PROJECT ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ
- 2022 -

Η Εργασία εκπονήθηκε από:

- *Σούρλας Ζήσης, 1072477*
- *Παπαδοπούλου Μαρία, 1072494*
- *Σφήκας Θοδωρής, 1072550*

◊ ΕΙΣΑΓΩΓΗ:

System Specifications:

Το πρόγραμμα για την παράδοση των τελικών αποτελεσμάτων εκτελέστηκε σε επεξεργαστή Ryzen 5600x και σε σύστημα με 16GB RAM 3200Mhz. Ακολουθούν περαιτέρω διευκρινίσεις πάνω στα χαρακτηριστικά του επεξεργαστή:

» AMD Ryzen 5 Desktop Processor

# of CPU Cores	→ 6
# of Threads	→ 12
Max. Boost Clock	→ Up to 4.6GHz (ενεργό)
Base Clock	→ 3.7GHz
L1 Cache	→ 64 KB (per core)
L2 Cache	→ 512 KB (per core)
L3 Cache	→ 32 MB (shared)

ΟΔΗΓΙΕΣ ΕΚΤΕΛΕΣΗΣ:

Πραγματοποιήθηκε μία προσπάθεια για παραμετροποίηση του κώδικα καθώς και για την δημιουργία μίας 'ολοκληρωμένης' παρουσίασης του. Προτείνεται η εκτέλεση σε λειτουργικό Linux με την ύπαρξη gcc. Στο φάκελο εκτέλεσης παρουσιάζονται αρχικά δύο bash scripts, οι 4 κώδικες προς εκτέλεση, ένα script σε python και το αναγκαίο makefile.

Αρχικά προτείνεται η δημιουργία ενός python virtual environment → python -m venv env (πιθανή προϋπόθεση για την δημιουργία virtual environments αποτελεί αν δεν υπάρχει ήδη το αντίστοιχο package. Η χρήση ονόματος env προτείνεται). Στην συνέχεια θα προτείναμε την εγκατάσταση των απαραίτητων βιβλιοθηκών για την εκτέλεση του python Script (κατά βάση την matplotlib και numpy) → pip install -r requirements.txt. Στην περίπτωση που τα προαναφερθέντα πακέτα είναι εγκατεστημένα σε global python interpteter, με ελάχιστες προφανείς αλλαγές στα scripts μπορεί να αποφευχθεί η παραπάνω διαδικασία.

Η εκτέλεση του script με όνομα "multipleExecutionScript" δέχεται τρία command line execution arguments. Το πλήθος των Trials στα οποία θα εκτελεστούν τα executables που θα παραχθούν, το μέγιστο πλήθος threads-processes με τα οποία θα εκτελεστούν αυξητικά τα executables (για όρισμα 6 θα γίνει εκτέλεση των προγραμμάτων με 1, 2, 4, 6 threads) και τέλος το κατά πόσο agressive optimization arguments θα δωθούν ως flags στον cc compiler (περισσότερες πληροφορίες είναι commented στο makefile). Στην περίπτωση που δεν δωθούν τα απαραίτητα command line arguments, το bash script θα εκτελεστεί με βάση

τις Default τιμές της εκφώνησης μέχρι 14 threads. Ο στόχος της αναφερθούσας διαδικασίας είναι πως αφού γίνει η εκτέλεση των προγραμμάτων, παράγονται txt files με τα αποτελέσματα κάθε εκτέλεσης τα οποία χρησιμοποιεί το python script για την δημιουργία των απαραίτητων γραφημάτων. Με το συγκεκριμένο script παράγονται και τα γραφήματα στο τμήμα αποτελεσμάτων της αναφοράς.

ΠΑΡΑΔΟΧΕΣ:

1) Αν και δεν έχουμε κατανεμημένο σύστημα για ορθό έλεγχο των αποτελεσμάτων του MPI σε αντίστοιχο περιβάλλον προσπαθήσαμε να κάνουμε optimize τον κώδικα με στόχο την ορθή λειτουργία του σε αυτά τα συστήματα.

◊ OpenMP:

Η υλοποίηση του OpenMP βασίστηκε στη λογική της συγκέντρωσης δεδομένων σε τοπικές μεταβλητές εντός του κάθε thread με την σύγκριση και επιλογή των καλύτερων εξ αυτών. Έπειτα η σύγκριση των τοπικών καλύτερων μεταβλητών με τις shared αντίστοιχες τους γίνεται μέσα σε critical region για την αποφυγή των συνθηκών ανταγωνισμού.

Συγκεκριμένα, οι μεταβλητές best_fx, best_trial, best_jj και best_pt παρέμειναν global ενώ οι μεταβλητές fx, trial, jj, startpt και endpt ορίστηκαν εντός της παράλληλης περιοχής που δημιουργήθηκε και κατά συνέπεια έγιναν τοπικές σε κάθε thread. Δημιουργήθηκαν επίσης εντός της παράλληλης περιοχής οι μεταβλητές l_best_fx, l_best_trial, l_best_jj και l_best_pt οι οποίες διατηρούν τις τιμές των καλύτερων αποτελεσμάτων εντός του κάθε thread. Επιπλέον δημιουργήθηκε η thread private μεταβλητή l_funevals, ενέργεια η οποία απαίτησε την τροποποίηση ορισμένων συναρτήσεων έτσι ώστε να τη δέχονται ως όρισμα με call by reference για να μπορούν να τροποποιούν την τιμή της όπως προβλέπει η ορθή λειτουργία του προγράμματος.

Με στόχο την παραλληλοποίηση της for loop επιλέξαμε την χρήση του default static scheduling με το μέγιστο chunksize. Κρίνουμε ότι το workload μεταξύ επαναλήψεων δεν προκύπτει ως συνάρτηση του αριθμού των trials αλλά είναι τυχαίο, βάσει των επιστρεφόμενων τιμών της erand. Δεν βρίσκουμε λόγο λοιπόν να μειώσουμε το memory handling advantage που προσφέρει το static συγκριτικά με τα dynamic και το guided scheduling.

Οι συναρτήσεις srand48 και drand48 αντικαταστάθηκαν από την thread safe erand48 η οποία λαμβάνει ως όρισμα τον randBuffer η τιμή του οποίου διαφέρει από thread σε thread ανάλογα με τον αριθμό του.

Παραλληλοποιήθηκε ο κύριος βρόγχος της main με static scheduling λαμβάνοντας και το όρισμα nowait καθώς δεν υπήρχε λόγος ύπαρξης barrier στο τέλος του παράλληλου βρόγχου. Εντός του βρόγχου το κάθε thread εκτελεί

τις κατάλληλες συναρτήσεις και ελέγχει εάν τα αποτελέσματα που παρήγαγε (fx) είναι καλύτερα από αυτά που έχει αποθηκευμένα ως καλύτερα (l_best_fx). Σε αυτή την περίπτωση ανανεώνει τις τιμές των l_best μεταβλητών.

Με την ολοκλήρωση της εκτέλεσης του βρόγχου ένα thread εισέρχεται στην critical region και ελέγχει εάν η τιμή που έχει αποθηκευμένη στην l_best_fx είναι καλύτερη (μικρότερη) από την τιμή που είναι αποθηκευμένη στην global best_fx και σε τέτοια περίπτωση ενημερώνει τις τιμές των global μεταβλητών. Επιπλέον προσθέτει στην global μεταβλητή funevals τις τιμές της l_funevals. Οι συγκεκριμένες ενέργειες γίνονται εντός critical region καθώς υπάρχουν συνθήκες ανταγωνισμού επειδή τα threads διαβάζουν και γράφουν πάνω σε global μεταβλητές.

◊ OpenMP tasks:

Για την υλοποίηση των tasks αρχικά θεωρούμε σημαντικό να αναφέρουμε τις διαφοροποιήσεις των επιλογών που κάναμε συγκριτικά με το conventional openmp καθώς και γιατί τις επιλέξαμε. Αρχικά το scoping των Tasks διαφοροποιείται από τις private μεταβλητές μίας parallel περιοχής που εξηγήσαμε προηγουμένως. Τα Tasks by default ορίζουν εκ νέου όλες τις private μεταβλητές του νήματος ως firstprivate στο task σε ένα μοναδικό για κάθε Task scope. Αυτό σημαίνει πως για να συγκρατήσουμε τις μεταβολές στις τιμές και τα καλύτερα αποτελέσματα που θα προκύψουν από την εκτέλεση των tasks επιλέξαμε την χρήση ενός shared πίνακα από structures με πλήθος τον αριθμό των χρησιμοποιούμενων threads στον οποίο συγκρατούνται οι καλύτερες τιμές από κάθε thread από τα tasks που εκτέλεσε. Θα μπορούσαμε να ανανεώνουμε ατομικά τις shared best μεταβλητές αλλά πρώτον αυτό θα εισήγαγε ένα overhead για την αποφυγή των race conditions με mutex και δεύτερον θα παρουσιαζόταν κρίνουμε false sharing. Για παρόμοιους λόγους συγκρατούμε στην ίδια δομή αποθήκευσης τα funevals του κάθε thread. Το overhead από την ατομική προσαύξησης της μεταβλητής είναι παρόμοιας τάξης μεγέθους με τον χρόνο ενός iteration με αποτέλεσμα καταστεί μη αποδοτική την παραλληλοποίηση .

Συμπληρωματικά θα θέλαμε να αναφερθούμε στον τρόπο που χρησιμοποιήσαμε την erand για την δημιουργία μίας thread safe ψευδοτυχαιότητας. Έπρεπε να βεβαιωθούμε πως σε κάθε επανάληψη - task του προγράμματος ο πίνακας startpt αρχικοποιείται με διαφορετικά στοιχεία. Η erand με την χρήση του δοθέντος κάθε φορά seed παράγει μία ψευδοτυχαία ακολουθία και κάθε φορά που καλείται επιστρέφει την επόμενη τιμή αυτής. Ωστόσο όταν το scoping της erand τερματίσει και οριστεί εκ νέου (πιθανώς σε καινούργιο task) οι τιμές της ακολουθίας επιστρέφονται από την αρχή. Το φαινόμενο αυτό θα οδηγούσε στα ίδια startpt σε κάθε task. Αρχικά λοιπόν επιλέξαμε να παραμετροποιήσουμε τον

randbuffer ώστε να δέχεται στην τρίτη θέση του και το trial. Στην λύση αυτή, αν και φαίνεται πως δεν παρουσιάζεται το πρόβλημα που συζητήθηκε παραπάνω, παρατηρούμε πως ο αριθμός των funevals είναι σημαντικά μεγαλύτερος συγκριτικά με αυτόν του sequential κώδικα. Εντέλει αρχικοποιώντας τον πίνακα startpt έξω από κάθε task (αλλά μέσα σε κάθε επανάληψη Trial) και περνώντας τον ως firstprivate όρισμα στα tasks φαίνεται πως ούτε έχουμε διπλότυπα των startpt ούτε παρουσιάζεται πρόβλημα στο task scoping και στα funevals.

Μία άλλη σημαντική επιλογή η οποία παρουσιάζει ενδιαφέρον είναι και ο τρόπος δημιουργίας των tasks. Δημιουργούμε μία παράλληλη περιοχή (static thread spawning) και το συνηθισμένο μοντέλο παραλληλοποίησης αποτελείται από την δημιουργία των tasks από ένα thread με την χρήση ενός #pragma omp single. Με κάθε δημιουργία ενός task ανατίθεται σε ένα από τα idle threads. Μετά από πειραματισμούς παρατηρήσαμε πως όταν χωρίζουμε ακόμα και την δημιουργία των tasks στα threads με την χρήση της #pragma omp for schedule(static) nowait, γεμίζουμε πολύ γρηγορότερα το task queue και έχουμε σε όλη την εκτέλεση του προγράμματος μεγαλύτερο thread utilization. Ίσως επειδή το workload του κάθε task να είναι μικρό όταν ένα μόνο thread εκτελεί το task creation δεν φτάνουμε ποτέ στο μέγιστο thread utilization. Η παρατήρηση αυτή φυσικά αφορά αποκλειστικά την περίπτωση του προβλήματος μας καθώς και το υπολογιστικό σύστημα εκτέλεσης που χρησιμοποιούμε. Ένας διαφορετικός scheduler πιθανώς να μας έδινε ανάποδα συμπεράσματα. Σε κάθε περίπτωση η διαφοροποίηση μεταξύ των δύο εκτελέσεων ήταν χρονικά πολύ μικρή.

» Παρατηρούμε ότι όσο περισσότερα threads χρησιμοποιούμε τόσο μεγαλύτερο φαίνεται το thread utilization από το γράφημα. Ωστόσο τα αποτελέσματα εξαρτώνται από το problem dimension και το γεγονός πως η κάθε εκτέλεση παρουσιάζει διαφοροποίηση στο χρόνο.

Τέλος θα θέλαμε να αναφερθούμε στο πώς τα threads δεν εκτελούν το καθένα έλεγχο για το αν το προσωπικό τους ελάχιστο είναι το ολικό μέσα σε μια κρίσιμη περιοχή. Προσπαθώντας να αποφύγουμε την χρήση και το overhead των mutexes, θεωρούμε καλύτερο τον έλεγχο εύρεσης του ολικού ελαχίστου fx έξω από την παράλληλη περιοχή. Ύστερα με την χρήση της θέση αυτού στο array of structs γίνεται η εκτύπωση των ορθών αποτελεσμάτων. Ο ολικός αριθμός ελέγχων είναι ίδιος και ο χρόνος φαίνεται να είναι μικρότερος από τον χρόνο δημιουργίας κρίσιμης περιοχής.

◊ MPI:

Αρχικοποιούμε το περιβάλλον MPI και τις μεταβλητές κάθε διεργασίας. Επιστούμε την προσοχή στη διαμόρφωση του νέου seed για την erand48, ο οποίος αποτελείται από το άθροισμα του wall clock time και του rank της κάθε διεργασίας, ώστε να μην υπάρχουν 2 διεργασίες με το ίδιο seed. Αν το πλήθος των trials έχει δοθεί από το χρήστη κατά την κλήση του προγράμματος, τότε η αντίστοιχη μεταβλητή (ntrials) ανανεώνεται. Αυτή η διαδικασία αποτελεί μια μορφή παραμετροποίησης της εκτέλεσης.

Οι διεργασίες συγχρονίζονται και ξεκινά η μέτρηση χρόνου. Κάθε διεργασία ξεκινά με ένα σημείο έναρξης, ακολουθεί τη διαδικασία εύρεσης ελαχίστου το οποίο αποθηκεύει ως δικό της ολικό ελάχιστο. Η επικοινωνία μέσω μηνυμάτων για μεταφορά δεδομένων περιορίζεται στις μεταβλητές best_fx (τιμή ολικού ελαχίστου) και funevals (πλήθος κλήσεων της συνάρτησης Rosenbrock), τα οποία καταλαμβάνουν υποχρεωτικά μία cache line, για να αποφύγουμε τη μεταφορά περιττών δεδομένων στην κρυφή μνήμη. Με ανάλογο τρόπο αρχικοποιείται και ο πίνακας global_data, για την εύρεση του καλύτερου εκ των ολικών ελαχίστων που βρήκαν οι διεργασίες.

Στο σημείο αυτό κάνουμε μία παρένθεση για να εξηγήσουμε την custom operation που έχουμε ορίσει και χρησιμοποιήσει. Η cust_min (γραμμές 273-277) έχει διπλή λειτουργία: προσαυξάνει την τιμή του inoutvec[0] κατά την τιμή του invec[0] και ελέγχει αν η τιμή του invec[1] είναι μικρότερη αυτής του inoutvec[1]. Στην περίπτωση που αυτό ισχύει ανανεώνει την τελευταία με βάση την πρώτη. Η πρώτη λειτουργία αφορά την άθροιση των τιμών funevals και η δεύτερη την εξαγωγή του καλύτερου ολικού ελαχίστου μεταξύ των διεργασιών.

Δηλώνουμε την κατασκευή της custom operation στη main (γραμμές 355-356) και την ονομάζουμε MPI_MY_MIN (ίσως μπορούσε να γίνει cumulative). Οι πίνακες local data (που περιέχουν τα προαναφερθέντα δεδομένα) κάθε διεργασίας που γίνονται reduced στον πίνακα global_data και το αποτέλεσμα γίνεται broadcast σε όλες τις διεργασίες. Ο συγχρονισμός μεταξύ των διεργασιών επιτυγχάνεται μέσα από την MPI_Allreduce, ως blocking communication function, δηλαδή δεν επιστρέφει μέχρι όλα τα απαραίτητα μηνύματα να έχουν

ανταλλαχθεί. Στο τέλος, κάθε διεργασία είναι σε θέση να καταλάβει αν είναι εκείνη που έχει βρει το καλύτερο ολικό ελάχιστο με ένα απλό έλεγχο μεταξύ της μεταβλητής της, best_fx και της global_data[1] (που την έχουν όλοι λόγω της Allreduce). Επειδή αυτή είναι η διεργασία που κατέχει όλα τα δεδομένα για το καλύτερο ολικό ελάχιστο, είναι η μόνη που κάνει τις εκτυπώσεις των αποτελεσμάτων στο τέλος του προγράμματος, πριν κλείσουμε το περιβάλλον MPI. Τα σχόλια τέλος για το free(output) αν και θα έπρεπε να ανήκουν στον κορμό του προγράμματος, κατά τις εκτελέσεις φαίνεται το πρόγραμμα να κάνει free τον FILE pointer πριν προλάβει να γράψει σε αυτό τα αποτελέσματα. Ίσως η λειτουργία της fprintf να γίνεται ασύγχρονα με αποτέλεσμα την πιθανή λάθος εκτέλεση του προγράμματος. Έτσι και αλλιώς κατά το τέλος της εκτέλεσης του προγράμματος η μνήμη αποδεσμεύεται, δεν θεωρούμε πως έχουμε κάποιο memory leak.

◊ Hybrid MPI+OpenMP:

Πρόκειται για συνδυασμό των τεχνικών που αναλύσαμε στις ομώνυμες ενότητες. Μόλις οι διεργασίες συγχρονιστούν, κάθε μία γεννά νήματα και αρχικοποιεί τις μεταβλητές τους. Κρίνεται χρήσιμο να σημειωθεί ότι οι μεταβλητές των διεργασιών που έχουν δηλωθεί πριν την paraller region είναι κοινά διαμοιραζόμενες σε όλα τα threads της διεργασίας που τα δημιουργεί.

Στο άθροισμα για τον seed της erand48 συμμετέχει πλέον και το thread_id με σκοπό κάθε thread να έχει διαφορετικό seed για την γεννήτρια ψευδοτυχαίων αριθμών. Κάθε thread αναλαμβάνει ένα μέρος της for λούπας μεγέθους (#επαναλήψεων)/(#threads) τα οποία (μέρη) τους ανατίθενται κυκλικά. Ακολουθεί η διαδικασία εύρεσης ολικού ελαχίστου από κάθε thread. Μόλις κάθε thread τελειώσει ελέγχει με αμοιβαίο αποκλεισμό αν το ελάχιστο που βρήκε είναι καλύτερο από το συνολικό της διεργασίας και ανανεώνουν τα δεδομένα για αυτό, αν ο έλεγχος είναι αληθής.

Η εύρεση ολικού ελαχίστου σε επίπεδο διεργασίας έχει τελειώσει. Ακολουθεί το reduction μεταξύ των ελαχίστων των διεργασιών με τον τρόπο που περιγράψαμε στην προηγούμενη ενότητα για την εξαγωγή του καλύτερου εκ των ελαχίστων που βρήκαν οι διεργασίες, δηλαδή του ολικού ελαχίστου.

Τέλος, κατά την εκτέλεση του Hybrid μοντέλου στο κομμάτι του OpenMP, επιλέγουμε την χρήση 2 threads ανά process λόγω περιορισμένων υπολογιστικών πόρων. Με στόχο την δίκαιη μέτρηση των αποτελεσμάτων συγκριτικά με τα υπόλοιπα εκτελέσιμα, ο αριθμός των processes*threads που χρησιμοποιούνται στο hybrid μοντέλο ταυτίζεται με τον αριθμό των threads στο OpenMp και των processes στο μοντέλο του MPI αντίστοιχα.

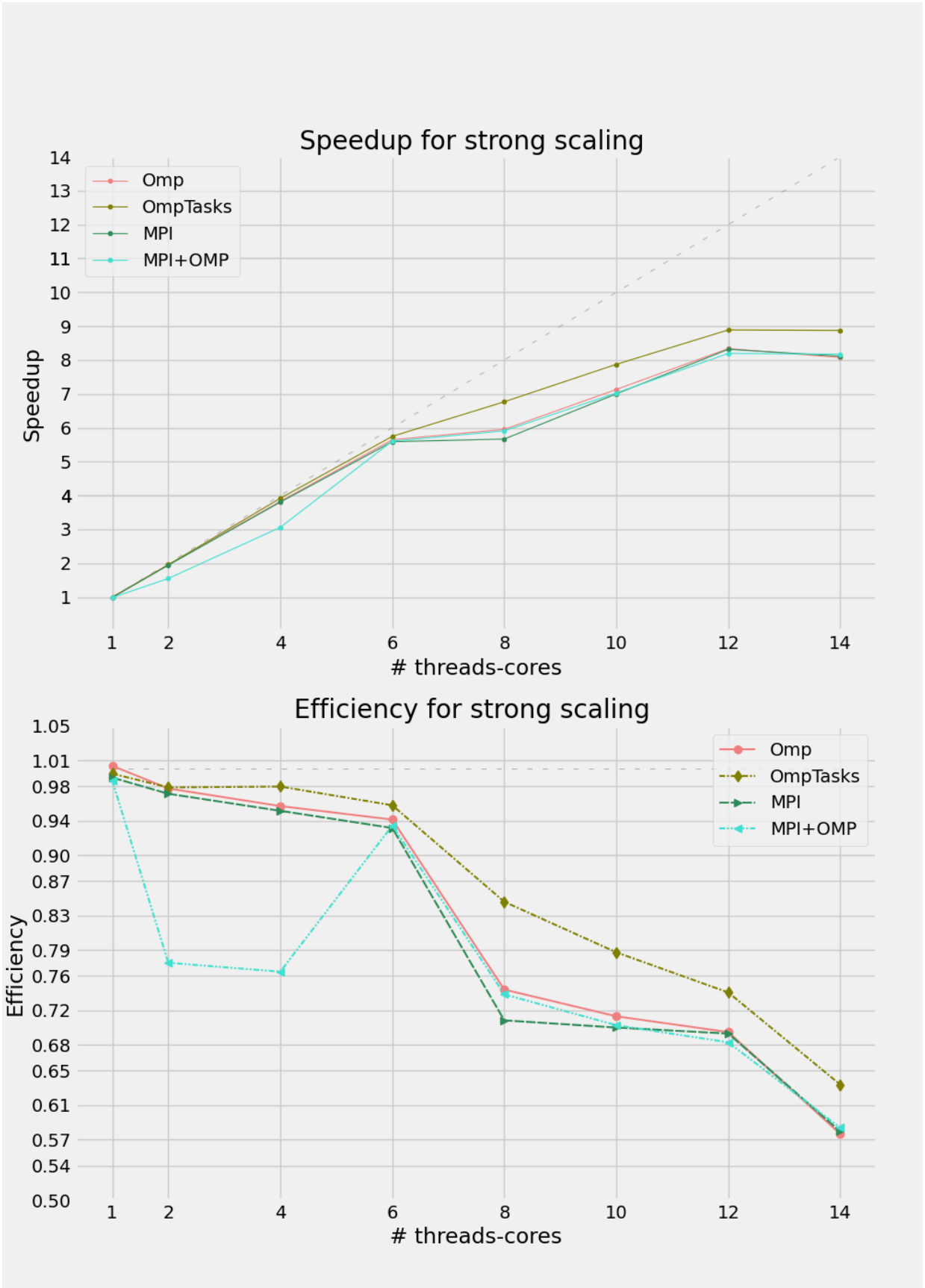
◊ ΑΠΟΤΕΛΕΣΜΑΤΑ:

» Για την μέτρηση των αποτελεσμάτων θεωρούμε σημαντικό να σημειώσουμε πως στον χρόνο εκτέλεσης υπολογίζεται και η παραλληλοποίηση του κώδικα και ο απαιτούμενος χρόνος συλλογής των απαραίτητων αποτελεσμάτων από τα νήματα. Τέλος αξίζει να σημειωθεί πως κάθε εκτέλεση του κώδικα παρουσιάζει μικρές διαφοροποιήσεις στον χρόνο εκτέλεσης, φαινόμενο που αιτιολογούμε τόσο στην φύση του αλγόριθμου όσο και στο scheduling του υπολογιστικού μας συστήματος

» Ακολουθούν αρχικά οι πίνακες των ζητούμενων αποτελεσμάτων για τους 5 κώδικες σε 64K trials και το συλλογικό γράφημα του Speedup και του Efficiency που παρατηρείται.

Πρόγραμμα εκτέλεσης	**Πυρήνες εκτέλεσης**	**Elapsed time (s)**	**Best_fx**	**Funevals**	**Speedup(p)**
Σειριακός	1	512.317	9.332967e-09	14788915635	-
Omp	1	510.456	9.6065020e-09	14803560169	1.0036
	2	262.077	8.4012495e-09	14811554607	1.9548
	4	133.840	1.1539624e-08	14877015457	3.8278
	6	71.839	8.8276352e-09	14812512440	5.6478
	8	86.058	9.0507081e-09	14838756093	5.5953
	10	71.839	8.8276352e-09	14835370810	7.1314
	12	61.443	8.6662800e-09	14833865687	8.3380
	14	63.467	1.3322927e-08	14830301985	8.0721
Omp_Tasks	1	528.022	1.1320976e-08	15122939863	0.9945
	2	261.767	6.7778500e-09	14860334974	1.9571

	4	130.742	1.1317912e-08	14739849990	3.9185
	6	89.141	1.5165060e-08	14841742761	5.7472
	8	75.701	2.8950366e-08	14874621595	6.7676
	10	65.068	9.7264075e-09	14815392355	7.8735
	12	57.633	1.4268178e-08	14763531365	8.8892
	14	57.731	2.2364821e-08	14802788608	8.8742
MPI	1	517.495	1.0495716e-08	14901887970	0.9899
	2	263.746	1.0538302e-08	14802438374	1.9424
	4	134.613	7.5548656e-09	14843686261	3.8058
	6	91.662	1.1315744e-08	14839211654	5.5891
	8	90.395	8.4451818e-09	14805482425	5.6675
	10	73.179	7.3767000e-09	14870352070	7.0008
	12	61.596	7.3114816e-09	14765360681	8.3173
	14	63.144	1.0655541e-08	14819959162	8.1134
MPI+OMP	1	519.288	6.4390237e-09	14858621906	0.9865
	2	330.434	7.6714993e-09	14864722751	1.5504
	4	167.441	9.2142665e-09	14733396035	3.0596
	6	91.385	9.2807101e-09	14771421621	5.6061
	8	86.703	1.3139553e-08	14779262119	5.9088
	10	72.896	9.4980880e-09	14862339050	7.0280
	12	62.514	1.0025896e-08	14885843776	8.1952
	14	62.681	9.4092223e-09	14801714001	8.1734


Speedup for strong scaling
Omp
OmpTasks
MPI
MPI+OMP
Speedup
threads-cores
Efficiency for strong scaling
Omp
OmpTasks
MPI
MPI+OMP
Efficiency
threads-cores

» Σχολιασμός Αποτελεσμάτων:

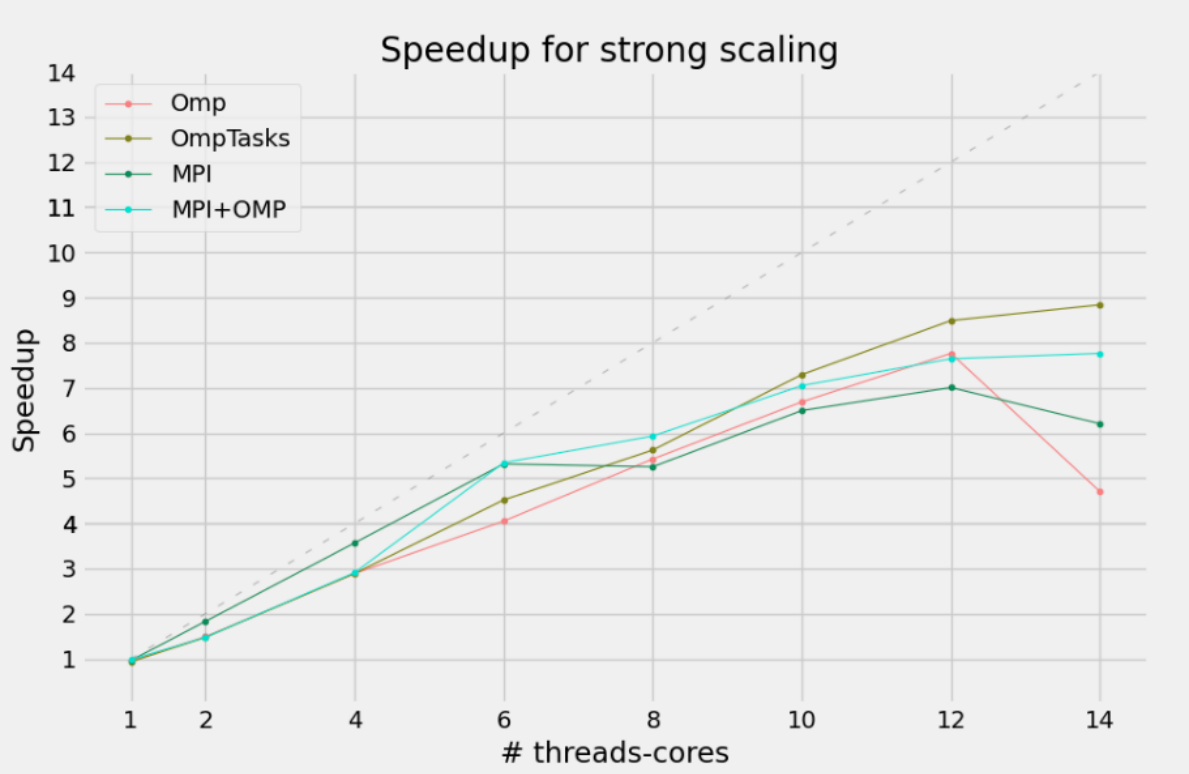
Αρχικά, φαίνεται πως μέχρι την χρήση των 6 threads στα executables (όσα και το πλήθος των physical cores) το speedup των mpi, openmp και OpenMPTasks πλησιάζουν πολύ το ιδανικό. Με την χρήση των λογικών πυρήνων η απόδοση των υλοποιήσεων μειώνεται εμφανώς, όπως αναμένεται. Όταν ξεπεραστεί και το πλήθος των λογικών cores, η απόδοση μειώνεται με αυξανόμενο ρυθμό.

Γενικά παρατηρούμε πως η υλοποίηση με Tasks είναι πιο γρήγορη από τις υπόλοιπες. Η αυξημένη ταχύτητα της ερμηνεύεται από το γεγονός ότι τα tasks είναι πιο lightweight διαδικασίες. Φαίνεται πως λόγω της φύσης του κώδικα διατηρούν πολύ καλύτερο Speedup αφού ξεπεράσουμε τον αριθμό των λογικών πυρήνων του υπολογιστικού μας συστήματος.

Ακολουθούν οι υλοποιήσεις με OpenMP και MPI, οι οποίες έχουν στο περίπου την ίδια απόδοση.

Αργότερη φαίνεται να είναι η υλοποίηση του υβριδικού μοντέλου MPI+OpenMP για την χρήση των λίγων πυρήνων. Υποθέτουμε ότι η πτώση που βλέπουμε για 1, 2 και 4 threads, σχετίζεται με την environmental variable OMP_PROC_BIND, η οποία είναι true, με σκοπό την δέσμευση των threads στο node από το οποίο κλήθηκε η διεργασία. Η υπόθεση βασίζεται στον παρακάτω συλλογισμό-πείραμα:

Τα αποτελέσματα που παραθέσαμε παραπάνω (πίνακας και διάγραμμα) δεν χρησιμοποιούν το OMP_PROC_BIND = true ως environmental variable όπως προτάθηκε. Παρατηρούμε πως αν και περιορίζοντας τις διεργασίες σε συγκεκριμένους πυρήνες επιτυγχάνουμε πολύ καλύτερο memory management και λιγότερες μεταφορές πληροφορίας, λόγω του scheduler του λειτουργικού μας συστήματος περιορίζεται σημαντικά η απόδοση. Σε έναν δρομολογητή τύπου fcfs θα βλέπαμε αναπόφευκτα διαφορετικά τελείως αποτελέσματα. Ωστόσο ο περιορισμός αυτός φαίνεται να αφήνει unutilized cores σε ένα συμβατικό σύστημα αφού παράλληλα με εκτελέσιμο υπάρχουν πολλές ακόμα διεργασίες που δεσμεύουν τους πυρήνες για διαφορετικές χρονικές περιόδους. Φαίνεται γρηγορότερη η μετακίνηση της απαραίτητης πληροφορίας σε ελεύθερους πυρήνες και η δέσμευση αυτών από το εκτελέσιμο μας παρά η παραμονή του σε συγκεκριμένο core queue. (Τα προγράμματα εκτελέστηκαν μετά από επανεκκίνηση με στόχο την ύπαρξη των λιγότερων δυνατών kernel processes). Παρακάτω φαίνονται και τα παραγόμενα αποτελέσματα για OMP_PROC_BIND = true.



» Η εκτέλεση για το διάγραμμα έγινε για 4096 trials και δεν είναι φυσικά επαρκής για την ανάδειξη ουσιωδών αποτελεσμάτων. Ωστόσο εμφανίζει ικανώς το μοτίβο πτώσης της επίδοσης σε λίγους πυρήνες όπου το OMP_PROC_BIND=true. Ενώ έγιναν και δοκιμές σε 64000 trials και το binding δεν πρόσφερε μεγαλύτερο Speedup ούτε με την χρήση 12 threads.

Πέρα από την εμφανή χειρότερη επίδοση για λίγα νήματα που παρατηρείται, θα θέλαμε να σημειώσουμε και το γεγονός πως για 2 και 4 threads τα εκτελέσιμα του Omp και OmpTasks φαίνεται να παρουσιάζουν το ίδιο performance drop με αυτό που παρατηρείται στο MPI+OMP για ίδιο πλήθος πυρήνων ανεξαρτήτως του binding. Αφού το MPI στοχεύει επί των πλείστων σε distributed συστήματα, εικάζουμε πως το hybrid μοντέλο του, όταν δημιουργεί threads, θα τα κάνει αυτόματα bind στο αντίστοιχο node από το οποίο καλείται η διεργασία. Για αυτό και για 1, 2 και 4 threads τα τρία εκτελέσιμα που βασίζονται στο OpenMP παρουσιάζουν παρόμοια απόδοση (το hybrid για 1 process λειτουργεί για n=1 και 1 thread ενώ για 2 και 4 χρησιμοποιεί n=1, n=2 αντίστοιχα και 2 threads). Χρησιμοποιείται το OpenMPI για την εκτέλεση.

◊ ΣΥΜΠΕΡΑΣΜΑ:

Ανακεφαλαιώνοντας, γίνεται αντιληπτό ότι, με βάση τις υλοποιήσεις μας, το OpenMP-Tasks είναι το πιο αποδοτικό. Ακολουθούν τα OpenMP και MPI με σχεδόν ίδια απόδοση, παρά τη χρήση threads και processes αντίστοιχα. Η λιγότερο αποδοτική υλοποίηση φαίνεται να είναι αυτή του υβριδικού OpenMP+MPI, για τους λόγους που περιγράψαμε παραπάνω.

Τα αποτελέσματα επιβεβαιώνουν τη διδαχθείσα θεωρία, σύμφωνα με την οποία το υβριδικό μοντέλο είναι καταλληλότερο για distributed systems (αφού βασικό του στοιχείο είναι το MPI), όπου τρέχουν πολλές διεργασίες σε πολλούς πυρήνες, οργανωμένους σε clusters. Αντίθετα, οι υλοποιήσεις με OpenMP και OpenMp-Tasks ενδείκνυνται για συμβατικά συστήματα, όπως οι προσωπικοί υπολογιστές.