ΜΥΕ023 – Παράλληλα Συστήματα και Προγραμματισμός

Σετ Ασκήσεων #2

Σπυρίδωνος Βασίλειος

Το δεύτερο σετ ασκήσεων αφορά τον παράλληλο προγραμματισμό μέσω του προτύπου MPI. Ζητείται η παραλληλοποίηση 2 εφαρμογών, μία με εύρεση πρώτων αριθμών και μία με πολλαπλασιασμό πινάκων.

Όλες οι μετρήσεις έγιναν χρησιμοποιώντας τα τετραπύρηνα μηχανήματα της σχολής (opti3060wsXX, XX = 03-17). Παρακάτω, τα χαρακτηριστικά ενός από αυτά τα συστήματα:

Όνομα υπολογιστή	opti3060ws04
Επεξεργαστής	Intel i3-8300
Πλήθος πυρήνων	4
Μεταφραστής	gcc 7.5.0

Άσκηση 1

Το πρόβλημα

Στην άσκηση αυτή ζητείται να παραλληλοποιηθεί και να χρονομετρηθεί ένα σειριακό πρόγραμμα που υπολογίζει το πλήθος των πρώτων αριθμών καθώς και το μεγαλύτερο πρώτο αριθμό μέχρι και το N, για οποιοδήποτε N. Συγκεκριμένα, ζητήθηκε η κάθε διεργασία να αναλαμβάνει να ελέγχει ένα υποσύνολο των αριθμών.

Μέθοδος παραλληλοποίησης

Χρησιμοποιήθηκε το πρόγραμμα από την ιστοσελίδα του μαθήματος. Δημιουργήθηκε η συνάρτηση **mpi_primes()** όπου θα χρησιμοποιήσει τον αλγόριθμο του σειριακού προγράμματος και θα μοιράσει τη δουλεία σε διεργασίες. Αρχικά, χρησιμοποιείται η συνάρτηση **MPI_Wtime()** για να αρχίσει η χρονομέτρηση. Ακολουθούν οι συναρτήσεις **MPI_Comm_rank()** και **MPI_Comm_size()** που χρησιμεύουν για τη διαφοροποίηση των διεργασιών. Στο **myid** θα αποθηκεύται το ακολουθιακό αναγνωριστικό κάθε διεργασίας και στο **nproc** το πλήθος διεργασιών. Έπειτα, έγινε η εξής αλλαγή στην επανάληψη for του αλγορίθμου:

for (i = myid; i < (n – 1) / 2; i += nproc)

Αυτό έγινε ώστε να μοιραστεί ο φόρτος των υπολογισμών σε κάθε διεργασία. Δίνοντας στο **i** την τιμή του **myid**, η κάθε διεργασία θα ξεκινήσει την επανάληψη με διαφορετικό **i**, αφού έχουν όλες διαφορετικό αναγνωριστικό. Η επανάληψη θα σταματήσει (όπως και στο σειριακό αλγόριθμο) στο (n – 1) / 2 και το **i** θα αυξάνεται κάθε φορά κατά το πλήθος διεργασιών **nproc** ώστε κάθε διεργασία να αναλάβει τη δική της δουλειά.

Ο αλγόριθμος συνεχίζει κανονικά, μόνο που αντί για τις μεταβλητές count και lastprime χρησιμοποιούνται οι proc_count και proc_lastprime που είναι το πλήθος των πρώτων αριθμών και ο μέγιστος πρώτος αριθμός αντίστοιχα που υπολόγισε η κάθε διεργασία. Έπειτα, προστέθηκε η εντολή:

MPI_Reduce(&proc_count, &count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)

Η συνάρτηση **MPI_Reduce** αποτελεί μια συνάρτηση συλλογικής επικοινωνίας. Δουλεύει ως εξής: όλες οι μεταβλητές **proc_count** που αναπαριστούν το πλήθος των πρώτων αριθμών που υπολογίστηκαν και είναι μοναδικές σε κάθε διεργασία, θα

συνδυαστούν με πρόσθεση (εφόσον δόθηκε σαν παράμετρος το **MPI_SUM**) και το αποτέλεσμα θα σταλεί στη διεργασία 0, όπου θα αποθηκευτεί στη μεταβλητή count. Αμέσως, μετά προστέθηκε η εντολή:

MPI_Reduce(&proc_lastprime, &lastprime, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD)

Λειτουργεί με παρόμοιο τρόπο με την προηγούμενη εντολή **MPI_Reduce** που περιγράφηκε παραπάνω. Συγκεκριμένα, όλες οι μεταβλητές **proc_lastprime** που αναπαριστούν το μεγαλύτερο πρώτο αριθμό που υπολογίστηκε και είναι μοναδικές σε κάθε διεργασία θα συνδυαστούν με τον τελεστή max (εφόσον δόθηκε σαν παράμετρος το **MPI_MAX**), το αποτέλεσμα θα σταλεί στη διεργασία 0 και θα αποθηκευτεί στη μεταβλητή **lastprime.** Οπότε, τελικά στη μεταβλητή θα αποθηκευτεί ο μεγαλύτερος πρώτος αριθμός.

Έτσι λοιπόν, το πρόγραμμα «σπάει» σε κομμάτια τον υπολογισμό του πλήθους των πρώτων αριθμών και του μέγιστου πρώτου αριθμού και αυτά τα κομμάτια αναλαμβάνουν οι διεργασίες να τα εκτελέσουν. Τα αποτελέσματα τους συνδυάζονται με τις εντολές reduce που αναλύθηκαν.

Οι δύο εντολές reduce περιβάλλονται από δύο εντολές που χρησιμοποιούν την **MPI_Wtime()** και αποσκοπούν να χρονομετρήσουν πόσο χρόνο σπαταλούν οι διεργασίες για τις επικοινωνίες.

Η εντολή **count += 1** προστέθηκε καθώς η μεταβλητή **count** έχει αρχική τιμή 1. Ωστόσο, λόγω της πρώτης **MPI_Reduce** που γράφει το άθροισμα των επιμέρους πληθών πρώτων αριθμών στη μεταβλητή **count**, η τιμή 1 τελικά χάνεται οπότε πρέπει να προστεθεί στο τέλος.

Έπειτα, χρησιμοποιείται η **MPI_Wtime()** για να πάρουμε το χρόνο λήξης των υπολογισμών. Στη συνέχεια, στις μεταβλητές **proc_computations_time** και **proc_communications_time** αποθηκεύονται οι χρόνοι που σπατάλησαν οι εργασίες για τους υπολογισμούς και τις επικοινωνίες αντίστοιχα. Ακολουθούν δύο εντολές **MPI_Reduce()**, η δουλεία των οποίων είναι να συνδυάσουν τους χρόνους υπολογισμών και επικοινωνιών αντίστοιχα, με τον τελεστή της πρόσθεσης, να στείλουν τα αποτελέσματα στη διεργασία 0 και να τα αποθηκεύσουν στις μεταβλητές **total_computations_time** και **total_communications_time**, οι οποίες αναπαριστούν το συνολικό χρόνο που σπατάλησαν οι διεργασίες για υπολογισμούς και επικοινωνίες αντίστοιχα. Τέλος, η διεργασία 0 (**myid = 0**) αναλαμβάνει να τυπώσει τα αποτελέσματα της συνάρτησης **mpi_primes()**, χρησιμοποιώντας τις τιμές των μεταβλητών **count** και **lastprimes**, καθώς και τους δύο συνολικούς χρόνους που αναφέρθηκαν πριν.

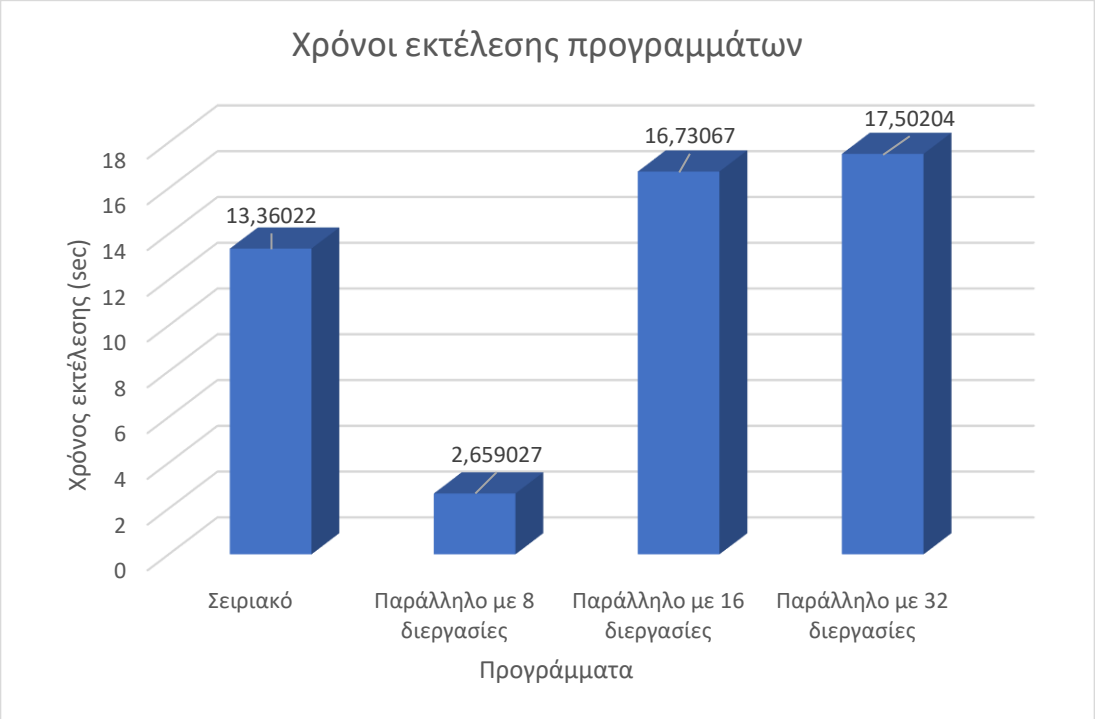
Πειραματικά αποτελέσματα – μετρήσεις

Το πρόγραμμα εκτελέστηκε στα συστήματα που αναφέρθηκαν στην εισαγωγή και η χρονομέτρηση έγινε με τη συνάρτηση **MPI_Wtime()**. Οι τρεις διαφορετικές «εικονικές» μηχανές στις οποίες εκτελέστηκε το πρόγραμμα είχαν 2, 4 και 8 υπολογιστές η κάθε μία. Επειδή τα μηχανήματα είναι τετραπύρηνα, ο αριθμός διεργασιών που δινόταν κάθε φορά σαν όρισμα στη γραμμή εντολών ήταν ίσος με τον αριθμό των πυρήνων.

Ο παρακάτω πίνακας περιέχει τα αποτελέσματα της χρονομέτρησης:

Πρόγραμμα	**1η εκτέλεση**	**2η εκτέλεση**	**3η εκτέλεση**	**4η εκτέλεση**	**Μέσος Όρος**
Σειριακό	13,35938	13,35949	13,35966	13,36235	13,36022
Παράλληλο με 8 διεργασίες	2,172492	2,829148	2,255961	3,378505	2,659027
Παράλληλο με 16 διεργασίες	16,24862	17,82621	16,23724	15,44261	16,73067
Παράλληλο με 32 διεργασίες	18,08593	14,73565	20,31765	16,86893	17,50204

Με βάση τους πίνακες των αποτελεσμάτων προκύπτει η παρακάτω γραφική παράσταση:



Συνολικός χρόνος που σπατάλησαν οι διεργασίες σε υπολογισμούς και επικοινωνίες:

8 διεργασίες

Υπολ./Επικοιν.	**1η εκτέλεση**	**2η εκτέλεση**	**3η εκτέλεση**	**4η εκτέλεση**	**Μέσος Όρος**
Υπολογισμοί	13,82402	14,86103	13,48291	16,42350	14,64787
Επικοινωνίες	4,272906	5,967574	4,465347	6,282051	5,246970

16 διεργασίες

Υπολ./Επικοιν.	**1η εκτέλεση**	**2η εκτέλεση**	**3η εκτέλεση**	**4η εκτέλεση**	**Μέσος Όρος**
Υπολογισμοί	17,32479	19,28469	19,28970	18,55973	18,61473
Επικοινωνίες	28,55824	24,52720	26,44532	29,56461	27,27385

32 διεργασίες

Υπολ./Επικοιν.	**1η εκτέλεση**	**2η εκτέλεση**	**3η εκτέλεση**	**4η εκτέλεση**	**Μέσος Όρος**
Υπολογισμοί	16,26396	15,58182	16,73719	17,12290	16,42647
Επικοινωνίες	25,52007	26,63247	33,92352	32,80717	29,72080

Σχόλια

Με βάση τα αποτελέσματα κάνουμε τις εξής παρατηρήσεις :

- Τα αποτελέσματα είναι κοντά στα ιδεώδη. Συγκεκριμένα, το παράλληλο πρόγραμμα που εκτελέστηκε στην εικονική μηχανή με 2 υπολογιστές και 8 διεργασίες είναι πολύ ταχύτερο του σειριακού προγράμματος. Είναι γύρω στις 6 φορές ταχύτερο. Το αποτέλεσμα είναι λογικό, εφόσον ο φόρτος των υπολογισμών μοιράζεται σε διεργασίες στους πυρήνες των υπολογιστών και εκμεταλλευόμαστε τα πλεονεκτήματα του παραλληλισμού.
- Τα προγράμματα που εκτελέστηκαν με 16 και 32 διεργασίες αντίστοιχα έχουν μια μικρή διαφορά στους μέσους χρόνους εκτέλεσής τους. Ωστόσο, είναι πολύ πιο αργά από το πρόγραμμα με τις 8 διεργασίες και λίγο πιο αργά

και από το σειριακό πρόγραμμα. Αυτό συμβαίνει γιατί, όπως παρατηρούμε και από τους πίνακες, οι χρόνοι που σπαταλούν οι διεργασίες για τους υπολογισμούς και τις επικοινωνίες είναι αυξημένοι στα προγράμματα με 16 και 32 διεργασίες. Περισσότερες διεργασίες πρέπει να επικοινωνήσουν μεταξύ τους, το οποίο δημιουργεί καθυστέρηση, όπως και το γεγονός ότι δημιουργούνται περισσότερες διεργασίες. Επίσης, άλλος ένας λόγος είναι η άνιση κατανομή του φόρτου υπολογισμών στις διεργασίες. Ειδικότερα, οι διεργασίες που θα αναλάβουν τις τελευταίες επαναλήψεις, είναι λογικό να αργήσουν περισσότερο να τελειώσουν καθώς ο φόρτος των υπολογισμών είναι μεγαλύτερος (πιο μεγάλες πράξεις). Έτσι, πολλές διεργασίες μπορεί να έχουν τελειώσει αλλά να πρέπει να περιμένουν τις τελευταίες.

- Μεταξύ των προγραμμάτων με 16 και 32 διεργασίες παρατηρούμε ότι οι διεργασίες του πρώτου σπαταλούν περισσότερο χρόνο στους υπολογισμούς και λιγότερο στις επικοινωνίες από το δεύτερο. Αυτό γίνεται γιατί οι διεργασίες του πρώτου προγράμματος είναι λιγότερες και τους ανατίθεται μεγαλύτερο κομμάτι υπολογισμών από αυτές του δεύτερου προγράμματος που είναι περισσότερες και αναλαμβάνουν μικρότερο κομμάτι η κάθε μία. Είναι επίσης λογικό ο χρόνος που σπαταλούν οι διεργασίες του δεύτερου και τρίτου προγράμματος για επικοινωνίες να είναι μεγαλύτερος από αυτόν του πρώτου, εφόσον περισσότερες διεργασίες πρέπει να επικοινωνήσουν μεταξύ τους.

Άσκηση 2

Το πρόβλημα

Στην άσκηση αυτή ζητείται να υλοποιηθεί και να χρονομετρηθεί ένα πρόγραμμα που υλοποιεί πολλαπλασιασμό τετραγωνικών πινάκων χρησιμοποιώντας το MPI και παραλληλοποίηση του εξωτερικού βρόχου που διατρέχει τις γραμμές. Η κάθε διεργασία αναλαμβάνει τον υπολογισμό μιάς «λωρίδας» συνεχόμενων γραμμών του αποτελέσματος. Οι πίνακες έχουν 1024 x 1024 ακέραια στοιχεία και δίνονται στην ιστοσελίδα του μαθήματος.

Μέθοδος παραλληλοποίησης

Αρχικά, χρησιμοποιείται η συνάρτηση **MPI_Wtime()** για να αρχίσει η χρονομέτρηση. Ακολουθούν οι συναρτήσεις **MPI_Comm_rank()** και **MPI_Comm_size()** που χρησιμεύουν για τη διαφοροποίηση των διεργασιών. Στο **myid** θα αποθηκεύται το ακολουθιακό αναγνωριστικό κάθε διεργασίας και στο **nproc** το πλήθος διεργασιών. Έπειτα, έγινε η εξής αλλαγή στον εξωτερικό βρόχο (αυτόν που διατρέχει τις γραμμές) του αλγορίθμου:

for (i = myid * N / nproc; i < (myid + 1) * N / nproc; i++)

Αυτό έγινε για την υλοποίηση της μεθόδου διαχωρισμού γραμμών (strip partitioning), ώστε κάθε διεργασία να αναλάβει να υπολογίσει τη δικιά της «λωρίδα» συνεχόμενων γραμμών. Δίνοντας στο **i** την τιμή **myid * N / nproc**, η κάθε διεργασία θα ξεκινήσει την επανάληψη με διαφορετικό **i**, αφού έχουν όλες διαφορετικό αναγνωριστικό. Σε κάθε μία διεργασία θα δοθεί ο αριθμός γραμμών που της αντιστοιχεί, ανάλογα και με το πλήθος διεργασιών. Η σταθερά N ισούται με 1024. Η επανάληψη θα σταματήσει πριν το **(myid + 1) * N / nproc**, όπου η τελευταία διεργασία θα αναλάβει τον τελευταίο αριθμό γραμμών και το **i** θα αυξάνεται κάθε φορά κατά 1. Με αυτόν τον τρόπο οι διεργασίες θα αναλάβουν ισάξια δουλειά και μοναδική δουλειά.

Ο αλγόριθμος συνεχίζει κανονικά, μέχρι και την εντολή **C[i][j] = sum**. Στη συνέχεια, η διεργασία 0 (**myid == 0)** καλεί επαναληπτικά (τόσες φορές όσες και το πλήθος των διεργασιών – 1, επειδή η ίδια δε στέλνει κάτι):

MPI_Recv(array, 3, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status)

, όπου **array** ένας πίνακας (αρχικοποιείται παρακάτω) που περιέχει 3 στοιχεία: τον αριθμό γραμμής **i**, τον αριθμό στήλης **j** και το άθροισμα **sum** που υπολογίζεται στον τρίτο βρόχο του αλγορίθμου πολλαπλασιασμού πινάκων. Τα στοιχεία αυτά του πίνακα παίρνουν την τιμή τους παρακάτω, στο **else**, δηλαδή για τις υπόλοιπες διεργασίες που δεν έχουν **myid = 0**. Αυτές, στέλνουν κάθε φορά τον πίνακα **array** στη διεργασία 0 χρησιμοποιώντας την εντολή:

MPI_Send(array, 3, MPI_INT, 0, 1, MPI_COMM_WORLD)

Όπως αναφέρθηκε και παραπάνω, η διεργασία 0 θα λαμβάνει τους πίνακες **array** όλων των διεργασιών και εφόσον λαμβάνει και τα **i, j** θα ξέρει σε ποια θέση του πίνακα αποτελέσματος να αποθηκεύσει το **sum**. Αυτό γίνεται (μέσα στο μπλοκ κώδικα του **if**) με την εντολή **C[array[0]][array[1]] = array[2]**, όπου **array[0] = i**, **array[1] = j** και **array[2] = sum.**

Το παραπάνω μπλοκ κώδικα if-else περιβάλλεται από δύο εντολές που χρησιμοποιούν την **MPI_Wtime()** και αποσκοπούν να χρονομετρήσουν πόσο χρόνο σπαταλούν οι διεργασίες για τις επικοινωνίες. Σε κάθε διεργασία, στη μεταβλητή **proc_communications_time** προστίθεται σε κάθε επανάληψη ο χρόνος που χρειάστηκε για την επικοινωνία των διεργασιών λόγω των εντολών **MPI_Send** και **MPI_Recv**.

Αφού υπολογιστεί για την κάθε διεργασία η τιμή της **proc_communications_time** **proc_communications_time**, χρησιμοποιείται για τον υπολογισμό της **proc_computations_time** που αναπαριστά το χρόνο που σπατάλησε η κάθε διεργασία για υπολογισμούς.

Ακολουθούν δύο εντολές **MPI_Reduce()**, η δουλεία των οποίων είναι να συνδυάσουν τους χρόνους υπολογισμών και επικοινωνιών αντίστοιχα, με τον τελεστή της πρόσθεσης, να στείλουν τα αποτελέσματα στη διεργασία 0 και να τα αποθηκεύσουν στις μεταβλητές **total_computations_time** και **total_communications_time**, οι οποίες αναπαριστούν το συνολικό χρόνο που σπατάλησαν οι διεργασίες για υπολογισμούς και επικοινωνίες αντίστοιχα. Έπειτα, η διεργασία 0 (**myid = 0**) αναλαμβάνει να τυπώσει το χρόνο εκτέλεσης του προγράμματος, καθώς και τους δύο συνολικούς χρόνους που αναφέρθηκαν πριν.

Τέλος, η διεργασία 0 (**myid = 0**) θα αναλάβει να γράψει τα αποτελέσματα στο αρχείο Cmat1024.

<u>Πειραματικά αποτελέσματα – μετρήσεις</u>

Το πρόγραμμα εκτελέστηκε στα συστήματα που αναφέρθηκαν στην εισαγωγή και η χρονομέτρηση έγινε με τη συνάρτηση **MPI_Wtime()**. Οι τρεις διαφορετικές «εικονικές» μηχανές στις οποίες εκτελέστηκε το πρόγραμμα είχαν 2, 4 και 8 υπολογιστές η κάθε μία. Επειδή τα μηχανήματα είναι τετραπύρηνα, ο αριθμός διεργασιών που δινόταν κάθε φορά σαν όρισμα στη γραμμή εντολών ήταν ίσος με τον αριθμό των πυρήνων.

Για να ελέγξω αν τα αποτελέσματα είναι ορθά, χρησιμοποίησα στο τερματικό την εντολή :

diff x y

, όπου **x** : το αρχείο με τα σωστά αποτελέσματα του σειριακού προγράμματος

και **y** : το αρχείο με τα αποτελέσματα του προγράμματος που υλοποίησα.

Ο παρακάτω πίνακας περιέχει τα αποτελέσματα της χρονομέτρησης:

Πρόγραμμα	**1η εκτέλεση**	**2η εκτέλεση**	**3η εκτέλεση**	**4η εκτέλεση**	**Μέσος Όρος**
Σειριακό	3,489074	3,489569	3,490294	3,447243	3,479045
Παράλληλο με 8 διεργασίες	1,691849	1,636658	1,514031	1,535291	1,594457
Παράλληλο με 16 διεργασίες	1,639683	1,812144	1,646529	1,831500	1,732464
Παράλληλο με 32 διεργασίες	1,635117	1,506328	1,435877	1,369724	1,486762

Με βάση τους πίνακες των αποτελεσμάτων προκύπτει η παρακάτω γραφική παράσταση:

Συνολικός χρόνος που σπατάλησαν οι διεργασίες σε υπολογισμούς και επικοινωνίες:

8 διεργασίες

Υπολ./Επικοιν.	**1η εκτέλεση**	**2η εκτέλεση**	**3η εκτέλεση**	**4η εκτέλεση**	**Μέσος Όρος**
Υπολογισμοί	7,153653	6,571627	6,779315	6,847472	6,838017
Επικοινωνίες	3,895815	3,808811	3,380081	3,043037	3,531936

16 διεργασίες

Υπολ./Επικοιν.	**1η εκτέλεση**	**2η εκτέλεση**	**3η εκτέλεση**	**4η εκτέλεση**	**Μέσος Όρος**
Υπολογισμοί	10,453526	9,117559	9,832568	9,216730	9,655100
Επικοινωνίες	7,421165	9,078160	6,906910	7,723722	7,782489

32 διεργασίες

Υπολ./Επικοιν.	1η εκτέλεση	2η εκτέλεση	3η εκτέλεση	4η εκτέλεση	Μέσος Όρος
Υπολογισμοί	7,135341	7,101420	8,157925	7,722486	7,529293
Επικοινωνίες	20,39192	22,46629	20,00364	19,53226	20,59853

Σχόλια

Με βάση τα αποτελέσματα κάνουμε τις εξής παρατηρήσεις :

- Τα αποτελέσματα είναι κοντά στα ιδεώδη, καθώς τα παραλληλοποιημένα προγράμματα είναι πιο γρήγορα από το σειριακό. Λογικό, εφόσον ο φόρτος των υπολογισμών μοιράζεται σε διεργασίες στους πυρήνες των υπολογιστών και εκμεταλλευόμαστε τα πλεονεκτήματα του παραλληλισμού. Επίσης, παρατηρούμε ότι οι χρόνοι των παράλληλων προγραμμάτων είναι περίπου το 1/2 του χρόνου του σειριακού προγράμματος.
- Τα παράλληλα προγράμματα έχουν μικρές διαφορές μεταξύ τους όσον αφορά το χρόνο εκτέλεσης. Από τα τρία, φαίνεται ότι το πρόγραμμα με τις 32 διεργασίες είναι λίγο πιο γρήγορο. Αυτό είναι λογικό καθώς οι πολλές διεργασίες υπολογίζουν αρκετά πιο γρήγορα το τελικό αποτέλεσμα, αφού ο αρχικός φόρτος μοιράζεται σε μικρά κομμάτια στις διεργασίες.
- Μεταξύ των προγραμμάτων με 16 και 32 διεργασίες παρατηρούμε ότι οι διεργασίες του πρώτου σπαταλούν περισσότερο χρόνο στους υπολογισμούς και λιγότερο στις επικοινωνίες από το δεύτερο. Αυτό γίνεται γιατί οι διεργασίες του πρώτου προγράμματος είναι λιγότερες και τους ανατίθεται μεγαλύτερο κομμάτι υπολογισμών από αυτές του δεύτερου προγράμματος που είναι περισσότερες και αναλαμβάνουν μικρότερο κομμάτι η κάθε μία. Είναι επίσης λογικό ο χρόνος που σπαταλούν οι διεργασίες του δεύτερου και τρίτου προγράμματος για επικοινωνίες να είναι μεγαλύτερος από αυτόν του πρώτου, εφόσον περισσότερες διεργασίες πρέπει να επικοινωνήσουν μεταξύ τους.

Άσκηση 3

Το πρόβλημα

Για την άσκηση αυτή έπρεπε να αναζητηθούν πληροφορίες για το τι ακριβώς είναι οι μονόπλευρες επικοινωνίες, τι παρέχει το MPI και πώς χρησιμοποιούνται σε μια εφαρμογή. Ζητήθηκε, επιπλέον, να γραφτεί ένας μικρός οδηγός για τις μονόπλευρες επικοινωνίες.

Οδηγός

Οι μονόπλευρες επικοινωνίες (one-sided) είναι ένα χρήσιμο και πολλά υποσχόμενο εργαλείο για επικοινωνία υψηλής επίδοσης σε δίκτυα με χαμηλή καθυστέρηση, το οποίο χρησιμοποιείται εκτενώς τα τελευταία χρόνια. Η διαφορά και το πλεονέκτημα των μονόπλευρων επικοινωνιών σε σχέση με τις αμφίπλευρες (two-sided ή point to point) είναι ότι οι μονόπλευρες είναι σχετικά ασύγχρονες. Συγκεκριμένα, ενώ στις αμφίπλευρες επικοινωνίες ο αποστολέας και ο παραλήπτης ανταλλάσσουν συνεχώς μηνύματα καλώντας συναρτήσεις τύπου send και receive, στις μονόπλευρες επικοινωνίες, μόνο η αρχική διεργασία καλεί συναρτήσεις (put, get) για τη μεταφορά δεδομένων, η οποία συντελείται χωρίς να απαιτείται από τη διεργασία-«στόχο» να καλέσει κάποια συνάρτηση, χωρίς να χρειάζεται δηλαδή, να συγχρονιστεί με την αρχική διεργασία. Με αυτόν τον τρόπο, δεν είναι υποχρεωτικό τα παράλληλα προγράμματα να είναι πλήρως συγχρονισμένα και παρατηρείται αυξημένη επίδοση όσον αφορά τη μεταφορά δεδομένων από και προς διεργασίες.

Στις μονόπλευρες επικοινωνίες, κάθε διεργασία καταστεί διαθέσιμο ένα κομμάτι της μνήμης της στις άλλες διεργασίες. Αυτές, μπορούν να διαβάσουν ή να γράψουν απευθείας σε αυτό το κομμάτι μνήμης. Έτσι, οι μονόπλευρες επικοινωνίες ονομάζονται και RMA (Remote Memory Access), καθώς υπάρχει πρόσβαση σε απομακρυσμένες μνήμες. Κάθε διεργασία μπορεί να καθορίσει εξολοκλήρου ποια δεδομένα πρέπει να μεταφερθούν στην ίδια από τις «γειτονικές» της διεργασίες. Εναλλακτικά, κάθε διεργασία μπορεί να καθορίσει τι δεδομένα χρειάζονται οι γειτονικές της διεργασίες.

Κάθε μορφή μνήμης που δημιουργείται από μία διεργασία είναι εξ' ορισμού προσβάσιμη μόνο τοπικά, όπως για παράδειγμα η μνήμη που δεσμεύεται χρησιμοποιώντας την εντολή malloc. Για να μπορέσουν, λοιπόν, οι υπόλοιπες

διεργασίες να διαβάσουν ή να γράψουν σε αυτή τη μνήμη, πρέπει ο προγραμματιστής να επιτρέψει σε αυτή τη μνήμη να είναι προσβάσιμη από τις άλλες διεργασίες, χρησιμοποιώντας εργαλεία του MPI. Μία τέτοια μνήμη ονομάζεται window.

Η εντολή που καταστεί μια τοπική μνήμη προσβάσιμη από άλλες διεργασίες είναι η **MPI_Win_create**, η οποία δημιουργεί ένα καινούργιο window. Συντάσσεται ως εξής:

int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_WIN *win)

, όπου:

- **base:** δείκτης στην τοπική μνήμη που θέλουμε να γίνει διαθέσιμη για διάβασμα/γράψιμο.
- **size:** το μέγεθος της μνήμης σε bytes.
- **disp_unit:** χρησιμεύει για windows που περιλαμβάνουν πίνακες με στοιχεία. Συνήθως, χρησιμοποιείται το sizeof(τύπος).
- **info:** παρέχει πληροφορίες βελτιστοποίησης την ώρα της εκτέλεσης. Συνήθως χρησιμοποιείται το MPI_INFO_NULL.
- **comm:** ο communicator, δηλαδή ένα αντικείμενο που περιγράφει ένα σύνολο από διεργασίες. Συνήθως χρησιμοποιείται το MPI_COMM_WORLD.
- **win:** το αντικείμενο window που επιστρέφει η συνάρτηση.

Εναλλακτικά, υπάρχουν και οι εξής παρόμοιες εντολές με την **MPI_Win_create: MPI_Win_allocate, MPI_Win_create_dynamic** και **MPI_Win_allocate_shared**, οι οποίες δε θα αναλυθούν περαιτέρω.

Για να ελευθερωθεί ο χώρος που καταλαμβάνει το window, χρησιμοποιείται η εντολή **MPI_Win_free(&win)**.

Παρακάτω, αναφέρονται 3 εντολές που παρέχει το MPI για μεταφορά δεδομένων στις μονόπλευρες επικοινωνίες:

- MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_dtype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_dtype, MPI_Win win) : μεταφέρει δεδομένα από την αρχική διεργασία στη διεργασία-στόχο.
- MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_dtype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_dtype, MPI_Win win) : μεταφέρει δεδομένα από τη διεργασία-στόχο στην αρχική.
- MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype origin_dtype, int target_rank, MPI_Aint target_disp, int target_count,

MPI_Datatype target_dtype, MPI_Op op, MPI_Win win) : ανανεώνει τα δεδομένα σε απομακρυσμένη μνήμη χρησιμοποιώντας τοπικές μεταβλητές.

, όπου:

- origin_addr: η διεύθυνση του αρχικού buffer
- origin_count: το πλήθος των εγγραφών στον αρχικό buffer
- origin_dtype: ο τύπος δεδομένων κάθε εγγραφής στον αρχικό buffer
- target_rank: η τάξη (rank) της διεργασίας-στόχου
- target_disp: χρησιμεύει για windows που περιλαμβάνουν πίνακες με στοιχεία. Συνήθως, χρησιμοποιείται το sizeof(τύπος).
- target_count: το πλήθος των εγγραφών του buffer-στόχου
- target_dtype: ο τύπος δεδομένων κάθε εγγραφής στον buffer-στόχο
- op: η πράξη με την οποία θα συνδυαστούν (reduce) τα δεδομένα
- win: το αντικείμενο window

Από τα παραπάνω, προκύπτουν κάποια ζητήματα συγχρονισμού όπως: Πότε επιτρέπεται σε μία διεργασία να διαβάσει/γράψει σε μία απομακρυσμένη μνήμη; Ή πότε θα είναι διαθέσιμα τα δεδομένα που έγραψε μία διεργασία Α σε μια μνήμη, ώστε να τα διαβάσει μία διεργασία Β; Για την επίλυση αυτών των ζητημάτων, το MPI περιέχει τρία μοντέλα υλοποίησης συγχρονισμού: **Fence, Post-start-complete-wait, Lock/Unlock**. Θα περιγραφεί το μοντέλο **Fence**.

Για την υλοποίηση συγχρονισμού με το μοντέλο Fence υπάρχει η εντολή MPI_Win_fence(int assert, MPI_Win win), όπου:

- assert: χρησιμεύει για τη βελτιστοποίηση της εντολής. Εναλλακτικά, μπορεί να χρησιμοποιηθεί και η τιμή 0.
- win: το αντικείμενο window.

Όταν μια διεργασία καλέσει την παραπάνω συνάρτηση και αυτή επιστρέψει, η διεργασία μπορεί να εκτελέσει μονόπλευρες (one-sided) εντολές στο συγκεκριμένο κομμάτι μνήμης (window). Τέτοιες εντολές μπορεί να είναι οι get, put και accumulate που περιγράφηκαν παραπάνω. Αυτή η πρώτη κλήση της συνάρτησης, εξασφαλίζει ότι όλες οι διεργασίες που την κάλεσαν, έχουν φτάσει στο πρώτο “fence” και έτσι, πλέον, η πρόσβαση στα windows τους από τις άλλες διεργασίες είναι ασφαλής. Για παράδειγμα, το μοντέλο Fence απαγορεύει σε μια διεργασία Α να προσβεί και να εκτελέσει μονόπλευρες εντολές στο window μιας διεργασίας Β, αν αυτή δεν έχει καλέσει **MPI_Win_fence**, δηλαδή δεν έχει φτάσει στο πρώτο fence.

Έπειτα, όταν γίνει ξανά κλήση της συνάρτησης, αυτό θα σημάνει την ολοκλήρωση των εργασιών που έκανε η κάθε διεργασία, καθώς και την ολοκλήρωση των εργασιών που έκαναν οι υπόλοιπες διεργασίες στο window της συγκεκριμένης

διεργασίας. Η δεύτερη κλήση της συνάρτησης εγγυάται ότι μια διεργασία θα επιστρέψει από το δεύτερο fence αν και μόνο αν, όλες οι υπόλοιπες διεργασίες έχουν σταματήσει τις εργασίες τους στο window της, δηλαδή αν και μόνο αν σταματήσουν να έχουν πρόσβαση σε αυτό.

Ενδεικτικά, ένα παράδειγμα σειράς χρήσης όλων των εντολών που περιγράφηκαν παραπάνω σε μία διεργασία είναι: MPI_Win_create, MPI_Win_fence, εντολές MPI_Put/Get/Accumulate, MPI_Win_fence, MPI_Win_free.

Πηγές

An Evaluation of Implementation Options for MPI One-Sided Communication William Gropp and Rajeev Thakur

https://www.mcs.anl.gov/~thakur/papers/rma-impl.pdf

Advanced MPI: I/O and One-Sided Communication William Gropp, Rusty Lusk, Rob Ross, and Rajeev Thakur

https://www.mcs.anl.gov/research/projects/mpi/tutorial/advmpi/sc2005-advmpi.pdf

One-sided Communication in MPI - William Gropp
http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture34.pdf

One-sided Communication with MPI-2 Rolf Rabenseifner
https://fs.hlrs.de/projects/par/par_prog_ws/pdf/mpi_1sided_1.pdf

mpi-forum.org

https://www.mpi-forum.org

mpich.org

https://www.mpich.org/