**ΕΡΓΑΣΙΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ**

Περιεχόμενα

[Μέρος Α: Εισαγωγή 2](#_Toc74243492)

[Μέρος Β: Αναλυτική Τεκμηρίωση Παράλληλων Υλοποιήσεων 3](#_Toc74243493)

[*OpenMP:* 3](#_Toc74243494)

[*OpenMP tasks:* 4](#_Toc74243495)

[*MPI:* 5](#_Toc74243496)

[*MPI and OpenMP hybrid:* 6](#_Toc74243497)

[Μέρος Γ: Αποτελέσματα 7](#_Toc74243498)

[*OpenMP:* 7](#_Toc74243499)

[*OpenMP tasks:* 7](#_Toc74243500)

[*MPI:* 7](#_Toc74243501)

[*MPI and OpenMP hybrid:* 7](#_Toc74243502)

[Μέρος Δ: Συμπεράσματα 8](#_Toc74243503)

# Μέρος Α: Εισαγωγή

Όλοι οι κώδικες έχουν τρέξει στο ίδιο υπολογιστικό σύστημα και για τον σκοπό αυτό αναπτύχθηκαν, πέρα από τις παράλληλες υλοποιήσεις που μας ζητούνται δύο επιπλέον αρχεία (benchmark.c και graph\_benchmark.py), ώστε να είμαστε σε θέση να τρέξουμε αυτόματα τους παράλληλους κώδικες. Το πρώτο αρχείο λοιπόν τρέχει για διάφορους αριθμούς νημάτων τους κώδικές (αν πρόκειται για MPI κώδικα, για διάφορο αριθμό communicator), με ένα διάστημα sleep, ώστε να μην υπερθερμαίνεται το laptop στο οποίο το τρέχαμε και άρα να αποφύγουμε throttle, οπότε και λανθασμένες μετρήσεις στους χρόνους.

Πέρα από την υπερθέρμανση φροντίσαμε να μην λειτουργούν διεργασίες, πέρα από τις βασικές. Το μηχάνημα στο οποίο τρέξαμε τους παράλληλους κώδικες είναι laptop με dual boot (Windows, ubuntu) και προφανώς χρησιμοποιήθηκε το 2ο. Έχει 6 cores (12 νήματα) και 16GB RAM. Η αρχιτεκτονική του φαίνεται παρακάτω:

Σημαντικό επίσης είναι το γεγονός ότι για τους κώδικες OpenMP (και με tasks) τρέξαμε παραπάνω μετρήσεις και βγάλαμε ένα μέσο όρο των αποτελεσμάτων, ώστε να βεβαιωθούμε ότι δεν έχουμε πολύ μεγάλες αποκλίσεις από τις πραγματικές τιμές, που οφείλονται σε εξωγενείς παράγοντες. Κάτι τέτοιο δεν έγινε στους MPI κώδικες, λόγω έλλειψης χρόνου, μιας και για να τρέξουμε τα benchmark απαιτείται αρκετός.

Ενημερώθηκε τέλος και το Makefile που μας δόθηκε, με όλα τα ονόματα των υλοποιήσεών μας, ώστε με ένα απλό make να γίνονται compile. Τα warnings τα οποία τυπώνονται δεν είναι σημαντικά, αφού αφορούν κυρίως uninitialized μεταβλητές, που όμως στη πραγματικότητα έχουν γίνει initialized, μόνο για το thread ή το MPI rank που θα τα χρησιμοποιήσει.

# Μέρος Β: Αναλυτική Τεκμηρίωση Παράλληλων Υλοποιήσεων

## OpenMP:

Σε αυτή την παράλληλη υλοποίηση, όπως και στις υπόλοιπες που θα παρουσιαστούν παρακάτω, παραλληλοποιήσαμε την main του αρχείου που μας δόθηκε, αφού εκεί υπάρχει ένα for loop με 128\*1,024 = 131,072 επαναλήψεις που έπρεπε να μοιραστούν στα threads. Έπειτα από παρατήρηση συμφοιτητή μας στο μάθημα και οδηγία που δόθηκε από τον καθηγητή, προστατέψαμε επίσης την global μεταβλητή funevals.

Καταρχάς, πριν ξεκινήσουμε οποιαδήποτε προσπάθεια παραλληλοποίησης του κώδικα, σκεφτήκαμε τα βήματα τα οποία θα ακολουθήσουμε προκειμένου να πετύχουμε τον στόχο μας και συνέχεια τα σημειώσαμε με TODO σχόλια πάνω στον σειριακό κώδικα ώστε να διευκολύνουμε τη διαδικασία. Τα βήματα που θέσαμε ήταν κλιμακωτά, ώστε να μπορούμε στο τέλος κάθε βήματος να ελέγξουμε ότι ο κώδικας λειτουργεί σωστά.

1. Αλλαγή της drand48() στην erand48() που είναι thread safe.
2. Παραλληλοποίηση της for loop (με τα trials), χωρίς κάποιο reduction και τύπωση των αποτελεσμάτων του κάθε thread, ώστε να βεβαιωθούμε ότι το καθένα έχει βρει ένα τοπικό ελάχιστο.
3. Manual reduction πάνω στις μεταβλητές best\_fx, best\_trial, best\_jj, best\_pt, με τη χρήση critical region, μετά το τέλος των υπολογισμών κάθε thread. Το reduction προφανώς έγινε με κριτήριο το ελάχιστο fx που έχει βρει κάθε thread. Μειώσαμε και τα barrier με nowait στο #pragma omp for.
4. Προστασία της global μεταβλητής funevals.

Πιο αναλυτικά για το Βήμα 4, είχαμε 2 επιλογές. Είτε να κλειδώναμε μέσω ενός atomic την μεταβλητή σε κάθε iteration, κάθε thread, το οποίο βέβαια, δεδομένου και ότι σε κάθε iteration η f μπορεί να καλείται και παραπάνω από μια φορά θα είναι ιδιαίτερα κοστοβόρο, από άποψη χρόνου. Η άλλη επιλογή που είχαμε είναι και αυτή που επιλέξαμε τελικά, να χρησιμοποιήσουμε μια βοηθητική μεταβλητή (\*local\_func\_evals), την οποία θα περνάμε ως argument (by reference) στην f και στις συναρτήσεις που καλούν την f. Αυτός ο pointer ουσιαστικά θα δείχνει σε μια θέση μνήμης, private για κάθε thread, την οποία θα χρησιμοποιούμε ως counter για το thread. Μόλις το thread ολοκληρώσει τους υπολογισμούς, αναθέτει την τιμή αυτού του counter στην μεταβλητή funevals και στη συνέχεια μέσω του openmp κάνουμε reduce(+:funevals).

Τέλος, κάναμε μερικές ακόμα αλλαγές στον κώδικα της main ώστε να αποθηκεύει τα αποτελέσματα σε κατάλληλο .txt αρχείο (Results/res\_omp.txt), που θα χρησιμοποιήσουμε στη συνέχεια για να εξάγουμε τα plot που μας ζητούνται.

## OpenMP tasks:

Η ίδια λογική ακολουθήθηκε και σε αυτή την περίπτωση παραλληλοποίησης. H κύρια διαφορά είναι ότι δεν βρήκαμε κάποιο τρόπο, ώστε το κάθε thread να μπορεί να αναλαμβάνει tasks τα οποία θα μεταβάλουν τα τοπικά ελάχιστα του συγκεκριμένου thread, ώστε να μπορέσουμε μετά, μέσω critical region να κάνουμε το reduction, όπως ακριβώς υλοποιήθηκε και στην προηγούμενη υλοποίηση. Μια λύση που βρέθηκε, είναι να μην «κλειδώνουμε» σε κάθε task το critical region, αλλά να ελέγχουμε αν το σημείο που το task μόλις υπολόγισε είναι καλύτερο από το σημείο που μπορεί να «δει» εκείνη τη στιγμή ως global ελάχιστο. Στην περίπτωση που δεν είναι το thread απλά κάνει “discard” την πληροφορία που μόλις υπολόγισε και συνεχίζει επιλέγοντας ένα νέο task από το task queue. Στην περίπτωση που το σημείο που υπολόγισε είναι καλύτερο ελάχιστο (περάσει δηλαδή το πρώτο if statement) μπορεί να μπει στο critical region. Όμως δεν θα πρέπει να ξεχνάμε ότι υπάρχει race condition στο πρώτο if (το εξωτερικό του critical) συνεπώς δύο thread μπορούν να περάσουν «ταυτόχρονα» τον έλεγχο αυτό. Το πρόβλημα περιγράφεται με παράδειγμα πάνω στον κώδικα και λύνεται απλώς με την εισαγωγή ενός ακόμα ελέγχου μέσα στο critical region.

Τα funevals προστατεύθηκαν με τον ίδιο τρόπο που προστατεύθηκαν στην προηγούμενη παράλληλη υλοποίηση (μέσω του pointer σε έναν μετρητή), όμως λόγω του προβλήματος που αναφέρουμε παραπάνω έπρεπε το κάθε task να ανανεώνει τον μετρητή. Αυτό το υλοποιήσαμε με τη χρήση ενός atomic. Μετατοπίσαμε δηλαδή το πρόβλημα στο υλικό περισσότερο. Το πρόβλημα βέβαια είναι ότι σε κάθε εκτέλεση task αυτό κλειδώνει, το οποίο δεν είναι ιδανικό αλλά δεν μπορέσαμε να βρούμε κάτι το οποίο να το αντιμετωπίζει.

Τέλος, προσθέσαμε μερικές γραμμές κώδικα ώστε να αποθηκεύουμε τα αποτελέσματα που παίρνουμε από κάθε εκτέλεση του παράλληλου κώδικα στο αρχείο Results/res\_omp\_tasks. Προφανώς μια βελτιστοποίηση που θα μπορούσαμε να υλοποιήσουμε στο πρόγραμμα είναι να thread-local minimization με χρήση των tasks και έπειτα να χρησιμοποιήσουμε αυτά τα thread-local ελάχιστα, ώστε να τα συγκρίνουμε και να εξάγουμε το ολικό ελάχιστο που υπολογίσαμε. Βρήκαμε μια δημοσίευση από ένα συνέδριο, που δεν προλάβαμε όμως να διαβάσουμε αναλυτικά και να υλοποιήσουμε [εδώ](https://www.osti.gov/servlets/purl/1582180):

[1] J. Ciesko et al., “Towards Task-Parallel Reductions in OpenMP,” in OpenMP: Heterogenous Execution and Data Movements, Cham: Springer International Publishing, 2015, pp. 189–201.

## MPI:

Όπως και στις υλοποιήσεις του OpenMP, έτσι και στο MPI επιλέξαμε να σκεφτούμε αρχικά τα βήματα που θα ακολουθήσουμε, ώστε να παραλληλοποιήσουμε τον κώδικα. Κάναμε τα απαραίτητα σχόλια πάνω στον σειριακό και στη συνέχεια προχωρήσαμε με την υλοποίηση αυτών.

Βήματα:

1. Προσθήκη των απαραίτητων MPI\_Init() και MPI\_Finalize(), καθώς και καθορισμός των μεταβλητών rank, size για κάθε MPI διεργασία.
2. Αντικατάσταση του drand48() με το erand48()
3. Καθορισμός των iteration που θα πραγματοποιήσει κάθε MPI διεργασία.
4. Δημιουργία struct, ώστε να μπορέσουμε στη συνέχεια να μοιράσουμε πιο εύκολα τα τοπικά ελάχιστα που θα υπολογιστούν σε κάθε MPI διεργασία. Αυτό θα πρέπει να περιέχει τις εξής μεταβλητές: fx, trial, jj, endpt
5. Κάθε MPI διεργασία θα πρέπει να υπολογίζει και να ενημερώνει κατάλληλα το “private” (σε εισαγωγικά γιατί όλα είναι private, μέχρι να υπάρξει επικοινωνία) σε αυτή struct με τα τοπικά ελάχιστα.
6. Gather όλα τα ελάχιστα στον κόμβο με rank == 0.
7. Το rank == 0 μόνο θα πρέπει να δεσμεύσει μνήμη για ένα array με structs, ώστε να αποθηκεύσει τις πληροφορίες που θα λάβει από όλες τις υπόλοιπες MPI διεργασίες.
8. Προστασία του funevals.

Το τελευταίο βήμα ήταν πιο απλό στην υλοποίησή του, από ότι ήταν στην περίπτωση του OpenMP. Κάθε στοιχείο MPI έχει τον δικό του counter, οπότε το μόνο που χρειάζεται να κάνουμε εμείς είναι, μετά το τέλος των υπολογισμών να το στείλουμε στο root rank, το οποίο με τη σειρά του θα το ελαχιστοποιήσει. Στο σημείο αυτό μπορούσαμε είτε να χρησιμοποιήσουμε την έτοιμη εντολή του MPI (MPI\_Reduce) είτε, μιας και ήδη κάναμε μια αποστολή πληροφορίας μέσω struct, να συμπεριλάβουμε και αυτή την πληροφορία. Επιλέξαμε τον δεύτερο τρόπο και επομένως ενημερώσαμε το struct με ένα ακόμα πεδίο (local\_funevals), το οποίο στάλθηκε στο rank == 0 για να τη διαχειριστεί εκείνος.

Το root rank λοιπόν μόνο, φροντίζει στη συνέχεια να επιλέξει το ελάχιστο των ελαχίστων που συγκέντρωσε, αποθηκεύοντάς το ως προσωπικό ελάχιστο (στο δικό του struct δηλαδή), καθώς επίσης να ενημερώσει τον counter του local\_funevals με τον συνολικό αριθμό από επαναλήψεις (όλων των διεργασιών) και τέλος να τυπώσει το συνολικό αποτέλεσμα. Όμοια και σε αυτόν τον κώδικα εξάγουμε τα αποτελέσματα στο αρχείο Results/res\_mpi.txt.

## MPI and OpenMP hybrid:

Χρησιμοποιώντας τον προηγούμενο κώδικα με την MPI παραλληλοποίηση, καθώς και ιδέες από την υλοποίηση OpenMP (χωρίς tasks), αναπτύξαμε και αυτόν τον κώδικα.

Η βασική ιδέα της υλοποίησής μας ήταν ότι δεν χρειάζεται να πειράξουμε τίποτα από τον προηγούμενο κώδικα και απλά να φροντίσουμε κάθε MPI διεργασία να χρησιμοποιεί threads, ώστε να εκτελέσει παράλληλα τις επαναλήψεις που της έχουν ήδη ανατεθεί. Χρησιμοποιήσαμε το struct που ορίσαμε προηγουμένως, ώστε να έχουμε πιο οργανωμένα τα δεδομένα κάθε thread και στο τέλος της παραλληλοποίησης με OpenMP, κάναμε reduction πάνω στο MPI struct με όνομα best. Εφαρμόσαμε επίσης και την ιδέα με το (\*local\_func\_evals), που χρησιμοποιήσαμε στους κώδικες (OpenMP και OpenMP-tasks), μέσω της μεταβλητής thread\_min.local\_funevals .

# Μέρος Γ: Αποτελέσματα

Παρακάτω παραθέτουμε τα γραφήματα που εξάγαμε από τις μετρήσεις, καθώς εκτελούσαμε τους παράλληλους κώδικες για διάφορους αριθμούς στοιχείων:

## OpenMP:

## OpenMP tasks:

## MPI:

## MPI and OpenMP hybrid:

# Μέρος Δ: Συμπεράσματα

Από τα αποτελέσματα παρατηρούμε καταρχάς ότι όλοι οι κώδικες τρέχουν σχεδόν στους ίδιους χρόνους. Χειρότερο να είναι το απλό MPI με καλύτερο χρόνο 13.708 δευτερόλεπτα και αμέσως καλύτερο να είναι το υβριδικό μοντέλο. Οι παράλληλοι κώδικες με OpenMP και OpenMP tasks τρέχουν με περίπου τους ίδιους καλύτερους χρόνους, γεγονός που μας οδηγεί στο συμπέρασμα ότι το OpenMP με tasks δεν επιβαρύνεται ιδιαίτερα από την πιο συχνή είσοδο των thread σε critical και atomic περιοχές και πως η προστασία που εφαρμόσαμε, αν και δεν είναι ιδανική πέτυχε τον σκοπό της. Το γεγονός ότι το MPI είναι γενικά πιο αργό από το OpenMP στην περίπτωσή μας, που τρέχουμε τον κώδικα στο ίδιο υπολογιστικό σύστημα, οφείλεται στο ότι κάθε communicator δεσμεύει και από έναν πυρήνα, επομένως ίσως δεν γίνεται τόσο καλή αξιοποίηση των thread των πυρήνων. Φυσικά δεν αποκλείεται να έχουμε και καθυστέρηση λόγω των επικοινωνιών μέσω του MPI\_Gather που χρησιμοποιήσαμε. Οφείλουμε βέβαια να ενημερώσουμε ότι μόνο στην περίπτωση του hybrid μοντέλου τρέξαμε τους κώδικες διαδοχικά με sleep 20 δευτερολέπτων, ενώ στην περίπτωση των υπόλοιπων τρέξαμε με sleep 60 δευτερολέπτων, επομένως μπορεί και αυτό να είχε επίπτωση στην απόδοση του παράλληλου κώδικα.

//Κάνε σχόλια για το speedup εδώ