Set 1 - MPI and OpenMP

Φοιτητής 1

Ονοματεπώνυμο: Αγγουρά Ρουμπίνη Μαρία

ΑΜ: 1084634

Έτος Σπουδών: 5ο

Φοιτητής 2

Ονοματεπώνυμο: Παυλόπουλος Ιάσονας

ΑΜ: 1084565

Έτος Σπουδών: 5ο

Επεξήγηση Υλοποίησης

Ερώτημα 1

Α) Κάθε διεργασία στέλνει τη δική της τιμή (send_val) σε όλες τις διεργασίες υψηλότερου rank μέσω της MPI_Send. Έπειτα, κάθε διεργασία λαμβάνει τιμές από διεργασίες χαμηλότερου rank μέσω της MPI_Recv και τις προσθέτει για να υπολογίσει ένα επιμέρους άθροισμα το οποίο αποθηκεύεται στην recv_val. Για να επιτευχθεί η λειτουργία της Exscan, η διεργασία 0 επιστρέφει πάντα recv_val = 0, ενώ μετά οι υπόλοιπες επιστρέφουν το επιμέρους άθροισμα χωρίς να προστεθεί η τιμή τους σε αυτό.

Β) Κάθε διεργασία υπολογίζει το συνολικό άθροισμα των τιμών όλων των νημάτων της (thread_values) χρησιμοποιώντας OpenMP reduction με SUM Operator. Η πρώτη διεργασία (rank 0) ξεκινά με prefix sum = 0. Για τις υπόλοιπες διεργασίες, λαμβάνεται το prefix sum από την προηγούμενη διεργασία μέσω της MPI_Recv, προστίθεται στο άθροισμα των νημάτων της τρέχουσας διεργασίας και, αν δεν είναι η τελευταία διεργασία, αποστέλλεται στην επόμενη μέσω της MPI_Send.

Στη συνέχεια, κάθε νήμα υπολογίζει την τιμή της exclusive scan για το ίδιο, λαμβάνοντας υπόψη το prefix sum της διεργασίας και το άθροισμα όλων των προηγούμενων νημάτων (μέσα στη διεργασία) μέσω βρόχου. Το αποτέλεσμα αποθηκεύεται στο exclusive_scan_results για κάθε νήμα.

Γ) Αρχικά, δημιουργούμε ένα N*N*N Matrix για κάθε νήμα με τυχαίες τιμές.

Με την βοήθεια της συνάρτησης που υλοποιήσαμε στο προηγούμενο ερώτημα, κάθε νήμα υπολογίζει το offset του στο αρχείο χρησιμοποιώντας τα αποτελέσματα του Exclusive Scan (exclusive_scan_results) και το μέγεθος του Matrix. Το offset αυτό καθορίζει τη θέση από την οποία θα γράψει τα δεδομένα του.

Με τη χρήση της MPI_File_write_at, κάθε νήμα γράφει τα δεδομένα του matrix στο matrix_data.bin στη θέση που υπολογίστηκε από το offset.

Με τη χρήση της MPI_File_read_at, κάθε νήμα διαβάζει από το matrix_data.bin το κομμάτι δεδομένων που του αναλογεί (με βάση το offset που έχει υπολογιστεί νωρίτερα) και τα δεδομένα αποθηκεύονται στον πίνακα read_matrix.

Τέλος, κάθε νήμα συγκρίνει τα δεδομένα που διαβάστηκαν (read_matrix) με τα αρχικά δεδομένα (matrix). Εάν υπάρχει διαφορά, εμφανίζεται μήνυμα σφάλματος με τη θέση της ανακολουθίας. Αν τα δεδομένα συμφωνούν, επιστρέφεται μήνυμα επιτυχούς επικύρωσης.

Δ) Όπως και στο προηγούμενο ερώτημα, δημιουργούμε ένα N*N*N Matrix για κάθε νήμα με τυχαίες τιμές. Με την χρήση των συναρτήσεων της ZLIB, συμπιέζουμε τα δεδομένα και με τη βοήθεια της συνάρτησης που υλοποιήσαμε στο Ερώτημα Β, κάθε νήμα υπολογίζει το offset του στο αρχείο χρησιμοποιώντας τα αποτελέσματα του Exclusive Scan (exclusive_scan_results) και το μέγεθος των δεδομένων που έχει συμπιεστεί. Το offset αυτό καθορίζει τη θέση από την οποία θα γράψει τα δεδομένα του.

Με τη χρήση της MPI_File_write_at, κάθε νήμα γράφει τα συμπιεσμένα δεδομένα του πίνακα (compressed_data) στο αρχείο matrix_data_compressed.bin στη θέση που έχει υπολογιστεί από το offset.

Με τη χρήση της MPI_File_read_at, κάθε νήμα διαβάζει από το matrix_data_compressed.bin το κομμάτι των συμπιεσμένων δεδομένων που του αναλογεί (με βάση το offset που έχει υπολογιστεί νωρίτερα). Τα δεδομένα που διαβάζονται αποθηκεύονται στο Matrix read_compressed_data και στη συνέχεια αποσυμπιέζονται στο Matrix read_matrix.

Τέλος, κάθε νήμα συγκρίνει τα δεδομένα που διαβάστηκαν (read_matrix) με τα αρχικά δεδομένα (matrix). Εάν υπάρχει οποιαδήποτε διαφορά, εμφανίζεται μήνυμα σφάλματος, και το πρόγραμμα τερματίζει με αποτυχία. Αν τα δεδομένα συμφωνούν, επιστρέφεται μήνυμα επιτυχούς επικύρωσης από κάθε νήμα, το οποίο υποδεικνύει ότι η διαδικασία ολοκληρώθηκε σωστά.

Παρατηρήσεις

1. Όπου κρίθηκε αναγκαίο, υπάρχουν τα απαραίτητα Comments στον κώδικα για την ευκολότερη ανάγνωση και κατανόησή του.
2. Επειδή γίνεται δυναμική ανάθεση μνήμης για τις μεταβλητές και τους πίνακες που δημιουργούμε, σε κάθε περίπτωση που δεν έγινε σωστή ανάθεση, έχουμε προσθέσει μηνύματα λάθους για ευκολότερο debugging.
3. Είναι σημαντικό, επίσης, να αναφέρουμε πως παρατηρήθηκε μια αστάθεια στην εκτέλεση του κώδικα των ερωτημάτων Γ και Δ, ειδικά όταν χρησιμοποιούταν μεγάλος αριθμός από Processes (Processes > 4). Το πρόβλημα, φαίνεται να οφείλεται στην διαχείριση της μνήμης μεταξύ των Threads που γίνονται Spawn. Επομένως, προτείνεται η εκτέλεση αυτών των Scripts με 2-4 Processes.

Ερώτημα 2

Προτού δημιουργήσουμε τα scripts που παραλληλοποιούν την διαδικασία του gridsearch, μετρήσαμε τον χρόνο εκτέλεσης του gs.py, χρησιμοποιώντας την συνάρτηση time().

Α) Η multiprocessing.Pool χρησιμοποιείται για την παράλληλη εκτέλεση της συνάρτησης evaluate_model για κάθε συνδυασμό υπερπαραμέτρων στο Grid. Αρχικά, δημιουργείται μια ομάδα διεργασιών συνήθως ίσες με τους πυρήνες του επεξεργαστή σε αριθμό. Η συνάρτηση pool.map διαμοιράζει το έργο στις διεργασίες, αναθέτοντας σε κάθε διεργασία έναν ή περισσότερους συνδυασμούς υπερπαραμέτρων από το Grid. Τέλος, κάθε διεργασία υπολογίζει την ακρίβεια για τους συνδυασμούς που της ανατέθηκαν και επιστρέφει τα αποτελέσματα, καθώς επίσης και το speedup που υπολογίζεται σε σχέση με την σειριακή εκτέλεση του grid search.

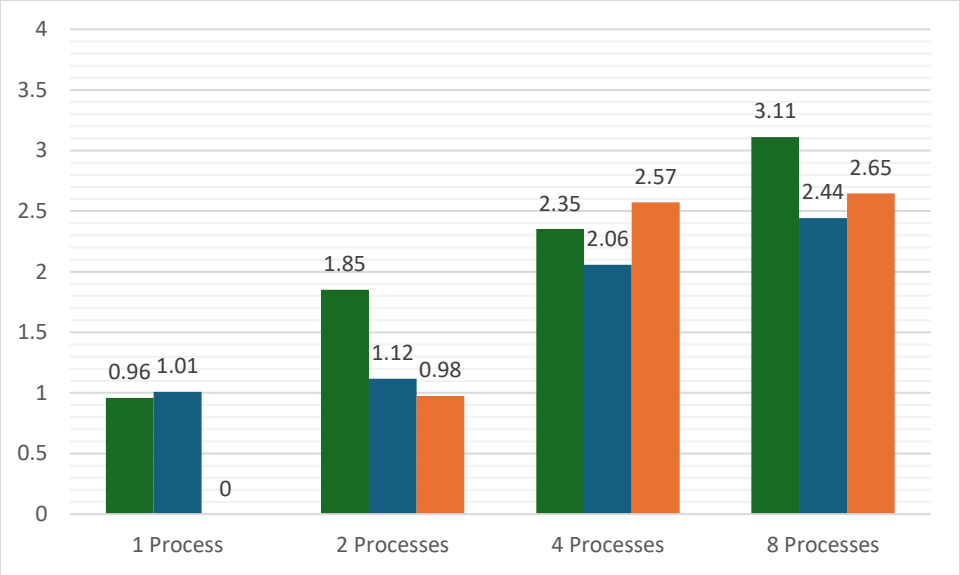
Β) Με την χρήση της βιβλιοθήκης mpi4py και το MPICommExecutor δημιουργείται ένας εκτελεστής διεργασιών για τον παραλληλισμό του gridsearch με διεργασίες MPI. Με την χρήση της executor.map, η συνάρτηση evaluate_model εκτελείται από τις διαφορετικές διεργασίες MPI με εισόδους τα pg, X_train, X_test κ.λπ. Κάθε διεργασία υπολογίζει το αποτέλεσμα για ένα υποσύνολο του πλέγματος υπερπαραμέτρων. Τέλος, η αρχική διεργασία (rank 0) συγκεντρώνει τα αποτελέσματα από όλες τις διεργασίες και τα εκτυπώνει, μαζί με το speedup που υπολογίζεται όπως παραπάνω.

Γ) Πάλι χρησιμοποιώντας την βιβλιοθήκη mpi4py, αλλά χρησιμοποιώντας μοντέλο Master-Worker, διανέμουμε τις εργασίες χωρίζοντας το Grid σε chunks (μεγέθους chunk_size) με βάση τον αριθμό των Workers (Workers - 1). Έπειτα, τα κομμάτια αποστέλλονται στους workers χρησιμοποιώντας την comm.send(). Η διεργασία με rank 0, λαμβάνει τα αποτελέσματα από τους εργάτες χρησιμοποιώντας την comm.recv(). Τέλος, εκτυπώνει τα αποτελέσματα και το speedup που υπολογίζεται όπως παραπάνω.

Παρατηρήσεις

1. Όπου κρίθηκε αναγκαίο, υπάρχουν τα απαραίτητα Comments στον κώδικα για την ευκολότερη ανάγνωση και κατανόησή του.
2. Τα παρακάτω αποτελέσματα των χρόνων εκτέλεσης και του Speedup, συλλέχθηκαν με τον υπολογισμό του μέσου όρου 5 εκτελέσεων του κάθε script για κάθε διαφορετικό αριθμό processors. Παρατηρώντας τα αποτελέσματα στο παρακάτω διάγραμμα Speedup/#Processors, μπορούμε να συμπεράνουμε πως δίνοντας περισσότερους processors για χρήση πετυχαίνουμε μεγαλύτερο speedup ειδικά με την αύξηση από 2 σε 4, κατά την χρήση MPI, ενώ η συνάρτηση multiprocessing.Pool φαίνεται να είναι optimized για την Python και έχει σχεδόν γραμμική βελτίωση ανάλογα με το πόσα processors έχει διαθέσιμα. Ενδιαφέρον προκαλεί και το Speedup με την αύξηση από 4 σε 8 processors για τα

script που κάνουν χρήση MPI, καθώς δεν βλέπουμε σημαντική αύξηση, σε αντίθεση με την αύξηση από 2 σε 4.



Ερώτημα 3

Α) Με την εντολή #pragma omp parallel for ο compiler θα παραλληλίσει την εκτέλεση του βρόχου. Οι επαναλήψεις του βρόχου θα κατανεμηθούν σε πολλαπλά threads, επιτρέποντάς τους να εκτελούνται ταυτόχρονα, καθένα από τα οποία θα εργάζεται σε διαφορετικό τμήμα του βρόχου.

(schedule(dynamic, 2)): Το όρισμα schedule ορίζει τον τρόπο με τον οποίο οι επαναλήψεις κατανέμονται μεταξύ των νημάτων. Το όρισμα dynamic κατανέμει δυναμικά τις επαναλήψεις στα νήματα κατά τον χρόνο εκτέλεσης, σε αντίθεση με το static, όπου οι επαναλήψεις κατανέμονται εκ των προτέρων και κατανέμονται ομοιόμορφα στα νήματα. Το όρισμα 2 προσδιορίζει το μέγεθος του chunk που θα αναλάβει κάθε νήμα. Δηλαδή το κάθε νήμα θα αναλαμβάνει 2 επαναλήψεις κάθε φορά.

Β) Ο κώδικας είναι ο εξής:

```
extern double work(int i);

void initialize(double *A, int N)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < N; i++) {
                #pragma omp task firstprivate(i)
                A[i] = work(i);
            }
        }
    }
}
```

Με την εντολή #pragma omp parallel, όπως και παραπάνω, ο compiler θα παραλληλίσει την εκτέλεση του βρόχου. Με την εντολή #pragma omp single εξασφαλίζεται ότι μόνο ένα νήμα εκτελεί το βρόχο for που δημιουργεί τις εργασίες. Χωρίς την οδηγία single, όλα τα νήματα θα μπορούσαν να δημιουργήσουν εργασίες, οδηγώντας σε περιττή δημιουργία εργασιών. Με την εντολή #pragma omp task firstprivate(i) ορίζουμε ότι κάθε επανάληψη του βρόχου θα είναι μία ανεξάρτητη εργασία. Το όρισμα firstprivate(i) διασφαλίζει ότι κάθε task λαμβάνει ένα private αντίγραφο της μεταβλητής i το οποίο όμως έχει αρχικοποιηθεί από το for loop, ώστε να εργάζεται στο σωστό δείκτη του πίνακα A. Κάθε task θα καλεί A[i] = work(i) ανεξάρτητα. Τέλος, αφού δημιουργηθεί ένα task, οποιοδήποτε διαθέσιμο thread μπορεί να το αναλάβει και να εκτελέσει την αντίστοιχη επανάληψη του βρόχου. Οι εργασίες εκτελούνται ασύγχρονα, πράγμα που σημαίνει ότι η επεξεργασία τους θα γίνεται παράλληλα καθώς τα threads γίνονται διαθέσιμα.

Εκτέλεση των Scripts

Για την ευκολότερη εκτέλεση των Scripts, πέρα από τα scripts, δημιουργήσαμε μέσα σε κάθε φάκελο (task1, task2), ένα Makefile το οποίο τρέχει τα scripts του κάθε ερωτήματος. Μέσα στο καθένα από αυτά, μπορείτε να αλλάξετε τον αριθμό των Processors που είναι διαθέσιμα για κάθε script, αλλάζοντας τον αριθμό των μεταβλητών N_PROCESSORS.