Εργασία Παράλληλα και διανεμημένα συστήματα

Μιχαήλ Μπαλτζάκη

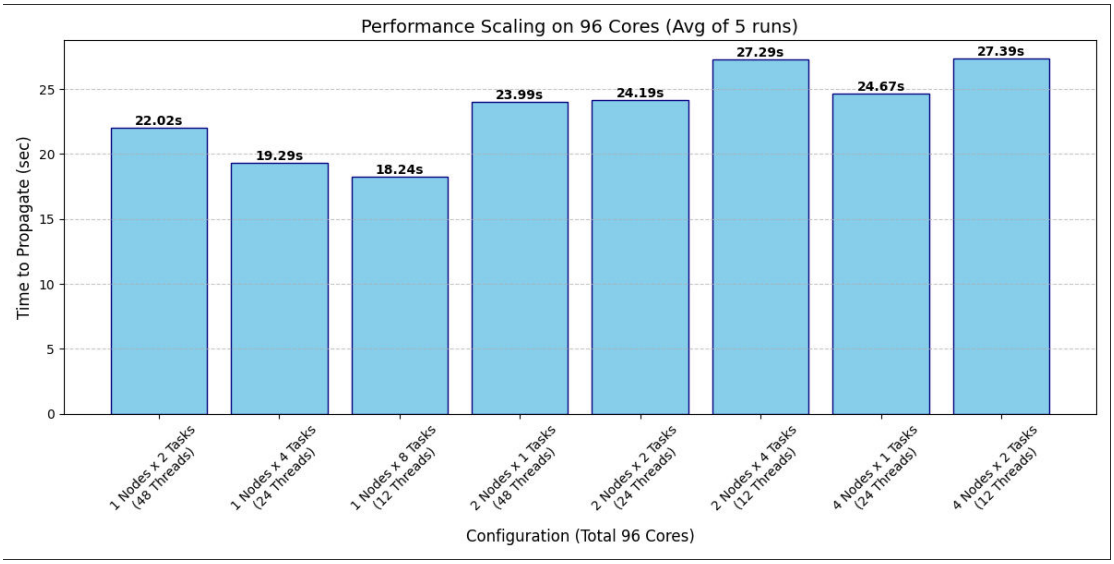
AM:10989

Link: https://github.com/MixalisBaltzakis/commit_for_second_exercise.git

Η εργασία αυτή είναι επέκταση της πρώτης εργασίας. Συζητάμε ξανά το ίδιο πρόβλημα, τον αλγόριθμο label_propagation κατά μήκος ενός unweighted binary graph. Αυτή τη φορά επεκτεινόμαστε σε μεγάλους γράφους οι οποίοι εξαντλούν τη μνήμη ενός συμβατικού υπολογιστή (στην περίπτωσή μου ειχα άνω όριο 8G στον υπολογιστή μου) και εκτελούμε τα πειράματα στο aristotle cluster όπου έχουμε πρ'οσβαση σε μεγάλη μνήμη πολλά μηχανήματα και πολλά cores. Η εργασία είναι υλοποιημένη σε συνδυασμό openmp και mpi ώστε να μπορέσουμε να αξιοποιήσουμε τη κοινή δύναμη δύο μηχανημάτων και πάνω. Ξεκινώντας από τον τρόπο που διαβάζω τον γράφο έχω ένα βοηθητικό python script που παίρνει το .mat αρχείο ,το οποίο έχουμε κατεβάσει μέσα στον scratch folder ώστε να μην έχουμε θέματα με τη μνήμη στο home directory , και μου δίνει ένα φάκελο με 3 αρχεία μέσα. Τα 2 είναι binary και περιέχουν σε binary format το row_ptr το ένα και το col_indx το άλλο. Το 3^{ο} είναι text και περιλαμβάνει τα nnz του γράφου και τον αριθμό των κόμβων του (node_number). Η διαδικασία αυτή απαιτείται μόνο τη πρώτη φορά που τρέχουμε κάποιο τεστ για ένα συγκεκριμένο γράφο, τις επόμενες τσεκάρουμε ότι ο φάκελος με τα 3 αρχεία υπάρχει και τα διαβάζουμε χωρίς να χρειαστεί να τα παράξουμε ξανά. Στο C πρόγραμμά μου για να διαβάσω τον γράφο χρησιμοποιώ τη συναρτηση load_problem_parallel που παίρνει ορίσματα το rank του process που την καλεί, τον συνολικό αριθμό των processes και το όνομα του φακέλου με τα 3 αρχεία που ανέφερε πριν. Η συνάρτηση σπάει τον γράφο σε κομμάτια και κάθε process φορτώνει μόνο δικό του κομμάτι διαβάζοντας μόνο τα αντίστοιχα bytes από τα αρχεία. Κάθε process παίρνει node_number/procnum κόμβους εκτός το τελευταίο που παίρνει και ότι έχει περισσέψει από την ακέραια διαίρεση. Στον row_ptr χρησιμοποιώ long long τύπο καθώς για γράφους με πάρα πολλά nnz ο απλός int ξεχειλίζει και κατ επέκταση όποια μεταβλητή έχει να κάνει με τα nnz έχει τύπο long long. Οι συναρτήσεις που μου επιτρέπουν το παράλληλο διάβασμα είναι οι MPI_File_read_at που με πηγαίνουν κατευθείαν στο byte που θέλω να διαβάσω(offset) και λόγω της απουσίας κοινού κέρσορα όλα τα processes μπορούν να διαβάσουν ταυτόχρονα χωρίς να πέφτει το ένα πάνω στο άλλο. Αυτά καθιστούν την ανάγνωση του γράφου εξαιρετικά γρήγορη (μερικά seconds) και παρατηρούμε επιτάχυνση της διαδικασίας με την αύξηση ,έως ένα λογικό σημείο, των processes που συμμετέχουν. Στο επόμενο βήμα υπολογίζω δύο πίνακες τους recvcounts και rdispls που θα μου χρειαστούν στο κομμάτι της επικοινωνίας. Μπαίνοντας στο κύριο κομμάτι η λογική μου έχει την δομή: επικοινωνία -> υπολογισμοί -> έλεγχος για λήξη του αλγορίθμου. Στο κομμάτι της επικοινωνίας

χρησιμοποιώ την MPI_Allgatherv η οποία μαζεύει από κάθε process ένα συγκεκριμένο κομμάτι του labels, συνθέτει το ενημερωμένο labels και το μοιράζει πίσω σε όλους στο τέλος. Το κόστος της επικοινωνίας αυτής είναι μεγάλο καθώς προϋποθέτει συγχρονισμό και ανταλλαγή μνημάτων και γι αυτό έχω επιλέξει να το κάνω κάθε 10 επαναλήψεις του αλγορίθμου μετά από δοκιμές τιμών. Στο κομμάτι του υπολογισμού έχω αφήσει ελεύθερα τα labels σε data race καθώς κάποια σύγκρουση δε θα μας δώσει ‘σκουπίδια’ απλώς θα καθυστερήσει μερικές επαναλήψεις την σύγκλιση του αλγορίθμου. Στο κομμάτι του ελέγχου τερματισμού συγκρίνω αν το labels άλλαξε **τοπικά** (δηλαδή στους κόμβους υπευθυνότητας του εκάστοτε process) στις τελευταίες 20 επαναλήψεις του αλγορίθμου(για λόγους performance και εδώ το βήμα αυτό). Αν άλλαξε κάνω το τοπικό exit 0 ενημερώνω το labels_check για το επόμενο τσεκάρισμα μετά από 20 επαναλήψεις και συνεχίζω. Η μόνη περίπτωση να βγω είναι όταν όλα μου τα processes δώσουν τοπικό exit =1 καθώς με ένα reducer αθροίζω τα exit και μόνο όταν το global_exit == procnum μπορώ να τερματίσω. Στη συνέχεια αφού έχω φύγει από την while(true) τυπώνω μερικά στατιστικά κάνω free τη μνήμη που έχω δεσμεύσει και τερματίζω.

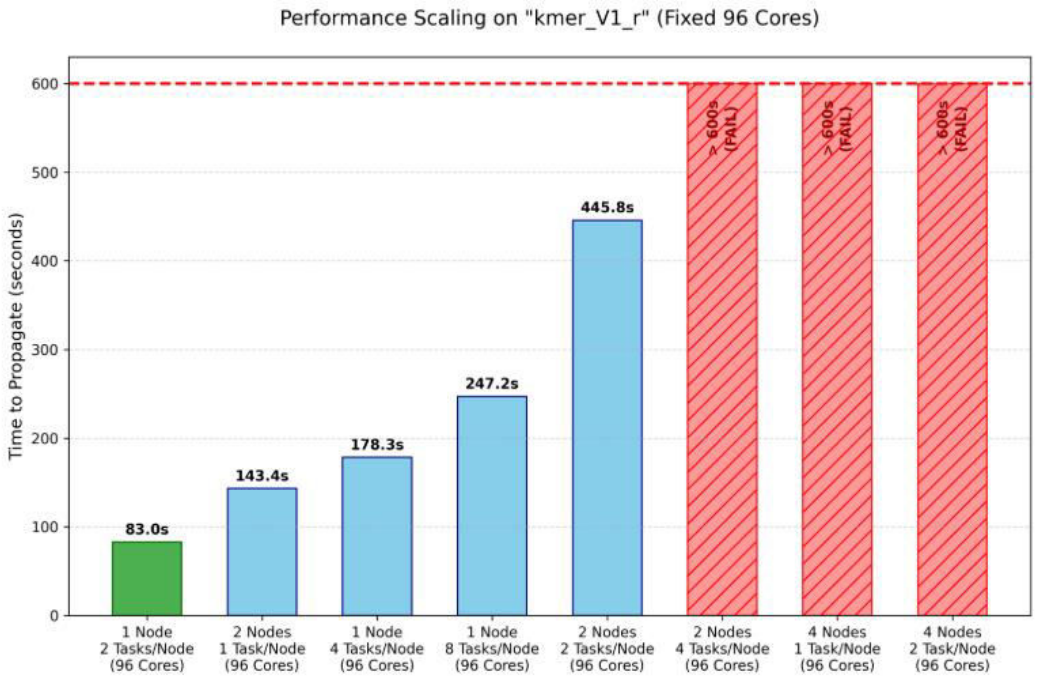
Στο κομμάτι του testing για αρχή έχω διαλέξει τον **com-Friendster** ο οποίος είναι ένας τεράστιος γράφος κοινωνικών δικτύων.



Έχω δουλέψει με σταθερά 96 cores αλλάζοντας την κατανομή τους με min ntasks=2 και max ntasks=8. Στο διάγραμμα παρατηρούμε πως η βέλτιστη περίπτωση είναι όταν τρέχουμε σε 1 node με 8 processes και 12 threads per process. Αυτό συμβαίνει καθώς με 8 processes έχουμε μικρότερες ομάδες νημάτων άρα μείωση κόστους συγχρονισμού, καλύτερη τοπικότητα δεδομένων χωράνε αποδοτικότερα στην cache και αποφυγή προσπέλασης απομακρυσμένης μνήμης. Μόλις εισάγουμε το πρόγραμμα σε παραπάνω από 1 nodes (2, 4) βλέπουμε πως ενώ πλέον τα μυνήματα ταξιδεύουν μέσα στο δίκτυο για να πάνε από μηχάνημα σε μηχάνημα τα αποτελέσματα δεν είναι τραγικά υπάρχει μια μικρή αύξηση του χρόνου αλλά

βιώσιμη. Αυτό συμβαίνει καθώς ο Friendster είναι γράφος με υψηλή συνδεσιμότητα, μικρή διάμετρο και έτσι ο αλγόριθμος μας έχει σχετικά λίγα βήματα για το μέγεθος του γράφου (γρήγορη εξάπλωση των labels από άκρη σε άκρη) περιορίζοντας τον αριθμό των επικοινωνιών. Όσο ανεβαίνουν τα processes για nodes = 2 και nodes = 4 παρατηρούμε ελαφριά αύξηση του χρόνου που είναι λογική καθώς πλέον τα μυνήματα μας ταξιδεύουν μέσα στο δίκτυο και όχι τοπικά όπως στη περίπτωση με node = 1 και ntasks = 8.

Στη συνέχεια θα παρουσιάσουμε τον **kmer_V1r** ο οποίος είναι ένας γράφος εντελώς διαφορετικός από τον com-Friendster. Είναι γραμμικός και έχει πάρα πολλή μεγάλη διάμετρο προσομοιώνοντας αλυσίδες DNA και έτσι θέλουμε πάρα πολλές επαναλήψεις του αλγορίθμου για να μεταβούμε από τη μία μεριά της αλυσίδας στην άλλη.



Εδώ παρατηρούμε την τεράστια διαφορά σε χρόνο, της καλύτερης λύσης ανάμεσα στους 2 γράφους από 2 έως 8 processes. Εδώ η εντελώς βέλτιστη λύση είναι με 1 process και 96 threads που δεν την συπεριέλαβα στο διάγραμμα καθώς είναι pure openmp κώδικας. Το κόστος επικοινωνίας 'γονατίζει' την ταχύτητα καθώς τα βήματα του αλγορίθμου μέχρι την ολοκλήρωση είναι πάρα πολλά και κατ επέκταση και οι επικοινωνίες το ίδιο. Παρατηρούμε ότι όσο αυξάνουμε τα processes είναι σαν να δημιουργούμε σύνορα στα οποία η πληροφορία κολλάει μέχρι την επόμενη ενημέρωση ώστε να περάσει και να πάρει επιτέλους τη μικρότερη τιμή και για αυτό ο χρόνος για 8 processes γίνεται πάνω από 5 φορές μεγαλύτερος. Αποκορύφωμα είναι οι κόκκινες στήλες που βλέπουμε στο διάγραμμα όπου εχουμε timeout στα 10 λεπτά άνω ορίου που είχα θέσει για την εκτέλεση του προγράμματος. Κλείνοντας για τον kmer_V1r παρατηρούμε το πόσο σημαντική είναι η τοπολογία του

προβλήματος για το αν είναι ή όχι βιώσιμο ένα πρόβλημα σε distributed memory. Ο com-Friendster είχε πολύ καλή συμπεριφορά ενώ ο kmer_V1r ήδη από nodes = 2 με ntasks = 2 ήταν πολύ προβληματικός.