερη από τη χρήση διεργασιών, επειδή η δημιουργία και η διαχείριση νημάτων είναι πιο ελαφριά σε όρους απαιτούμενης μνήμης και επεξεργαστικής ισχύος σε σχέση με τη δημιουργία και διαχείριση διεργασιών. Επιπρόσθετα, τα νήματα μοιράζονται την ίδια μνήμη , κάτι που μπορεί να είναι χρήσιμο όταν πρέπει να ανταλλάξουν δεδομένα γρηγορότερα μεταξύ τους.

Απαιτείται προσεκτική διαχείριση κατά τη χρήση των νημάτων καθώς μπορεί να προκαλέσει προβλήματα όπως τον κίνδυνο του ανταγωνισμού και του κλειδώματος .

Ο ανταγωνισμός αναφέρεται στην κατάσταση όπου πολλά νήματα ή διεργασίες προσπαθούν να αποκτήσουν πρόσβαση στους ίδιους πόρους την ιδία στιγμή. Οι πόροι μπορεί να είναι μνήμη, αρχεία, συσκευές εισόδου/εξόδου, κ.α. Ο ανταγωνισμός για πόρους μπορεί να οδηγήσει σε καθυστέρηση και σφάλματα κατά την εκτέλεση, καθώς τα νήματα ή οι διεργασίες πρέπει να περιμένουν να αποκτήσουν πρόσβαση στους πόρους πριν συνεχίσουν. Μπορεί να οδηγήσουν σε ανεπιθύμητη

συμπεριφορά του προγράμματος, καθώς η ανάμιξη διαφόρων νημάτων στην ίδια μεταβλητή μπορεί να παραβιάσει την ακεραιότητα των δεδομένων. Για την αποφυγή του ανταγωνισμού, πρέπει να χρησιμοποιούμε μηχανισμούς συγχρονισμού, όπως τα κλειδώματα, προκειμένου να διασφαλίσουμε ότι μόνο ένα νήμα μπορεί να αυξήσει τη μεταβλητή για να εξασφαλίσουμε τη σωστή λειτουργία του προγράμματος.

Ορισμένα από τα πλεονεκτήματα της χρήσης νημάτων για I/O-bound εργασίες είναι:

- Η δημιουργία ενός νήματος είναι πιο γρήγορη και απαιτεί λιγότερη μνήμη από τη δημιουργία μιας νέας διεργασίας.
- Τα νήματα έχουν κοινόχρηστη μνήμη οπότε μπορούν να ανταλλάσσουν δεδομένα μεταξύ τους χωρίς την ανάγκη για περίπλοκη επικοινωνία.
- Επειδή τα νήματα μοιράζονται τη μνήμη, η επικοινωνία μεταξύ τους μπορεί να είναι γρηγορότερη από την επικοινωνία μεταξύ διεργασιών.

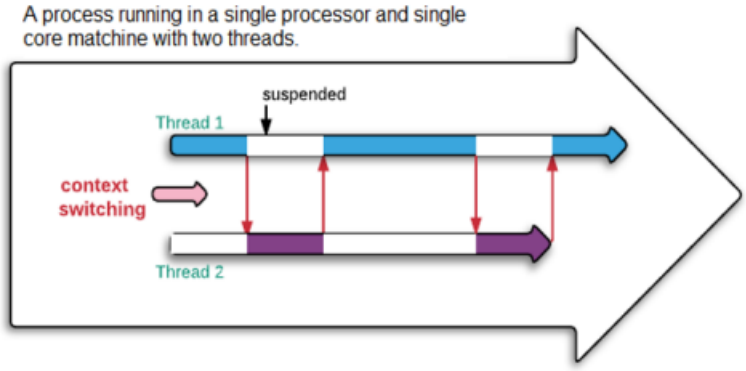
Στις I/O-bound διεργασίες η εκτέλεση με νήματα υπερτερεί αυτής των διεργασιών επειδή η δημιουργία και η διαχείριση των τελευταίων απαιτεί περισσότερο χρόνο.

Από όλα τα παραπάνω καταλήγουμε στα εξής συμπεράσματα:

- Αν ο αλγόριθμος είναι CPU-bound ο παραλληλισμός γίνεται σε επίπεδο διεργασιών (processes).

- Αν ο αλγόριθμος είναι I/O-bound ο παραλληλισμός γίνεται σε επίπεδο νημάτων (threads).
- Αν δεν είμαστε βέβαιοι για το είδος του αλγορίθμου, η ασφαλής επιλογή είναι ο παραλληλισμός σε επίπεδο διεργασιών.

GIL

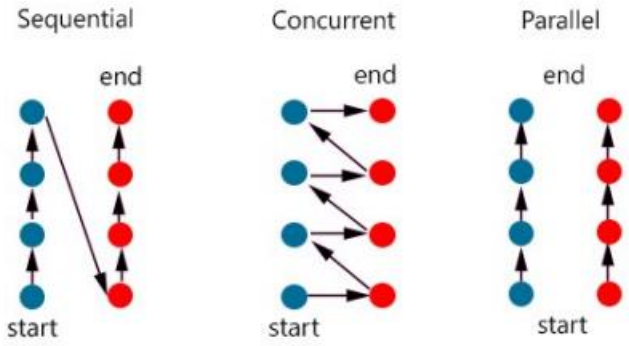


Η εκτέλεση πολλαπλών νημάτων στην ίδια υπολογιστική μονάδα (cpu ή core) είναι συντρέχουσα (concurrent). Ο διερμηνέας της Python είναι υπεύθυνος για την μη επιτρεπτή εκτέλεση παραπάνω από ενός νήματος κατά την ίδια χρονική στιγμή. Αυτό ρυθμίζεται με το GIL (Global Interpreter Lock) που είναι ένα είδος κλειδώματος το όποιο επιτρέπει μόνο σε ένα νήμα να έχει τον έλεγχο του διερμηνέα (έκδοση cpython) ακόμα και αν ανατεθούν σε διαφορετικούς επεξεργαστές ή πυρήνες. Ο λόγος για τον οποίο χρησιμοποιείται το GIL είναι η απλότητα της υλοποίησης του διερμηνέα, αφού τον προστατεύει από προβλήματα συναρτήσεων και δεδομένων που μπορεί να προκύψουν από παράλληλη εκτέλεση.

ΚΕΦΑΛΑΙΟ 3

Τρόποι εκτέλεσης εργασιών στην cpu

Η εκτέλεση εργασιών στην cpu μπορεί να γίνει με τρεις διαφορετικούς τρόπους: σειριακή ή ακολουθιακή (sequential), συντρέχουσα (concurrent) και παράλληλη (parallel).

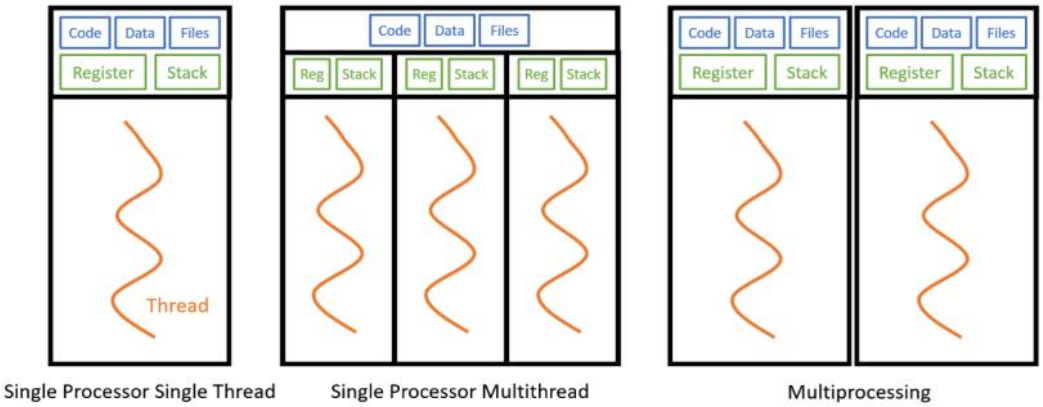


Η Σειριακή ή Ακολουθιακή είναι η συμβατική εκτέλεση ενός προγράμματος. Οι εντολές εκτελούνται η μία μετά την άλλη. Πιο συγκεκριμένα, η εκτέλεση της επόμενης εντολής ξεκινά αφού έχει ολοκληρωθεί η προηγούμενη. Ακόμα και αν έχουμε περισσότερους του ενός επεξεργαστή ένας μόνο εκτελεί το πρόγραμμα και οι υπόλοιποι μένουν ανενεργοί. Όταν ο όγκος των δεδομένων είναι μικρός υπερτερεί της παράλληλης εκτέλεσης, επειδή η επιπλέον επιβάρυνση που εισάγεται για τη

διαχείριση του παραλληλισμού είναι μεγαλύτερη από το «χρήσιμο» χρόνο επεξεργασίας.

Στη Συντρέχουσα εκτέλεση δρομολογούνται δυο ή περισσότερες εργασίες για ταυτόχρονη εκτέλεση. Σε κάθε χρονική στιγμή μόνο μια εργασία απασχολεί την cpu. Την απόφαση για το ποια εργασία θα αποκτήσει προσωρινά τον έλεγχο της cpu για να εκτελέσει μέρος αυτής, την παίρνει ο δρομολογητής διεργασιών του λειτουργικού συστήματος . Λόγω του ότι οι εργασίες εναλλάσσονται πολύ γρήγορα δημιουργείται η ψευδαίσθηση της παράλληλης επεξεργασίας.

Παράλληλη εκτέλεση είναι όταν δυο ή περισσότερες εργασίες εκτελούνται ταυτόχρονα. Το πλήθος των εργασιών μπορούν να εκτελεστούν πραγματικά ταυτόχρονα αλλά πάντα εξαρτάται από το πλήθος των διαθέσιμων μονάδων επεξεργασίας. Η παράλληλη εκτέλεση μπορεί να γίνει είτε σε επίπεδο νημάτων (threads) είτε σε επίπεδο διεργασιών (processes). Οι σειριακοί αλγόριθμοι θα πρέπει να επανασχεδιαστούν κάτι που δεν είναι πάντα εφικτό (π.χ. υπολογισμός ακολουθίας Fibonacci).



Single Processor Single Thread Single Processor Multithread Multiprocessing

Ο ηλεκτρονικός υπολογιστής αποτελείται από διάφορα τσιπάκια όπως η CPU, η RAM, οι συσκευές I/O κ.α. Δεν μπορούμε να γράψουμε ένα πρόγραμμα σε Python και να το φορτώσουμε κατευθείαν στα τσιπάκια. Αυτό μπορεί να γίνει μόνο στους μικροελεγκτές όπως έχουν τα ψυγεία. Στους προσωπικούς υπολογιστές υπάρχει ένα φράγμα το οποίο είναι το λειτουργικό σύστημα. Οτιδήποτε θέλουμε να κάνουμε με το υλικό του ηλεκτρονικού υπολογιστή πρέπει να το ζητήσουμε από το λειτουργικό σύστημα. Ακόμη και αν υποστηρίζει χαμηλού επιπέδου εντολές όπως η C γίνεται μέσω του λειτουργικού συστήματος και δεν μπορεί να παρακαμφθεί.

Όταν ζητάμε από το λειτουργικό σύστημα να εκτελέσει ένα πρόγραμμα, το λειτουργικό σύστημα αναλαμβάνει τη διαχείριση των πόρων και την εκτέλεση του προγράμματος. Δηλαδή φορτώνει το πρόγραμμα στη μνήμη, διαμορφώνει το περιβάλλον εκτέλεσης και εκκινεί την εκτέλεση του προγράμματος. Αυτή η διαδικασία είναι υπεύθυνη για την εξασφάλιση της σωστής εκτέλεσης του προγράμματος και την διαχείριση των πόρων του υπολογιστή.

Δρομολογητής διεργασιών

Η δρομολόγηση διεργασιών σε ένα υπολογιστικό σύστημα επεξεργαστή αποτελεί κρίσιμο στοιχείο για την επίτευξη υψηλών επιδόσεων. Κατά τη διαχείριση των διεργασιών, επιδιώκουμε να επιτύχουμε αρκετούς στόχους, οι οποίοι όμως μπορεί να είναι αντίθετοι μεταξύ τους.

Ένας από τους στόχους είναι το υψηλό ποσοστό χρήσης του επεξεργαστή (CPU utilization), προκειμένου να εκμεταλλευθούμε πλήρως τη δυνατότητα του. Παράλληλα, επιδιώκουμε υψηλή ρυθμοαπόδοση (High throughput), με τον υπολογισμό του πλήθους διεργασιών που ολοκληρώνονται κατά μονάδα χρόνου. Ωστόσο, αυτοί οι στόχοι μπορεί να είναι αλληλοσυγκρουόμενοι με τον στόχο του μικρού χρόνου απόκρισης (response time).

Ο μικρός χρόνος απόκρισης είναι ουσιώδης για την αποτελεσματική ανταπόκριση του συστήματος σε αιτήματα χρηστών. Αν καταβάλλουμε προσπάθειες για τη μεγιστοποίηση του CPU utilization και του throughput, ενδέχεται να αυξηθεί ο χρόνος απόκρισης, καθιστώντας το σύστημα λιγότερο αποδοτικό για τους χρήστες.

Συνεπώς, η δρομολόγηση διεργασιών πρέπει να επιτυγχάνει μια ισορροπία μεταξύ αυτών των αντιφατικών στόχων, στοχεύοντας στη βέλτιστη απόδοση του συστήματος. Η συνεχής παρακολούθηση και προσαρμογή της δρομολόγησης είναι καίρια για την αντιμετώπιση των διαρκών αλλαγών στις απαιτήσεις και συνθήκες του συστήματος.

Κριτήρια αποτίμησης της απόδοσης

Η απόδοση ενός συστήματος και η αποτελεσματικότητα της δρομολόγησης διεργασιών από το λειτουργικό σύστημα αξιολογούνται με βάση πολλά κριτήρια. Αυτά περιλαμβάνουν:

• Δικαιοσύνη (fairness): αναφέρεται συχνή χρήση της CPU από κάθε διεργασία.

• Χρησιμοποίηση (utilization): το ποσοστό του χρόνου κατά το οποίο μια

συσκευή χρησιμοποιείται , όπως εκφράζεται από το πηλίκο του χρόνου χρήσης προς το συνολικό χρόνο.

• Ρυθμό-απόδοση (throughput): Ο αριθμός των εργασιών που ολοκληρώνονται κατά μέσο όρο σε μια χρονική περίοδο, μετρώντας τη συνολική απόδοση του συστήματος.

• Χρόνος επιστροφής (turnaround time): Ο συνολικός χρόνος από την υποβολή μιας διεργασίας έως την ολοκλήρωσή της.

• Χρόνος αναμονής (waiting time): Ο χρόνος που μια διεργασία περιμένει σε ουρά αναμονής πριν από την εκτέλεσή της.

• Χρόνος απόκρισης (response time): Ο χρόνος από την υποβολή μιας απαίτησης μέχρι την πρώτη απόκριση του συστήματος και όχι τη στιγμή εξόδου.

• Εναλλαγές πλαισίων (context switches): χρόνος που σπαταλιέται για να

γίνει εναλλαγή διεργασιών στη CPU

• Πολυπλοκότητα του αλγορίθμου δρομολόγησης: χρόνος που απαιτείται για την επιλογή της επόμενης διεργασίας από τη λίστα των έτοιμων διεργασιών

Κριτήρια βελτιστοποίησης

Η δρομολόγηση διεργασιών πρέπει να επιδιώκει να επιτύχει ένα ισορροπημένο σύνολο κριτηρίων βελτιστοποίησης. Αρχικά, υπάρχει η πρόκληση να μεγιστοποιηθεί η χρήση του επεξεργαστή και η ρυθμοαπόδοση του συστήματος. Ταυτόχρονα, όμως, έχουμε την ανάγκη να ελαχιστοποιηθούν οι χρόνοι επιστροφής, αναμονής και απόκρισης των διεργασιών.

Ως αποτέλεσμα, αυτά τα κριτήρια βελτιστοποίησης συχνά είναι αλληλοσυγκρουόμενα. Επιδίωξη της μέγιστης χρήσης του επεξεργαστή και της υψηλής ρυθμοαπόδοσης μπορεί να οδηγήσει σε αύξηση των χρόνων αναμονής και απόκρισης, επηρεάζοντας αρνητικά την εμπειρία του χρήστη. Επομένως, η διαδικασία δρομολόγησης πρέπει να επιλύει αυτήν την αντίφαση και να βρίσκει την ιδανική ισορροπία που εξυπηρετεί τους διάφορους στόχους του συστήματος.

Τύποι δρομολόγησης του επεξεργαστή

Η δρομολόγηση διεργασιών χωρίζεται σε τρεις κατηγορίες: μακροπρόθεσμη, μεσοπρόθεσμη και βραχυπρόθεσμη. Η μακροπρόθεσμη δρομολόγηση καθορίζει ποιες διεργασίες θα γίνουν αποδεκτές για επεξεργασία από το σύστημα. Στη μεσοπρόθεσμη δρομολόγηση, αποφασίζεται ποιες διεργασίες θα προστεθούν στις διεργασίες που βρίσκονται στη μνήμη. Τέλος, η βραχυπρόθεσμη δρομολόγηση επιλέγει ποια διεργασία θα εκτελεστεί στον επεξεργαστή.

Στη μακροπρόθεσμη δρομολόγηση, καθορίζεται ποια προγράμματα θα γίνουν αποδεκτά για επεξεργασία, ελέγχοντας το βαθμό πολυπρογραμματισμού του συστήματος. Προσπαθεί να διατηρήσει ισορροπία μεταξύ των διεργασιών που είναι εκτεταμένες στην CPU και εκείνων που ασχολούνται με λειτουργίες I/O.

Στη μεσοπρόθεσμη δρομολόγηση, οι αποφάσεις βασίζονται στη διαχείριση του πολυπρογραμματισμού, ενώ υλοποιείται από το λογισμικό του λειτουργικού συστήματος για τη διαχείριση της μνήμης.

Η βραχυπρόθεσμη δρομολόγηση επιλέγει ποια διεργασία θα εκτελεστεί επόμενη και είναι υπεύθυνη για τον έλεγχο του επεξεργαστή. Αποφασίζει με βάση κριτήρια όπως ο χρόνος απόκρισης και ο χρόνος επιστροφής, καθώς και η χρησιμοποίηση της CPU, η δικαιοσύνη και η ρυθμοαπόδοση.

Κάθε κατηγορία διαδραματίζει σημαντικό ρόλο στη διαχείριση των διεργασιών, με τη βραχυπρόθεσμη δρομολόγηση να επηρεάζει άμεσα την απόδοση του συστήματος.

Πολιτικές δρομολόγησης

Η δρομολόγηση διεργασιών είναι ουσιαστικά η διαδικασία επιλογής της επόμενης διεργασίας που θα εκτελεστεί στο σύστημα. Η πολιτική δρομολόγησης καθορίζει πότε λαμβάνονται αποφάσεις σχετικά με την επιλογή αυτής της διεργασίας. Στην περίπτωση δρομολόγησης χωρίς προεκχώρηση, μια διεργασία συνεχίζει να

εκτελείται μέχρι να ολοκληρωθεί ή να ανασταλεί από μόνη της περιμένοντας για την ολοκλήρωση I/O, χωρίς να διακόπτεται.

Στην περίπτωση προεκχώρησης, η τρέχουσα διεργασία που εκτελείται μπορεί να διακοπεί και να ανασταλεί, επιτρέποντας σε άλλη διεργασία να αναλάβει τον έλεγχο της CPU. Αυτό βοηθά στην αποτελεσματικότερη χρήση του επεξεργαστή, αφού καμία διεργασία δεν μονοπωλεί τη CPU για μεγάλο χρονικό διάστημα.

Ο δρομολογητής του επεξεργαστή είναι μια διεργασία που επιλέγει διεργασίες από την ουρά των έτοιμων διεργασιών στην κύρια μνήμη. Αυτή η διεργασία αποτελεί κρίσιμο συστατικό του λειτουργικού συστήματος, καθώς επηρεάζει την απόδοση του συστήματος και τον χρόνο που αναλογεί σε κάθε διεργασία.

Πολλοί παράγοντες επηρεάζουν τη δρομολόγηση, όπως είναι αν η διεργασία είναι CPU-bound ή I/O-bound, αν είναι αλληλεπιδραστική ή batch, η προτεραιότητά της, ο χρόνος που έχει ήδη εκτελεστεί και ο χρόνος που απαιτείται για να ολοκληρωθεί. Η πολιτική δρομολόγησης είναι σημαντική για την εξασφάλιση ισορροπίας και αποδοτικότητας στο σύστημα.

Αλγόριθμοι Δρομολόγησης

Οι αλγόριθμοι δρομολόγησης καθορίζουν τον τρόπο με τον οποίο ο δρομολογητής επιλέγει την επόμενη διεργασία προς εκτέλεση και τη σειρά με την οποία οι διεργασίες εκτελούνται στο σύστημα. Αυτοί οι αλγόριθμοι διαμορφώνουν τη συμπεριφορά του δρομολογητή και χωρίζονται σε δύο κατηγορίες: προεκχωρούμενοι και μη προεκχωρούμενοι.

Στην ουσία, αυτοί οι αλγόριθμοι περιγράφουν τη λογική πίσω από τις ενέργειες που πραγματοποιεί ο δρομολογητής. Οι προεκχωρούμενοι αλγόριθμοι επιτρέπουν τη διακοπή της εκτέλεσης μιας διεργασίας για να εκτελεστεί μια άλλη, ενώ οι μη προεκχωρούμενοι απαιτούν την ολοκλήρωση της τρέχουσας διεργασίας πριν από την επιλογή μιας νέας.

Υπάρχουν πολλοί αλγόριθμοι δρομολόγησης, όπως οι First-Come-First-Serve (FCFS), Shortest Job First (SJF), Shortest Remaining Time First (SRTF), βασισμένοι σε προτεραιότητες, round robin, καθώς και οι πολυεπίπεδοι με ανατροφοδότηση (multilevel feedback). Κάθε ένας από αυτούς τους αλγορίθμους έχει τις δικές του χαρακτηριστικές ιδιότητες και επιδόσεις, επηρεάζοντας τη συνολική απόδοση του συστήματος.

ΚΕΦΑΛΑΙΟ 4

Παραλληλοποίηση με νήματα

```
import time #Εισαγωγή χρονομέτρου
import threading as th #Εισαγωγή νημάτων
lock = th.Lock() # Δήλωση μεταβλητής κλειδώματος

def worker(a):

   lock.acquire() # Δέσμευση κλειδιού
  ……        #πράξεις
   lock.release() #Απελευθέρωση κλειδιού

## main thread

start_time=time.perf_counter()  #Εναρξη χρονομέτρου

t1=th.Thread(target=worker,args=(aa,)) #Δήλωση νημάτων
t2=th.Thread(target=worker,args=(aa,))
t1.start() #Ξεκίνημα Νημάτων
t2.start()
t1.join()  # Περιμένει μέχρι να ολοκληρωθεί το νήμα
t2.join()

end_time=time.perf_counter() #Τερματισμός χρονομέτρου
```

Για να δημιουργήσουμε και να διαχειριστούμε τα νήματα πρέπει να εισάγουμε την βιβλιοθήκη threading <<import threading as th>>. Επίσης θα εισάγουμε και τη βιβλιοθήκη time για την χρονομέτρηση μας<< import time>>. Θα δημιουργήσουμε ένα κλείδωμα χρησιμοποιώντας τη βιβλιοθήκη threading , το όποιο θα μας βοηθήσει στο να διαχειριστούμε την πρόσβαση σε κοινόχρηστες μεταβλητές προκειμένου να αποφευχθούν προβλήματα ανταγωνισμού << lock = th.Lock()>>.

Μέσα στη συνάρτηση μας << def worker(a):>> και ανάμεσα στις γραμμές << lock.acquire()>> και << lock.release()>> οι γραμμές κώδικα που υπάρχουν

προστατεύονται από την απόκτηση και την απελευθέρωση του κλειδώματος . Αυτό διασφαλίζει ότι μόνο ένα νήμα κάθε φορά μπορεί να εκτελεί το συγκεκριμένο τμήμα κώδικα.

Στο κυρίως πρόγραμμα δημιουργούμε τα νήματα μας. Με την εντολή

<< t1=th.Thread(target=worker,args=(aa,))>> δηλώνουμε τα νήματα. Η μεταβλητή <t1>> αντιπροσωπεύει το νήμα, το <<th.Tread>> δημιουργεί ένα νέο αντικείμενο νήματος όπου το <<target>> καθορίζει ποια συνάρτηση θα εκτελεστεί όταν ξεκινήσει το νήμα, στην περίπτωση μας η <<worker>> και το <<args>> όπου παίρνει τα ορίσματα της συνάρτησης μας δηλαδή το <<aa>>. Τα νήματα ξεκινούν με τις εντολές <<start>>. Ενώ με τις εντολές <<join>> περιμένουν να ολοκληρωθεί η εκτέλεση όλων των νημάτων για να συνεχιστεί η εκτέλεση του κυρίως νήματος.

ΚΕΦΑΛΑΙΟ 5

Παραλληλοποίηση με διεργασίες

```
import time
import multiprocessing as mp

def worker(a,q):
            ……
   q.put(a)

## main processing
if __name__ == "__main__":
   start_time = time.perf_counter()

   queue = mp.Queue()
   p1 = mp.Process(target=worker,args=(aa,queue))
   p2 = mp.Process(target=worker,args=(aa,queue))
   p1.start
   p2.start
   p1.join
   p2.join
   total = queue.get() + queue.get()

   end_time=time.perf_counter()
```

Ο παραπάνω κώδικας είναι ένα παράδειγμα χρήσης παράλληλων διεργασιών (multiprocessing). Χρησιμοποιούμε τις βιβλιοθήκες multiprocessing και time. Η multiprocessing χρησιμοποιείται για τη δημιουργία και τον έλεγχο των διεργασιών , ενώ η time για τη μέτρηση του χρόνου εκτέλεσης. <<import time>> και <<import multiprocessing as mp>>.

Στη συνεχεία έχουμε την συνάρτηση worker όπου δέχεται δυο ορίσματα το a και το q. Στην περίπτωση μας η συνάρτηση το μόνο που κάνει είναι να τοποθετεί το a στο q με την εντολή <<q.put(a).>>

Ο κυρίως κώδικας στις διεργασίες χρειάζεται να εκτελέσει έναν έλεγχο << if __name__ == "__main__":>>ετσι ώστε να διασφαλίσει ότι θα εκτελεστεί μόνο από το κυρίως πρόγραμμα. Θα ενεργοποιήσουμε το χρονόμετρο με την εντολή <<start_time = time.perf_counter()>>. Μετά θα δημιουργήσουμε μια ουρά για να πετυχουμε την επικοινωνία μεταξύ των διεργασιών <<queue = mp.Queue()>>. Οι διεργασίες δηλώνονται ως εξής << p1 = mp.Process(target=worker,args=(aa,queue))>>. Η μεταβλητή <<p1>> αντιπροσωπεύει τη διεργασία, το <<mp.Process>> δημιουργεί ένα νέο αντικείμενο διεργασίας όπου το <<target>> καθορίζει ποια συνάρτηση θα εκτελεστεί όταν ξεκινήσει η διεργασία, στην περίπτωση μας η <<worker>> και το <<args>> όπου παίρνει τα ορίσματα της συνάρτησης μας δηλαδή το <<aa>> και το <<queue>>. Οι διεργασίες μας εκκινούνται με τις εντολές <<start()>> . Επίσης με τις εντολές <<join >> περιμένει το πρόγραμμα μας μέχρι να ολοκληρωθούν οι αντίστοιχες διεργασίες . Από τη στιγμή που χρησιμοποιούμε την <<queue>> οι join είναι περιττές γιατί θα γίνει ο έλεγχος ολοκλήρωσης από την ουρά. Η άντληση των δεδομένων από την ουρά γίνεται με την εντολή <<queue.get()>> , ανάλογα με το πόσες διεργασίες έχουμε τόσες ουρές θα προσθέσουμε στο τελικό αποτέλεσμα.

ΚΕΦΑΛΑΙΟ 6

Δεξαμενές

Οι δεξαμενές εφαρμόζονται είτε στις διεργασίες είτε στα νήματα. Η συχνότερη χρήση είναι στις διεργασίες όπου έχουμε πραγματικά παράλληλο περιβάλλον.

Δεξαμενές με διεργασίες

Αρχικά δημιουργούμε όσες διεργασίες θέλουμε χωρίς να τις εκχωρήσουμε συναρτήσεις και τις έχουμε στην άκρη. Κάθε φορά που θέλουμε να κάνουμε έναν παράλληλο υπολογισμό στέλνουμε μια συνάρτηση στη δεξαμενή .Οι διεργασίες που είχαμε δημιουργήσει κάνουν τις πράξεις και μας επιστρέφουν το αποτέλεσμα. Επίσης οι δεξαμενές μένουν ενεργές για να τις ξαναχρησιμοποιήσουμε. Αυτή είναι και η βασική διάφορα με τις διεργασίες. Ενα μεγάλο πλεονέκτημα είναι ότι η συνάρτηση μας μπορεί να επιστρέψει τιμές με την return .

```
import time
import multiprocessing as mp

def worker(a):
    s=0
    for i in a:
        s = s + i**2
    return s

## main processing
if __name__ == "__main__":
    start_time = time.perf_counter()
    n = 10**8
    seq = range(1, n+1)
    pool = mp.Pool(4)
    result =
pool.map(worker,(seq[0:n//4],seq[n//4:n//2],seq[n//2:3*n//4],seq[3*n//4:n]))
    pool.close()
    total = sum(result)
    print(total)
    end_time=time.perf_counter()
    print("process pool : ",end_time-start_time,"sec")
```

Στον παραπάνω κώδικα χρησιμοποιούμε την συνάρτηση worker. Η συνάρτηση μας μπορεί να πάρει μόνο ένα όρισμα γιατί στην δεξαμενή έχουμε βάλει τη συνάρτηση map. Αν θέλουμε να χρησιμοποιήσουμε παραπάνω από ένα ορίσματα μπορούμε να χρησιμοποιήσουμε τη συνάρτηση starmap. Δηλαδή η συνάρτηση θα γίνει <<def worker(a,power): >> και η << result=pool.starmap(worker, ((seq , 2),(seq , 3)))>>.

Οι δεξαμενές είναι ένα ισχυρό εργαλείο στην παράλληλη εκτέλεση εργασιών . Είναι χρήσιμες στην επίλυση εργασιών μεγάλου όγκου. Μπορούμε να τις ξαναχρησιμοποιήσουμε βελτιώνοντας την απόδοση, μειώνοντας τον χρόνο που χρειάζεται για να τις ενεργοποιήσουμε και να τις απενεργοποιήσουμε.

Δεξαμενές με νήματα

```
import time
import concurrent.futures as cf

def worker(a):
    s = 0
    for i in a:
        s += i**2
    return s

start_time = time.perf_counter()
n = 10**8
seq = (range(1, n+1))
executor = cf.ThreadPoolExecutor(max_workers=4)
#  map  με νήματα
result = executor.map(worker,
(seq[0:n//4],seq[n//4:n//2],seq[n//2:3*n//4],seq[3*n//4:n]))
# κλεισιμο  ThreadPoolExecutor
executor.shutdown()
#  αποτελεσμα
total = sum(result)
end_time=time.perf_counter()
print("thread pool : ",end_time-start_time,"sec")
print(total)
```

Στον παραπάνω κώδικα χρησιμοποιούμε τη βιβλιοθήκη concurrent.futures για τον παράλληλο υπολογισμό του αθροίσματος των τετραγώνων μιας ακολουθίας αριθμών. Αρχικά, ορίζεται η συνάρτηση worker, η οποία δέχεται έναν πίνακα αριθμών και υπολογίζει το άθροισμα των τετραγώνων τους. Στη συνέχεια, καθορίζεται μια ακολουθία αριθμών από το 1 έως το n. Έπειτα, δημιουργείται ένα αντικείμενο ThreadPoolExecutor με έως και 4 νήματα εργασίας. Η ακολουθία διαιρείται σε τέσσερα ισόποσα κομμάτια και η συνάρτηση worker εφαρμόζεται παράλληλα σε αυτά χρησιμοποιώντας τη μέθοδο map του ThreadPoolExecutor. Μετά τον υπολογισμό, το αντικείμενο ThreadPoolExecutor κλείνει, και τα αποτελέσματα των υπολογισμών συγκεντρώνονται στην total , το συνολικό άθροισμα υπολογίζεται με τη χρήση της συνάρτησης sum. Τέλος μετριέται ο χρόνος εκτέλεσης του προγράμματος. Ο παραπάνω κώδικας επιτρέπει την παραλληλοποίηση των υπολογισμών, εκμεταλλευόμενος πολλά νήματα εργασίας για την επιτάχυνση της εκτέλεσης.

ΚΕΦΑΛΑΙΟ 7

Python CPU Παράδειγμα 1

Παρακάτω θα υλοποιήσουμε ένα παράδειγμα προσθέτοντας τα τετράγωνα και τους κύβους των ορών μιας ακολουθίας. Η δόκιμη θα γίνει σειριακά ,με νήματα και παράλληλα .Στο τέλος θα συγκρίνουμε τους χρόνους για να δούμε την αποδοτικότητα του κάθε αλγορίθμου.

Σειριακά

```
import time
total=0
# ΥΠΟΛΟΓΙΣΜΟΣ ΤΟΥ ΤΕΤΡΑΓΩΝΟΥ ΚΑΙ ΚΥΒΟΥ
def worker(a):
    global total
    for i in a:
        total=total + i**2 + i**3

#ΣΕΙΡΙΑΚΗ ΕΚΤΕΛΕΣΗ
n=10**7
seq=range(1,n+1)

start_time=time.perf_counter()

worker(seq)


end_time=time.perf_counter()

print("ΣΥΝΟΛΟ : ",total)
print("Ο ΧΡΟΝΟΣ ΤΗΣ ΣΕΙΡΙΑΚΗΣ ΕΚΤΕΛΕΣΗΣ ΕΙΝΑΙ : ",end_time-start_time,"sec")
```

ΣΥΝΟΛΟ : 250000083333408333335000000

Ο ΧΡΟΝΟΣ ΤΗΣ ΣΕΙΡΙΑΚΗΣ ΕΚΤΕΛΕΣΗΣ ΕΙΝΑΙ : 1,6725978000031319 sec

Με δυο νήματα

```
import time
import threading as th
total=0
lock = th.Lock()

def worker(a,power):
    global total
    local_sum=0
    for i in a:
        local_sum=local_sum + i**power
    lock.acquire()
    total=total+local_sum
    lock.release()

## main thread

start_time=time.perf_counter()
n=10**7
seq=range(1,n+1)
t1 = th.Thread(target=worker,args=(seq,2,))
t2 = th.Thread(target=worker,args=(seq,3,))

t1.start()
t2.start()
t1.join()
t2.join()
end_time=time.perf_counter()

print("ΣΥΝΟΛΟ : ",total)
print("Ο ΧΡΟΝΟΣ ΜΕ 2 ΝΗΜΑΤΑ ΕΙΝΑΙ : ",end_time-start_time,"sec")
```

ΣΥΝΟΛΟ : 250000083333340833333500000

Ο ΧΡΟΝΟΣ ΜΕ 2 ΝΗΜΑΤΑ ΕΙΝΑΙ : 1,7660266000020783 sec

Με τέσσερα νήματα

```
import time
import threading as th
total=0
lock = th.Lock()
#ΥΠΟΛΟΓΙΣΜΟΣ ΤΗΣ ΔΥΝΑΜΗΣ
def worker(a,power):
    global total
    local_sum=0
    for i in a:
        local_sum=local_sum + i**power
    lock.acquire()
    total=total+local_sum
    lock.release()

## main thread
start_time=time.perf_counter()
n=10**7
seq=range(1,n+1)
t1 = th.Thread(target=worker,args=(seq[0:n//2],2,))
t2 = th.Thread(target=worker,args=(seq[0:n//2],3,))
t3 = th.Thread(target=worker,args=(seq[n//2:n],2,))
t4 = th.Thread(target=worker,args=(seq[n//2:n],3,))
t1.start()
t2.start()
t3.start()
t4.start()
t1.join()
t2.join()
t3.join()
t4.join()
end_time=time.perf_counter()
```

ΣΥΝΟΛΟ : 250000083333340833333500000 0

Ο ΧΡΟΝΟΣ ΜΕ 4 ΝΗΜΑΤΑ ΕΙΝΑΙ : 1,8132508999842685sec

Με οχτώ νήματα

```
import time
import threading as th
total=0
lock = th.Lock()
#ΥΠΟΛΟΓΙΣΜΟΣ ΤΗΣ ΔΥΝΑΜΗΣ
def worker(a,power):
    global total
    local_sum=0
    for i in a:
        local_sum=local_sum + i**power
    lock.acquire()
    total=total+local_sum
    lock.release()

## main thread
start_time=time.perf_counter()
n=10**7
seq=range(1,n+1)
t1 = th.Thread(target=worker,args=(seq[0:n//4],2,))
t2 = th.Thread(target=worker,args=(seq[0:n//4],3,))
t3 = th.Thread(target=worker,args=(seq[n//4:n//2],2,))
t4 = th.Thread(target=worker,args=(seq[n//4:n//2],3,))
t5 = th.Thread(target=worker,args=(seq[n//2:3*n//4],2,))
t6 = th.Thread(target=worker,args=(seq[n//2:3*n//4],3,))
t7 = th.Thread(target=worker,args=(seq[3*n//4:n],2,))
t8 = th.Thread(target=worker,args=(seq[3*n//4:n],3,))
t1.start()
t2.start()
t3.start()
t4.start()
t5.start()
t6.start()
t7.start()
t8.start()
t1.join()
t2.join()
t3.join()
t4.join()
t5.join()
t6.join()
t7.join()
t8.join()
end_time=time.perf_counter()
print("ΣΥΝΟΛΟ : ",total)
print("Ο ΧΡΟΝΟΣ ΜΕ 8 ΝΗΜΑΤΑ ΕΙΝΑΙ : ",end_time-start_time,"sec")
```

ΣΥΝΟΛΟ : 250000083333340833333500000 0

Ο ΧΡΟΝΟΣ ΜΕ 8 ΝΗΜΑΤΑ ΕΙΝΑΙ : 1,8689316000090912 sec

Με δυο διεργασίες και ουρά

```
import time
import multiprocessing as mp
total=0
#ΥΠΟΛΟΓΙΣΜΟΣ ΤΩΝ  ΔΥΝΑΜΕΩΝ
def worker(a,q):
    global total
    local_sum=0
    for i in a:
        local_sum=local_sum + i**2
        local_sum=local_sum + i**3
    q.put(local_sum)

## main processing
if __name__ == "__main__":

    start_time=time.perf_counter()
    n=10**7
    seq=range(1,n+1)
    queue = mp.Queue()
    p1 = mp.Process(target=worker, args=(seq[0:n//2],queue))
    p2 = mp.Process(target=worker, args=(seq[n//2:n],queue))

    p1.start()
    p2.start()
    p1.join()
    p2.join()
    #ΑΘΡΟΙΣΜΑ ΟΥΡΩΝ
    total = queue.get() + queue.get()

    end_time=time.perf_counter()
    print("ΣΥΝΟΛΟ : ",total)
    print("Ο ΧΡΟΝΟΣ ΜΕ 2 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΟΥΡΑ ΕΙΝΑΙ : ",end_time-start_time,"sec")
```

ΣΥΝΟΛΟ : 250000083333408333335000000

Ο ΧΡΟΝΟΣ ΜΕ 2 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΟΥΡΑ ΕΙΝΑΙ : 1,04384459997527 sec

Με τέσσερις διεργασίες και ουρά

```
import time
import multiprocessing as mp

total=0

def worker(a,q):
    global total
    local_sum=0
    for i in a:
        local_sum=local_sum + i**2
        local_sum=local_sum + i**3
    q.put(local_sum)

## main processing
if __name__ == "__main__":

    start_time=time.perf_counter()
    n=10**7
    seq=range(1,n+1)
    queue = mp.Queue()
    p1 = mp.Process(target=worker, args=(seq[0:n//4],queue))
    p2 = mp.Process(target=worker, args=(seq[n//4:n//2],queue))
    p3 = mp.Process(target=worker, args=(seq[n//2:3*n//4],queue))
    p4 = mp.Process(target=worker, args=(seq[3*n//4:n],queue))
    p1.start()
    p2.start()
    p3.start()
    p4.start()
    p1.join()
    p2.join()
    p3.join()
    p4.join()

    total = queue.get() + queue.get() + queue.get() + queue.get()

    end_time=time.perf_counter()

    print("ΣΥΝΟΛΟ : ",total)
    print("Ο ΧΡΟΝΟΣ ΜΕ 4 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΟΥΡΑ ΕΙΝΑΙ : ",end_time-start_time,"sec")
```

ΣΥΝΟΛΟ : 250000083333408333335000000

Ο ΧΡΟΝΟΣ ΜΕ 4 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΟΥΡΑ ΕΙΝΑΙ: 0,728794800117611 sec

Με οχτώ διεργασίες και ουρά

```
import time
import multiprocessing as mp
total=0
def worker(a,q):
    global total
    local_sum=0
    for i in a:
        local_sum=local_sum + i**2
        local_sum=local_sum + i**3
    q.put(local_sum)

## main processing
if __name__ == "__main__":
    start_time=time.perf_counter()
    n=10**7
    seq=range(1,n+1)
    queue = mp.Queue()
    p1 = mp.Process(target=worker, args=(seq[0:n//8],queue))
    p2 = mp.Process(target=worker, args=(seq[n//8:n//4],queue))
    p3 = mp.Process(target=worker, args=(seq[n//4:3*n//8],queue))
    p4 = mp.Process(target=worker, args=(seq[3*n//8:n//2],queue))
    p5 = mp.Process(target=worker, args=(seq[n//2:5*n//8],queue))
    p6 = mp.Process(target=worker, args=(seq[5*n//8:6*n//8],queue))
    p7 = mp.Process(target=worker, args=(seq[6*n//8:7*n//8],queue))
    p8 = mp.Process(target=worker, args=(seq[7*n//8:n],queue))
    p1.start()
    p2.start()
    p3.start()
    p4.start()
    p5.start()
    p6.start()
    p7.start()
    p8.start()
    p1.join()
    p2.join()
    p3.join()
    p4.join()
    p5.join()
    p6.join()
    p7.join()
    p8.join()
    total = queue.get() + queue.get() + queue.get() + queue.get() + queue.get() + queue.get() + queue.get() + queue.get()
    end_time=time.perf_counter()
    print("ΣΥΝΟΛΟ : ",total)
    print("Ο ΧΡΟΝΟΣ ΜΕ 8 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΟΥΡΑ ΕΙΝΑΙ : ",end_time-start_time,"sec")
```

ΣΥΝΟΛΟ : 250000083333340833333500000 0

Ο ΧΡΟΝΟΣ ΜΕ 8 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΟΥΡΑ ΕΙΝΑΙ : 0,57165990001522 sec

Με δυο διεργασίες και δεξαμενές

```
import time
import multiprocessing as mp

def worker(a):
    local_sum=0
    for i in a:
        local_sum = local_sum + i**2
    return local_sum

def worker1(a):
    local_sum1=0
    for i in a:
        local_sum1 = local_sum1 + i**3
    return local_sum1

## main processing
if __name__ == "__main__":

    start_time=time.perf_counter()
    n=10**7
    seq=range(1,n+1)
    pool = mp.Pool(2)
    result = pool.map(worker, (seq[0:n//2],seq[n//2:n]))
    result1 = pool.map(worker1, (seq[0:n//2],seq[n//2:n]))
    pool.close()

    total = 0
    for n in result:
        total = total + n

    total1 = 0
    for n in result1:
        total1 = total1 + n

    end_time=time.perf_counter()
    print("ΣΥΝΟΛΟ : ",total+total1)
    print("Ο ΧΡΟΝΟΣ ΜΕ 2 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : ",end_time-start_time,"sec")
```

ΣΥΝΟΛΟ : 250000083333408333335000000

Ο ΧΡΟΝΟΣ ΜΕ 2 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : 1,24683149997144 sec

Με τέσσερις διεργασίες και δεξαμενές

```
import time
import multiprocessing as mp
def worker(a):
    local_sum=0
    for i in a:
        local_sum = local_sum + i**2
    return local_sum

def worker1(a):
    local_sum1=0
    for i in a:
        local_sum1 = local_sum1 + i**3
    return local_sum1

## main processing
if __name__ == "__main__":

    start_time=time.perf_counter()
    n=10**7
    seq=range(1,n+1)
    pool = mp.Pool(4)
    result = pool.map(worker, (seq[0:n//4],seq[n//4:n//2],seq[n//2:3*n//4],seq[3*n//4:n]))
    result1 = pool.map(worker1, (seq[0:n//4],seq[n//4:n//2],seq[n//2:3*n//4],seq[3*n//4:n]))
    pool.close()

    total = 0
    for n in result:
        total = total + n

    total1 = 0
    for n in result1:
        total1 = total1 + n

    end_time=time.perf_counter()
    print("ΣΥΝΟΛΟ : ",total+total1)
    print("Ο ΧΡΟΝΟΣ ΜΕ 4 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : ",end_time-start_time,"sec")
```

ΣΥΝΟΛΟ : 250000083333340833333500000 0

Ο ΧΡΟΝΟΣ ΜΕ 4 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : 0,782705199904739 sec

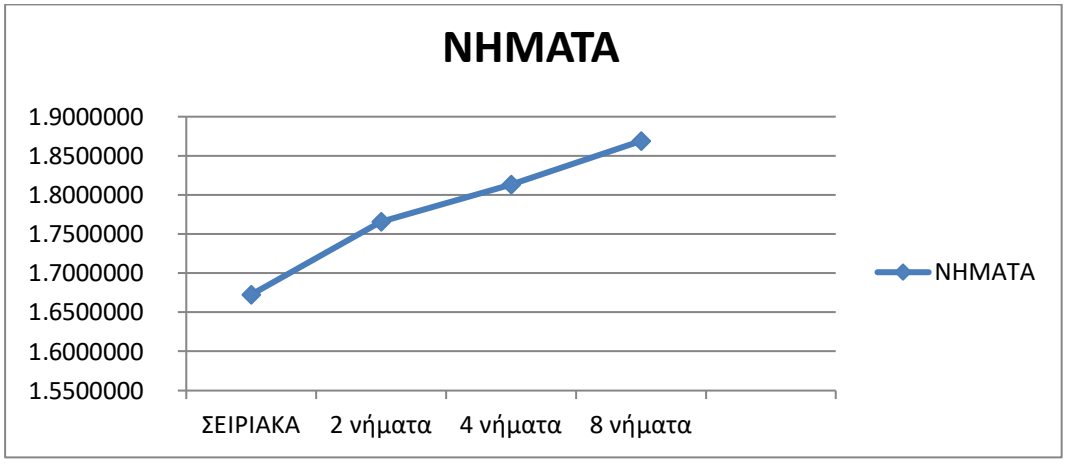
Με οχτώ διεργασίες και δεξαμενές

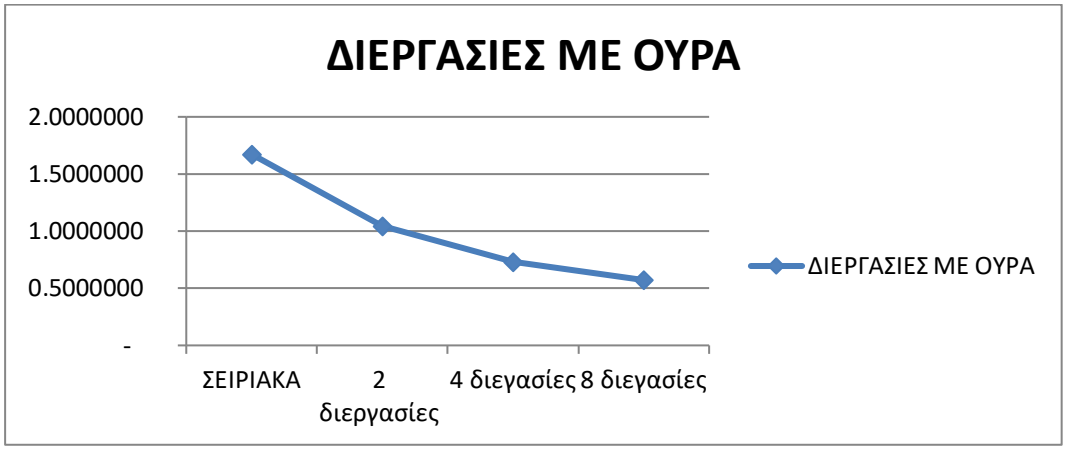
```
import time
import multiprocessing as mp
def worker(a):
    local_sum=0
    for i in a:
        local_sum = local_sum + i**2
    return local_sum

def worker1(a):
    local_sum1=0
    for i in a:
        local_sum1 = local_sum1 + i**3
    return local_sum1

## main processing
if __name__ == "__main__":

    start_time=time.perf_counter()
    n=10**7
    seq=range(1,n+1)
    pool = mp.Pool(8)
    result = pool.map(worker,
(seq[0:n//8],seq[n//8:2*n//8],seq[2*n//8:3*n//8],seq[3*n//8:4*n//8],seq[4*n//8:5*n//8],
seq[5*n//8:6*n//8],seq[6*n//8:7*n//8],seq[7*n//8:n]))
    result1 = pool.map(worker1,
(seq[0:n//8],seq[n//8:2*n//8],seq[2*n//8:3*n//8],seq[3*n//8:4*n//8],seq[4*n//8:5*n//8],
seq[5*n//8:6*n//8],seq[6*n//8:7*n//8],seq[7*n//8:n]))
    pool.close()

    total = 0
    for n in result:
        total = total + n

    total1 = 0
    for n in result1:
        total1 = total1 + n

    end_time=time.perf_counter()
    print("ΣΥΝΟΛΟ : ",total+total1)
    print("Ο ΧΡΟΝΟΣ ΜΕ 8 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : ",end_time-start_time,"sec")
```

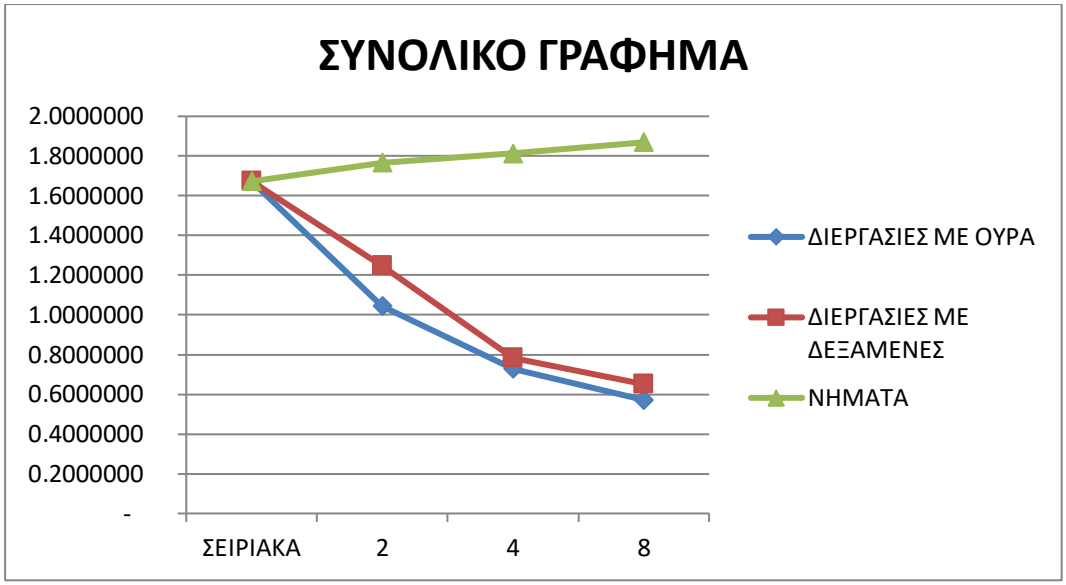
ΣΥΝΟΛΟ : 250000083333408333335000000

Ο ΧΡΟΝΟΣ ΜΕ 8 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ 0,651017599971964 sec

Γραφικές παραστάσεις







Πίνακας χρόνων

ΣΕΙΡΙΑΚΑ	1,6725978	
	ΝΗΜΑΤΑ	
2 νήματα	1,7660266	
4 νήματα	1,8132509	
8 νήματα	1,8689316	
	ΔΙΕΡΓΑΣΙΕΣ ΜΕ ΟΥΡΑ	ΔΙΕΡΓΑΣΙΕΣ ΜΕ ΔΕΞΑΜΕΝΕΣ
2 διεργασίες	1,0438446	1,2468315
4 διεργασίες	0,7287948	0,7827052
8 διεργασίες	0,5716599	0,6510176

Από τα παραπάνω αποτελέσματα συμπεραίνομαι ότι ο αλγόριθμος μας είναι cpu bound. Επειδή τα αποτελέσματα στους χρόνους των νημάτων είναι σχεδόν ιδία είτε με 2 είτε με 4 είτε με 8 νήμα. Αντίθετα τα αποτελέσματα στους χρόνους των διεργασιών μειώνονται όσο αυξάνονται το πλήθος των διεργασιών. Όποτε αυξάνεται η αποδοτικότητα του αλγορίθμου όσο αυξάνονται και οι διεργασίες.

Python CPU Παράδειγμα 2

Παρακάτω θα υλοποιήσουμε ένα παράδειγμα που θα εμφανίζουμε τους Armstrong αριθμούς μιας ακολουθίας. Η δόκιμη θα γίνει σειριακά ,με νήματα και παράλληλα .Στο τέλος θα συγκρίνουμε τους χρόνους για να δούμε την αποδοτικότητα του κάθε αλγορίθμου.

Σειριακά

```
import time
def is_armstrong_number(num):
    num_str = str(num)
    num_digits = len(num_str)
    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)
    return armstrong_sum == num

def find_armstrong_numbers(n):
    return list(filter(is_armstrong_number, range(1,n + 1)))
#############   main   #############
start_time = time.perf_counter()
n = 10**7
armstrong_numbers = find_armstrong_numbers(n)
end_time = time.perf_counter()
print("Οι Armstrong αριθμοί μέχρι το 10^7 είναι:", armstrong_numbers)
print("Ο ΧΡΟΝΟΣ ΣΕΙΡΙΑΚΑ ΕΙΝΑΙ : ",end_time-start_time,"sec")
```

Οι Armstrong αριθμοί μέχρι το 10^7 είναι: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474, 54748, 92727, 93084, 548834, 1741725, 4210818, 9800817, 9926315]

Ο ΧΡΟΝΟΣ ΣΕΙΡΙΑΚΑ ΕΙΝΑΙ : 16.248907000000145 sec

Με δυο διεργασίες

```
import time
import multiprocessing as mp

def is_armstrong_number(num):
    num_str = str(num)
    num_digits = len(num_str)
    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)
    return armstrong_sum == num

def find_armstrong_numbers_range(args):
    start, end = args
    armstrong_numbers_range = []
    for num in range(start, end + 1):
        if is_armstrong_number(num):
            armstrong_numbers_range.append(num)
    return armstrong_numbers_range
##############   main   #############
if __name__ == "__main__":
    start_time = time.perf_counter()
    n = 10**7
    seq = range(1, n + 1)
    pool = mp.Pool(2)

    results = pool.map(find_armstrong_numbers_range, [(1, n // 2), (n //
2 , n)])

    pool.close()

    armstrong_numbers = results[0] + results[1]

    end_time = time.perf_counter()

    print("Ο ΧΡΟΝΟΣ ΜΕ 2 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : ", end_time -
start_time, "sec")
    print("ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ:",
armstrong_numbers)
```

Ο ΧΡΟΝΟΣ ΜΕ 2 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : 8.763463700000102 sec

ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474, 54748, 92727, 93084, 548834, 1741725, 4210818, 9800817, 9926315]

Με τέσσερις διεργασίες

```
import time
import multiprocessing as mp

def is_armstrong_number(num):
    num_str = str(num)
    num_digits = len(num_str)
    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)
    return armstrong_sum == num

def find_armstrong_numbers_range(args):
    start, end = args
    armstrong_numbers_range = []
    for num in range(start, end + 1):
        if is_armstrong_number(num):
            armstrong_numbers_range.append(num)
    return armstrong_numbers_range
#############   main   #############
if __name__ == "__main__":
    start_time = time.perf_counter()
    n = 10**7
    seq = range(1, n + 1)
    pool = mp.Pool(4)

    results = pool.map(find_armstrong_numbers_range, [(1,n//4), (n//4,
n//2), (n//2,3*n//4), (3*n//4,n)])

    pool.close()

    armstrong_numbers = results[0] + results[1] + results[2] + results[3]

    end_time = time.perf_counter()
    print("Ο ΧΡΟΝΟΣ ΜΕ 4 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : ", end_time -
start_time, "sec")
    print("ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ:",
armstrong_numbers)
```

Ο ΧΡΟΝΟΣ ΜΕ 4 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : 4.960626200000661 sec

ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474, 54748, 92727, 93084, 548834, 1741725, 4210818, 9800817, 9926315]

Με οχτώ διεργασίες

```
import time
import multiprocessing as mp

def is_armstrong_number(num):
    num_str = str(num)
    num_digits = len(num_str)
    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)
    return armstrong_sum == num

def find_armstrong_numbers_range(args):
    start, end = args
    armstrong_numbers_range = []
    for num in range(start, end + 1):
        if is_armstrong_number(num):
            armstrong_numbers_range.append(num)
    return armstrong_numbers_range
#############   main   #############
if __name__ == "__main__":
    start_time = time.perf_counter()
    n = 10**7
    seq = range(1, n + 1)
    pool = mp.Pool(8)

    results = pool.map(find_armstrong_numbers_range, [(1,n//8),
                                                      (n//8, 2*n//8), (2*n//8,3*n//8),
                                                      (3*n//8,4*n//8), (4*n//8,5*n//8),
                                                      (5*n//8,6*n//8), (6*n//8,7*n//8),
                                                      (7*n//8,n)])

    pool.close()

    armstrong_numbers = results[0] + results[1] + results[2] + results[3] + results[4] +
results[5] + results[6] + results[7]

    end_time = time.perf_counter()

    print("Ο ΧΡΟΝΟΣ ΜΕ 8 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : ", end_time - start_time, "sec")
    print("ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ:", armstrong_numbers)
```

Ο ΧΡΟΝΟΣ ΜΕ 8 ΔΙΕΡΓΑΣΙΕΣ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : 3.198844100000315 sec

ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474, 54748, 92727, 93084, 548834, 1741725, 4210818, 9800817, 9926315]

Με δυο νήματα

```
import time
import concurrent.futures as cf

def is_armstrong_number(num):
    num_str = str(num)
    num_digits = len(num_str)
    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)
    return armstrong_sum == num

def find_armstrong_numbers_range(args):
    start, end = args
    armstrong_numbers_range = []
    for num in range(start, end + 1):
        if is_armstrong_number(num):
            armstrong_numbers_range.append(num)
    return armstrong_numbers_range
#############   main   #############
start_time = time.perf_counter()
n = 10**7
seq = range(1, n+1)
pool = cf.ThreadPoolExecutor(max_workers=2)

results = pool.map(find_armstrong_numbers_range, [(1, n // 2), (n // 2 , n)])
pool.shutdown()

armstrong_numbers = []
for result in results:
    armstrong_numbers.extend(result)

end_time = time.perf_counter()

print("Ο ΧΡΟΝΟΣ ΜΕ 2 ΝΗΜΑΤΑ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : ", end_time - start_time,
"sec")
print("ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ:", armstrong_numbers)
```

Ο ΧΡΟΝΟΣ ΜΕ 2 ΝΗΜΑΤΑ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : 16.888670599999386 sec

ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474, 54748, 92727, 93084, 548834, 1741725, 4210818, 9800817, 9926315]

Με τέσσερα νήματα

```
import time
import concurrent.futures as cf

def is_armstrong_number(num):
    num_str = str(num)
    num_digits = len(num_str)
    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)
    return armstrong_sum == num

def find_armstrong_numbers_range(args):
    start, end = args
    armstrong_numbers_range = []
    for num in range(start, end + 1):
        if is_armstrong_number(num):
            armstrong_numbers_range.append(num)
    return armstrong_numbers_range
##############   main   ##############
start_time = time.perf_counter()
n = 10**7
seq = range(1, n+1)
pool = cf.ThreadPoolExecutor(max_workers=4)

results = pool.map(find_armstrong_numbers_range, [(1,n//4), (n//4, n//2),
(n//2,3*n//4), (3*n//4,n)])

pool.shutdown()

armstrong_numbers = []
for result in results:
    armstrong_numbers.extend(result)

end_time = time.perf_counter()

print("Ο ΧΡΟΝΟΣ ΜΕ 4 ΝΗΜΑΤΑ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : ", end_time - start_time,
"sec")
print("ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ:", armstrong_numbers)
```

Ο ΧΡΟΝΟΣ ΜΕ 4 ΝΗΜΑΤΑ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : 16.963289299999815 sec

ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474, 54748, 92727, 93084, 548834, 1741725, 4210818, 9800817, 9926315]

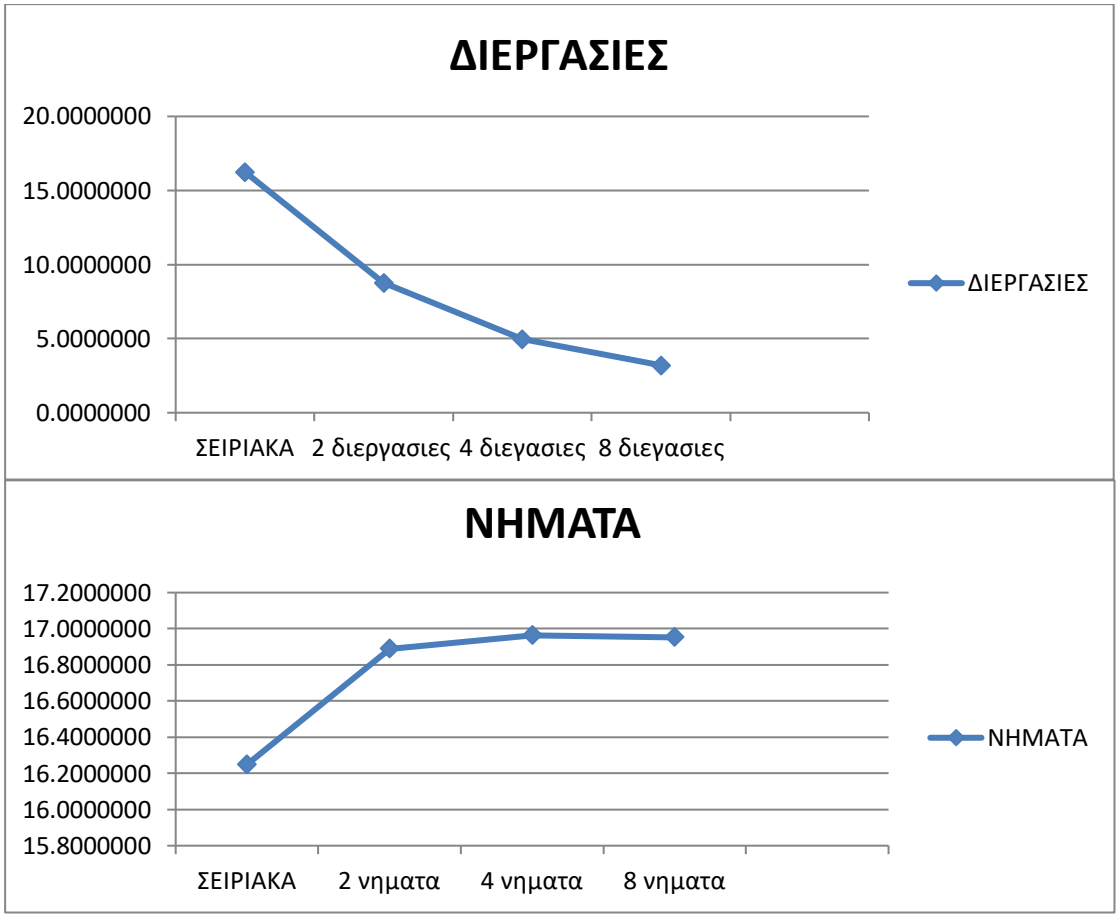
Με οχτώ νήματα

```
import time
import concurrent.futures as cf

def is_armstrong_number(num):
    num_str = str(num)
    num_digits = len(num_str)
    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)
    return armstrong_sum == num

def find_armstrong_numbers_range(args):
    start, end = args
    armstrong_numbers_range = []
    for num in range(start, end + 1):
        if is_armstrong_number(num):
            armstrong_numbers_range.append(num)
    return armstrong_numbers_range
#############   main   #############
start_time = time.perf_counter()
n = 10**7
seq = range(1, n+1)
pool = cf.ThreadPoolExecutor(max_workers=8)

results = pool.map(find_armstrong_numbers_range, [(1,n//8),
                                                  (n//8, 2*n//8), (2*n//8,3*n//8),
                                                  (3*n//8,4*n//8), (4*n//8,5*n//8),
                                                  (5*n//8,6*n//8), (6*n//8,7*n//8),
                                                  (7*n//8,n)])

pool.shutdown()

armstrong_numbers = []
for result in results:
    armstrong_numbers.extend(result)

end_time = time.perf_counter()

print("Ο ΧΡΟΝΟΣ ΜΕ 8 ΝΗΜΑΤΑ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : ", end_time - start_time, "sec")
print("ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ:", armstrong_numbers)
```

Ο ΧΡΟΝΟΣ ΜΕ 8 ΝΗΜΑΤΑ ΚΑΙ ΔΕΞΑΜΕΝΕΣ ΕΙΝΑΙ : 16.952636600000005 sec

ΟΙ ARMSTRONG ΑΡΙΘΜΟΙ ΑΠΟ ΤΟ 1 ΜΕΧΡΙ ΤΟ 10^7 ΕΙΝΑΙ: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474, 54748, 92727, 93084, 548834, 1741725, 4210818, 9800817, 9926315]

Γραφικές παραστάσεις



Πίνακας χρόνων

ΣΕΙΡΙΑΚΑ	16,2489070
	ΝΗΜΑΤΑ
2 νήματα	16,8886706
4 νήματα	16,9632830
8 νήματα	16,9526366
	ΔΙΕΡΓΑΣΙΕΣ
2 διεργασίες	8,7634637
4 διεργασίες	4,9606262
8 διεργασίες	3,1988441

Από τα παραπάνω αποτελέσματα συμπεραίνομαι ότι ο αλγόριθμος μας είναι cpu bound. Επειδή τα αποτελέσματα στους χρόνους των νημάτων είναι σχεδόν ιδία είτε με 2 είτε με 4 είτε με 8 νήμα. Αντίθετα τα αποτελέσματα στους χρόνους των διεργασιών μειώνονται όσο αυξάνονται το πλήθος των διεργασιών. Όποτε αυξάνεται η αποδοτικότητα του αλγορίθμου όσο αυξάνονται και οι διεργασίες.

ΚΕΦΑΛΑΙΟ 8

Python I/O Παράδειγμα 1

Στο παρακάτω παράδειγμα χρησιμοποιούμε την εντολή sleep για να καθυστερήσουμε την εκτέλεση ενός προγράμματος για ένα συγκεκριμένο χρονικό διάστημα. Όταν πρόκειται για παράλληλο προγραμματισμό με νήματα και διεργασίες, μπορεί να χρησιμοποιηθεί για να δημιουργήσει καθυστέρηση μεταξύ των ενεργειών των νημάτων ή των διεργασιών.

Σειριακά

```
import time

def work(n):
    for _ in range(n):
        time.sleep(1)

#### main ####
start_time = time.perf_counter()

work(8)

end_time = time.perf_counter()
print("Ο ΧΡΟΝΟΣ ΣΕΙΡΙΑΚΑ ΕΙΝΑΙ:", end_time - start_time, "sec")
```

Ο ΧΡΟΝΟΣ ΣΕΙΡΙΑΚΑ ΕΙΝΑΙ: 8.00386870000511 sec

Με δυο νήματα

```
import time
import threading as th

def work(n):
    for _ in range(n):
        time.sleep(1)

#### main ####
start_time = time.perf_counter()

t1 = th.Thread(target=work,args=(4,))
t2 = th.Thread(target=work,args=(4,))

t1.start()
t2.start()

t1.join()
t2.join()

end_time = time.perf_counter()
print("Ο ΧΡΟΝΟΣ ΜΕ 2 ΝΗΜΑΤΑ ΕΙΝΑΙ:", end_time - start_time, "sec" )
```

Ο ΧΡΟΝΟΣ ΜΕ 2 ΝΗΜΑΤΑ ΕΙΝΑΙ: 4.00322399998549 sec

Με τέσσερα νήματα

```
import time
import threading as th

def work(n):
    for _ in range(n):
        time.sleep(1)

#### main ####
start_time = time.perf_counter()

t1 = th.Thread(target=work,args=(2,))
t2 = th.Thread(target=work,args=(2,))
t3 = th.Thread(target=work,args=(2,))
t4 = th.Thread(target=work,args=(2,))
t1.start()
t2.start()
t3.start()
t4.start()
t1.join()
t2.join()
t3.join()
t4.join()

end_time = time.perf_counter()
print("Ο ΧΡΟΝΟΣ ΜΕ 4 ΝΗΜΑΤΑ ΕΙΝΑΙ:", end_time - start_time, "sec" )
```

Ο ΧΡΟΝΟΣ ΜΕ 4 ΝΗΜΑΤΑ ΕΙΝΑΙ: 2.00307670002803 sec

Με οχτώ νήματα

```
import time
import threading as th

def work(n):
    for _ in range(n):
        time.sleep(1)
#### main ####
start_time = time.perf_counter()

t1 = th.Thread(target=work,args=(1,))
t2 = th.Thread(target=work,args=(1,))
t3 = th.Thread(target=work,args=(1,))
t4 = th.Thread(target=work,args=(1,))
t5 = th.Thread(target=work,args=(1,))
t6 = th.Thread(target=work,args=(1,))
t7 = th.Thread(target=work,args=(1,))
t8 = th.Thread(target=work,args=(1,))
t1.start()
t2.start()
t3.start()
t4.start()
t5.start()
t6.start()
t7.start()
t8.start()
t1.join()
t2.join()
t3.join()
t4.join()
t5.join()
t6.join()
t7.join()
t8.join()

end_time = time.perf_counter()
print("Ο ΧΡΟΝΟΣ ΜΕ 8 ΝΗΜΑΤΑ ΕΙΝΑΙ:", end_time - start_time, "sec" )
```

Ο ΧΡΟΝΟΣ ΜΕ 8 ΝΗΜΑΤΑ ΕΙΝΑΙ: 1.00273099995683 sec

Με δυο διεργασίες

```
import time
import multiprocessing as mp

def work(n):
    for _ in range(n):
        time.sleep(1)

if __name__ == "__main__":
    start_time = time.perf_counter()
    p1 = mp.Process(target=work,args=(4,))
    p2 = mp.Process(target=work,args=(4,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    end_time = time.perf_counter()
    print("Ο ΧΡΟΝΟΣ ΜΕ 2 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ:", end_time - start_time, "sec")
```

Ο ΧΡΟΝΟΣ ΜΕ 2 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ: 4.11244389996863 sec

Με τέσσερις διεργασίες

```
import time
import multiprocessing as mp

def work(n):
    for _ in range(n):
        time.sleep(1)

if __name__ == "__main__":
    start_time = time.perf_counter()
    p1 = mp.Process(target=work,args=(2,))
    p2 = mp.Process(target=work,args=(2,))
    p3 = mp.Process(target=work,args=(2,))
    p4 = mp.Process(target=work,args=(2,))
    p1.start()
    p2.start()
    p3.start()
    p4.start()
    p1.join()
    p2.join()
    p3.join()
    p4.join()
    end_time = time.perf_counter()
    print("Ο ΧΡΟΝΟΣ ΜΕ 4 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ:", end_time - start_time, "sec")
```

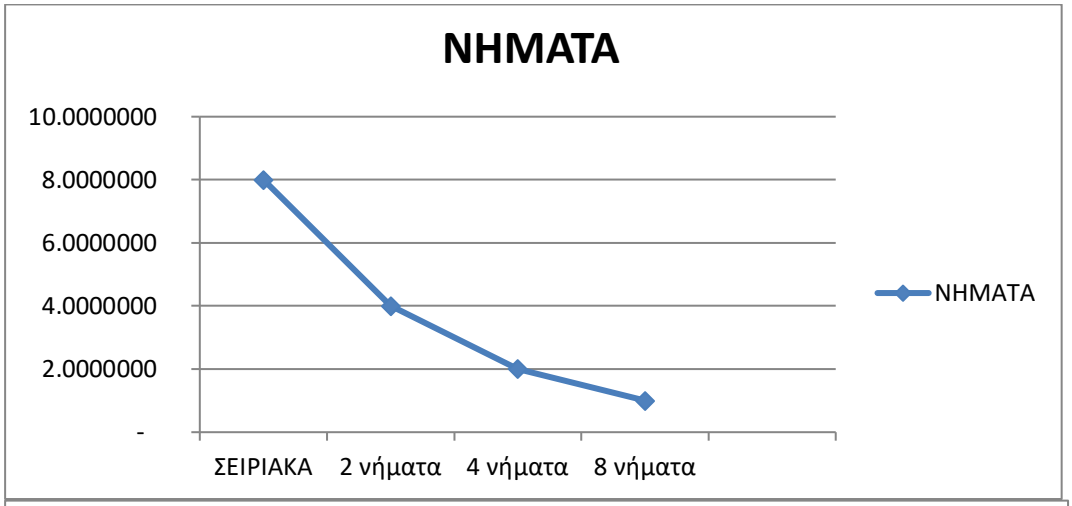
Ο ΧΡΟΝΟΣ ΜΕ 4 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ: 2.19097640004474 sec

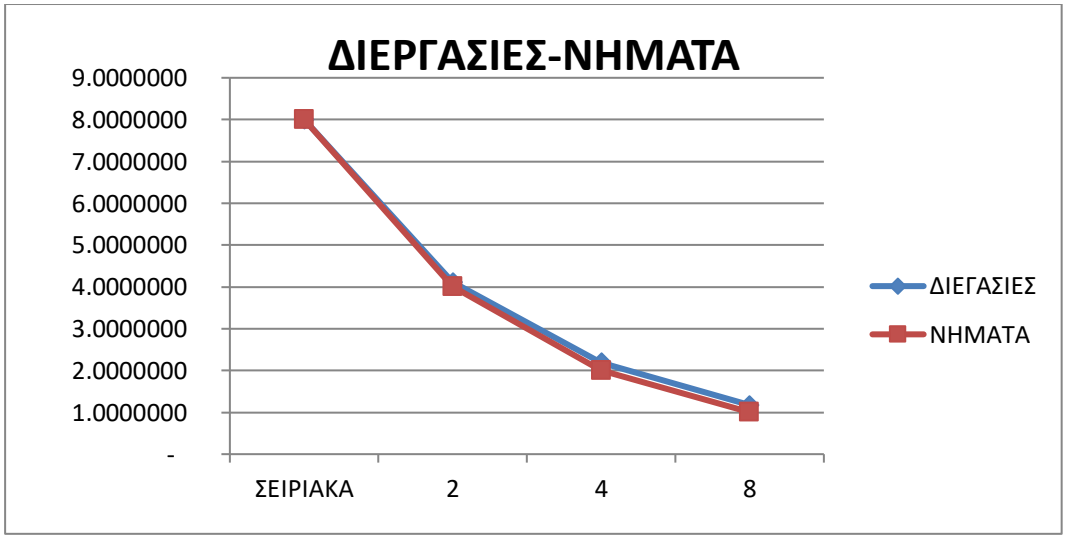
Με οχτώ διεργασίες

```
import time
import multiprocessing as mp

def work(n):
    for _ in range(n):
        time.sleep(1)

if __name__ == "__main__":
    start_time = time.perf_counter()
    p1 = mp.Process(target=work,args=(1,))
    p2 = mp.Process(target=work,args=(1,))
    p3 = mp.Process(target=work,args=(1,))
    p4 = mp.Process(target=work,args=(1,))
    p5 = mp.Process(target=work,args=(1,))
    p6 = mp.Process(target=work,args=(1,))
    p7 = mp.Process(target=work,args=(1,))
    p8 = mp.Process(target=work,args=(1,))
    p1.start()
    p2.start()
    p3.start()
    p4.start()
    p5.start()
    p6.start()
    p7.start()
    p8.start()
    p1.join()
    p2.join()
    p3.join()
    p4.join()
    p5.join()
    p6.join()
    p7.join()
    p8.join()

    end_time = time.perf_counter()
    print("Ο ΧΡΟΝΟΣ ΜΕ 8 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ:", end_time - start_time, "sec")
```

Ο ΧΡΟΝΟΣ ΜΕ 8 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ: 1.17441000009421 sec

Γραφικές παραστάσεις





Πίνακας χρόνων

ΣΕΙΡΙΑΚΑ	8,0038687
	ΝΗΜΑΤΑ
2 νήματα	4,0032240
4 νήματα	2,0030767
8 νήματα	1,0027310
	ΔΙΕΡΓΑΣΙΕΣ
2 διεργασίες	4,1124439
4 διεγασίες	2,1909764
8 διεγασίες	1,1744100

Από τους παραπάνω χρόνους συμπεραίνουμε ότι το πρόγραμμα μας είναι I/O bound. Επειδή την καλύτερη χρονική απόδοση την έχει με τα νήματα.

Python I/O παράδειγμα 2

Στο επόμενο παράδειγμα θα κάνουμε αιτήματα (requests) σε τέσσερις σελίδες της Amazon και τέσσερις της Google με διαφορετικές καταλήξεις (.com, .de, .fr, .it, .gr) και θα μετρήσουμε το χρόνο σειριακά , παράλληλα και με νήματα.

Σειριακα

```
import time, requests

#### main ####
start_time = time.perf_counter()
urls = ["https://www.amazon.com/", "https://www.amazon.de/",
"https://www.amazon.fr/", "https://www.amazon.it/",
        "https://www.google.com/","https://www.google.gr/",
"https://www.google.de/","https://www.google.fr/"]

for url in urls:
    requests.get(url)

end_time = time.perf_counter()
print("Ο ΧΡΟΝΟΣ ΣΕΙΡΙΑΚΑ ΕΙΝΑΙ:", end_time - start_time,"sec")
```

Ο ΧΡΟΝΟΣ ΣΕΙΡΙΑΚΑ ΕΙΝΑΙ: 5.85912689997348 sec

Με δυο νήματα

```
import threading as th
import time,requests
def request(urls):
    for url in urls:
        x = requests.get(url)

##### main ########
start_time = time.perf_counter()

urls = ["https://www.amazon.com/", "https://www.amazon.de/",
"https://www.amazon.fr/", "https://www.amazon.it/",
        "https://www.google.com/","https://www.google.gr/",
"https://www.google.de/","https://www.google.fr/"]

t1 = th.Thread(target=request, args=(urls[0:4],))
t2 = th.Thread(target=request, args=(urls[4:8],))
t1.start()
t2.start()
t1.join()
t2.join()
end_time = time.perf_counter()
total_time = end_time - start_time
print("Ο ΧΡΟΝΟΣ ΜΕ 2 ΝΗΜΑΤΑ ΕΙΝΑΙ: ", total_time,"sec")
```

Ο ΧΡΟΝΟΣ ΜΕ 2 ΝΗΜΑΤΑ ΕΙΝΑΙ: 3.43382129992824 sec

Με τέσσερα νήματα

```
import threading as th
import time,requests
def request(urls):
    for url in urls:
        x = requests.get(url)

##### main ########
start_time = time.perf_counter()

urls = ["https://www.amazon.com/", "https://www.amazon.de/",
"https://www.amazon.fr/", "https://www.amazon.it/",
        "https://www.google.com/","https://www.google.gr/",
"https://www.google.de/","https://www.google.fr/"]


t1 = th.Thread(target=request, args=(urls[0:2],))
t2 = th.Thread(target=request, args=(urls[2:4],))
t3 = th.Thread(target=request, args=(urls[4:6],))
t4 = th.Thread(target=request, args=(urls[6:8],))
t1.start()
t2.start()
t3.start()
t4.start()
t1.join()
t2.join()
t3.join()
t4.join()

end_time = time.perf_counter()
total_time = end_time - start_time
print("Ο ΧΡΟΝΟΣ ΜΕ 4 ΝΗΜΑΤΑ ΕΙΝΑΙ: ", total_time,"sec")
```

Ο ΧΡΟΝΟΣ ΜΕ 4 ΝΗΜΑΤΑ ΕΙΝΑΙ: 1.98016559996176 sec

Με οχτώ νήματα

```
import threading as th
import time,requests

def request(urls):
    for url in urls:
        x = requests.get(url)

##### main ########
start_time = time.perf_counter()

urls = ["https://www.amazon.com/", "https://www.amazon.de/",
"https://www.amazon.fr/", "https://www.amazon.it/",
        "https://www.google.com/","https://www.google.gr/",
"https://www.google.de/","https://www.google.fr/"]


t1 = th.Thread(target=request, args=(urls[0:1],))
t2 = th.Thread(target=request, args=(urls[1:2],))
t3 = th.Thread(target=request, args=(urls[2:3],))
t4 = th.Thread(target=request, args=(urls[3:4],))
t5 = th.Thread(target=request, args=(urls[4:5],))
t6 = th.Thread(target=request, args=(urls[5:6],))
t7 = th.Thread(target=request, args=(urls[6:7],))
t8 = th.Thread(target=request, args=(urls[7:8],))
t1.start()
t2.start()
t3.start()
t4.start()
t5.start()
t6.start()
t7.start()
t8.start()
t1.join()
t2.join()
t3.join()
t4.join()
t5.join()
t6.join()
t7.join()
t8.join()

end_time = time.perf_counter()
total_time = end_time - start_time
print("Ο ΧΡΟΝΟΣ ΜΕ 8 ΝΗΜΑΤΑ ΕΙΝΑΙ: ", total_time,"sec")
```

Ο ΧΡΟΝΟΣ ΜΕ 8 ΝΗΜΑΤΑ ΕΙΝΑΙ: 1.40140039997641sec

Με δυο διεργασίες

```
import multiprocessing as mp
import time,requests

def request(urls):
    for url in urls:
        x = requests.get(url)

##### main ########
if __name__ == "__main__":
    start_time = time.perf_counter()

    urls = ["https://www.amazon.com/", "https://www.amazon.de/",
"https://www.amazon.fr/", "https://www.amazon.it/",
        "https://www.google.com/","https://www.google.gr/",
"https://www.google.de/","https://www.google.fr/"]

    p1 = mp.Process(target=request, args=(urls[0:4],))
    p2 = mp.Process(target=request, args=(urls[4:8],))

    p1.start()
    p2.start()

    p1.join()
    p2.join()

    end_time = time.perf_counter()
    total_time = end_time - start_time
    print("Ο ΧΡΟΝΟΣ ΜΕ 2 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ", total_time,"sec")
```

Ο ΧΡΟΝΟΣ ΜΕ 2 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ 3.78281579993199 sec

Με τέσσερις διεργασίες

```
import multiprocessing as mp
import time,requests

def request(urls):
    for url in urls:
        x = requests.get(url)

##### main ########
if __name__ == "__main__":
    start_time = time.perf_counter()

    urls = ["https://www.amazon.com/", "https://www.amazon.de/",
"https://www.amazon.fr/", "https://www.amazon.it/",
        "https://www.google.com/","https://www.google.gr/",
"https://www.google.de/","https://www.google.fr/"]

    p1 = mp.Process(target=request, args=(urls[0:2],))
    p2 = mp.Process(target=request, args=(urls[2:4],))
    p3 = mp.Process(target=request, args=(urls[4:6],))
    p4 = mp.Process(target=request, args=(urls[6:8],))
    p1.start()
    p2.start()
    p3.start()
    p4.start()
    p1.join()
    p2.join()
    p3.join()
    p4.join()

    end_time = time.perf_counter()
    total_time = end_time - start_time
    print("Ο ΧΡΟΝΟΣ ΜΕ 4 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ", total_time,"sec")
```

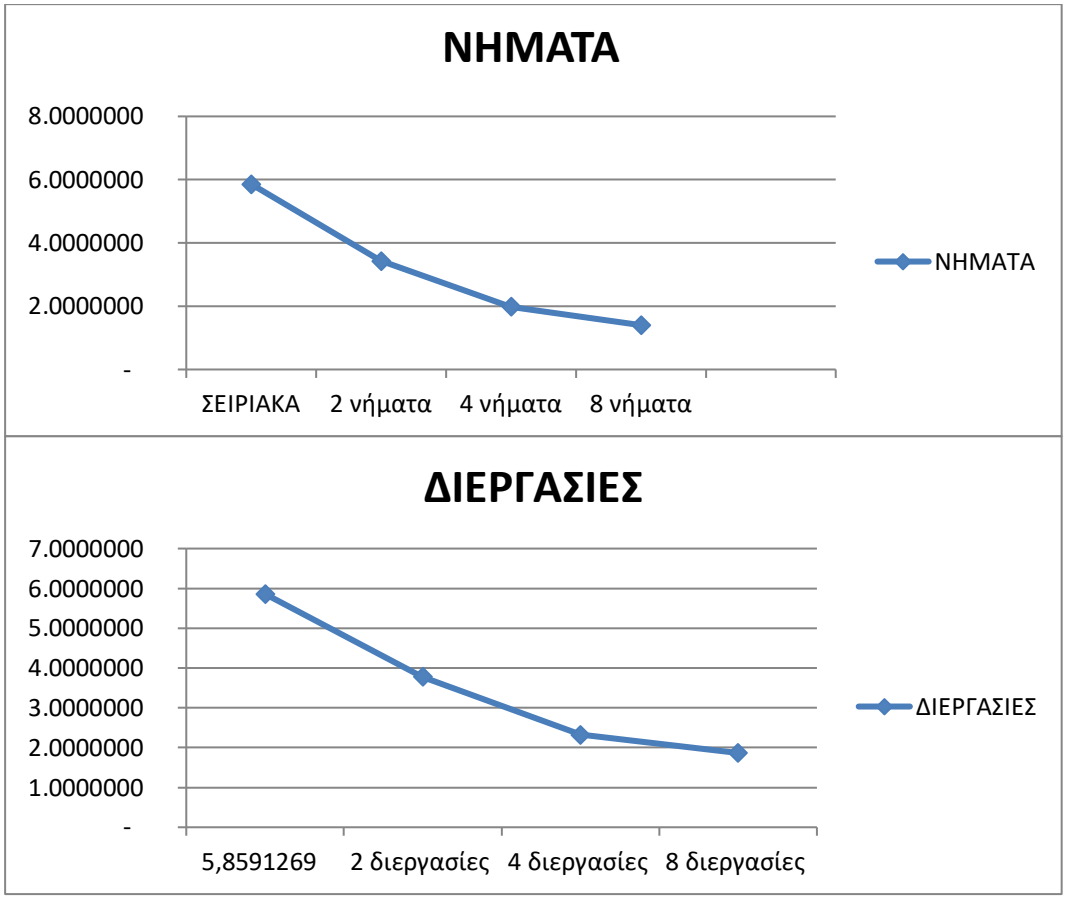
Ο ΧΡΟΝΟΣ ΜΕ 4 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ 2.3230991999153 sec

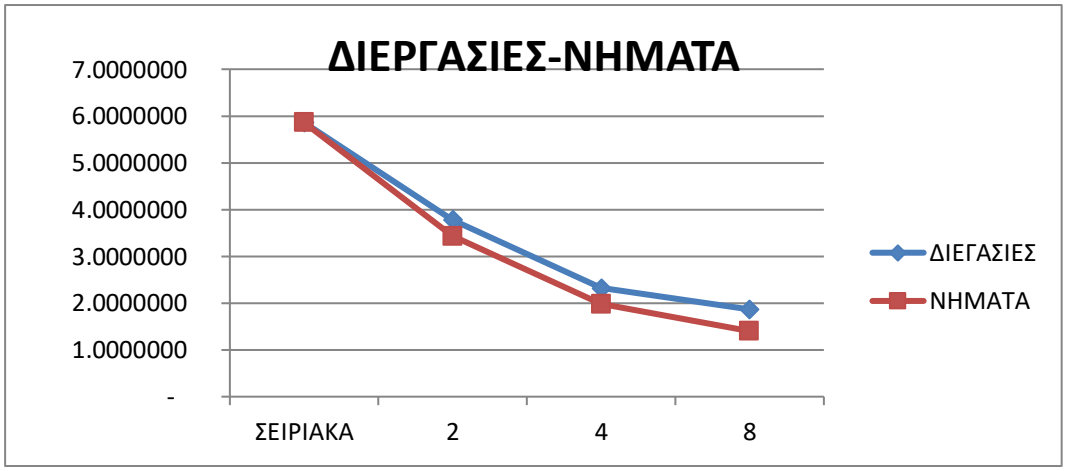
Με οχτώ διεργασίες

```
import multiprocessing as mp
import time,requests

def request(urls):
    for url in urls:
        x = requests.get(url)

##### main ########
if __name__ == "__main__":
    start_time = time.perf_counter()

    urls = ["https://www.amazon.com/", "https://www.amazon.de/",
"https://www.amazon.fr/", "https://www.amazon.it/",
        "https://www.google.com/","https://www.google.gr/",
"https://www.google.de/","https://www.google.fr/"]

    p1 = mp.Process(target=request, args=(urls[0:1],))
    p2 = mp.Process(target=request, args=(urls[1:2],))
    p3 = mp.Process(target=request, args=(urls[2:3],))
    p4 = mp.Process(target=request, args=(urls[3:4],))
    p5 = mp.Process(target=request, args=(urls[4:5],))
    p6 = mp.Process(target=request, args=(urls[5:6],))
    p7 = mp.Process(target=request, args=(urls[6:7],))
    p8 = mp.Process(target=request, args=(urls[7:8],))
    p1.start()
    p2.start()
    p3.start()
    p4.start()
    p5.start()
    p6.start()
    p7.start()
    p8.start()
    p1.join()
    p2.join()
    p3.join()
    p4.join()
    p5.join()
    p6.join()
    p7.join()
    p8.join()

    end_time = time.perf_counter()
    total_time = end_time - start_time
    print("Ο ΧΡΟΝΟΣ ΜΕ 8 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ", total_time,"sec")
```

Ο ΧΡΟΝΟΣ ΜΕ 8 ΔΙΕΡΓΑΣΙΕΣ ΕΙΝΑΙ 1.8657213000115 sec

Γραφικές παραστάσεις





Πίνακας χρόνων

ΣΕΙΡΙΑΚΑ	5,8591269
	ΝΗΜΑΤΑ
2 νήματα	3,4338213
4 νήματα	1,9801656
8 νήματα	1,4014004
	ΔΙΕΡΓΑΣΙΕΣ
2 διεργασίες	3,7828158
4 διεργασίες	2,3230992
8 διεργασίες	1,8657213

Από τους παραπάνω χρόνους συμπεραίνουμε ότι το πρόγραμμα μας είναι I/O bound. Επειδή την καλύτερη χρονική απόδοση την έχει με τα νήματα.

Βιβλιογραφία

- Zaccone, G. (2015). *Python parallel programming cookbook*. Packt Publishing Ltd.
- Beazley, D. (2010, February). Understanding the python gil. In *PyCON Python Conference. Atlanta, Georgia* (pp. 1-62).
- Λειτουργικά Συστήματα, 8η Έκδοση, Stallings William

- Λειτουργικά Συστήματα 9η Εκδ., Abraham Silberschatz, Peter Baer Galvin, Greg Gagne
- Σημειώσεις Παράλληλου Προγραμματισμού, Βαρσάμης Δημήτρης
- https://docs.python.org/2/library/multiprocessing.html
- https://github.com/python/cpython/blob/main/Lib/concurrent/futures/process.py
- https://github.com/python/cpython/blob/main/Lib/concurrent/futures/thread.py
- https://www.python-course.eu/threads.php
- https://www.bogotobogo.com/python/Multithread/python_multithreading_Synchronization_Semaphore_Objects_Thread_Pool.php