Εργαστήριο Παράλληλου Υπολογισμού - Εργασία 1

Χρήστος Μαργιώλης

Νοέμβριος 2020

Περιεχόμενα

1 Τεκμηρίωση

Το πρόγραμμα εν ολίγοις έχει την εξής επαναληπτική δομή:

- Διαβάζονται στον επεξεργαστή 0 τα δεδομένα από τον χρήστη
- Διαμοιράζει στους υπόλοιπους επεξεργαστές τα δεδομένα που δώθηκαν
- Γίνονται οι κατάλληλες συγκρίσεις και έλεγχοι
- Ο επεξεργαστής 0 συλλέγει όλα τα αποτελέσματα και εμφανίζει αν ο πίνακας εν τέλει είναι ταξινομημένος ή όχι, και αν όχι, σε ποιο στοιχείο χάλασε η ταξινόμηση
- Εμφανίζει ένα μενού επιλογών ώστε να συνεχίσει ή να τερματιστεί το πρόγραμμα

Είναι σημαντικό να εξηγηθεί το πώς ισοκατανέμεται ο υπολογιστικός φόρτος σε όλους τους p επεξεργατές. Αφού διαβαστεί το N, δηλαδή το πόσα στοιχεία έχει η ακολουθία T, το διαιρούμε δια όσους επεργαστές έχουμε. Αρχικά, ας υποθέσουμε ότι το N ειναι ακέραιο πολλαπλάσιο του p, και ότι έχουμε - αν το N για παράδειγμα είναι 10 και οι επεξεργαστές 2, τότε $N/p = 10/2 = 5$ στοιχεία από τον συνολικό πίνακα για κάθε επεξεργαστή.

Πρέπει όμως να καλύψουμε και την περίπτωση που το N δεν είναι ακέραιο πολλαπλάσιο του p, για παράδειγμα $N = 7$ και $p = 2$. Στην περίπτωση αυτή θα πάρουμε το αποτέλεσμα της πράξης $N \mod p$ η οποία θα μάς δώσει τον αριθμό των περισσευούμενων στοιχείων που πρέπει να κατανεμηθούν.

Όταν ξεκινάει το loop για την διαμοίραση του υπολογιστικού φόρτου, πρέπει κάθε φορά να υπολογίζουμε το μέγεθος του buffer που θέλουμε να σταλεί στον εκάστοτε επεξεργαστή. Έπειτα υπολογίζουμε το offset - δηλαδή από ποιά θέση του αρχικού πίνακα και μετά - θα στείλουμε, παίρνοντας πάντα υπόψη τα τυχόν περισσευούμενα στοιχεία (αν υπάρχουν). Αν τύχει και υπάρχουν περισσευούμενα, κάποιοι επεργαστές θα παραπάνω στοιχεία μέχρι να έχει καλυφθεί ο αριθμός τον περισσευούμενων στοιχείων. Πρέπει να τονιστεί ότι έχω φροντίσει οι επεξεργατές να έχουν διαφορά εώς ενα στοιχείο όσο αφορά την διαμοίραση, δηλαδή αν υποτοθεί ότι $N = 5$ και $p = 2$ πρέπει να αποφύγουμε για παράδειγμα το να έχει ο p_0 4 στοιχεία και ο p_1 1 στοιχείο - θέλουμε ο p_0 να έχει 3 στοιχεία και ο p_1 2.

Αφού υπολογίσουμε το offset, βρίσκουμε ποιό είναι το τελευταίο στοιχείο του προηγούμενου επεξεργαστή ώστε να μπορούμε να το συγκρίνουμε με το πρώτο στοιχείο του επόμενου. Για να γίνει κατανοητό το γιατί χρειάζομαστε αυτόν τον επιπλέον έλεγχο, ας υποθέσουε ότι έχουμε την ακολουθία

$$1, 2, 6, 3, 4, 5, 1, 2, 4, 1, 2, 3$$

Για $p = 4$ αυτή η ακολουθία θα μοιραστεί ως εξής

$$p_0 = 1, 2, 6$$

$$p_1 = 3, 4, 5$$

$$p_2 = 1, 2, 4$$

$$p_3 = 1, 2, 3$$

Στην περίπτωση που δεν πάρουμε υπόψη το τελευταίο στοιχείο του προηγούμενου επεξεργαστή, βλέπουμε ότι όλοι οι επεξεργαστές θα μάς επιστρέψουν πως οι υπο-ακολουθίες τους είναι ταξινομημένες, ενώ στον γενικό πίνακα, προφανώς, δεν είναι. Όμως, με τον τρόπο που προανέφερα, μπορούμε να είμαστε σίγουροι ότι ο κάθε επεξεργαστής έχει εις γνώσην του τί υπάρχει πριν από αυτόν, και χρειάζεται να γνωρίζει μόνο το τελευταίο στοιχείο του προηγούμενού του, εφόσον μονο στις θέσεις $p_n[end]$ με $p_{n+1}[0]$ μπορεί να υπάρξει λάθος ταξινόμηση και να μην το κατάλαβει το πρόγραμμα.

Αφού ο επεξεργαστής 0 στείλει στους υπόλοιπους επεξεργαστές τα κατάλληλα δεδομένα, θα ξεκινήσουν από όλους τους p επεξεργαστές οι συγκρίσεις ώστε να δούμε αν η κάθε ακολουθία ήτανε ταξινομημένη. Η μία περίπτωση στην οποία η ακολουθία δεν θα είναι ταξινομημένη είναι να ισχύει η συνθήκη $T_i > T_{i+1}$. Η άλλη περίπτωση είναι να ισχύει ότι $T_i < prev$, δηλαδή το τελευταίο στοιχείο του προηγούμενου επεξεργαστή να είναι μεγαλύτερο από το τρέχον στοιχείο στην ακολουθία. Σημείωση ότι αυτή η περίπτωση ελέγχεται μόνο στην περίπτωση που $\delta\epsilon\nu$ είμαστε στον επεξεργαστή 0, εφόσον αυτός δεν έχει κάποιον προηγούμενο επεξεργαστή.

Τα αποτελέσματα συλλέγονται όλα στον επεξεργαστή 0, ο οποίος έχει μερικές επιπλέον μεταβλητές που θα καθορίσουν τα τελικά αποτελέσματα. Η πιο σημαντική είναι η `f_sorted` - αυτή η μεταβλητή θα μάς πει στο τέλος αν ο γενικός πίνακας ήτανε ταξινομημένος. Για να υπολογιστεί κάτι τέτοιο, ενώνουμε με AND όλες τις `sorted` flags που προέκυψαν από τις συγκρίσεις προηγούμενως. Ο λόγος που επέλεξα την πράξη AND είναι γιατί αν έστω και μία από τις `sorted` έτυχε να είναι 0, τότε και η `f_sorted` θα γίνει 0, ασχέτως του αν όλες οι άλλες `sorted` ήτανε 1 ή όχι.

Τέλος, αφού βρεθεί αν ο πίνακας είναι τελικά ταξινομημένος ή όχι, ο επεξεργαστής 0 θα τυπώσει ένα κατάλληλο μήνυμα. Αν ο πίνακας δεν είναι ταξινομημένος, θα τυπώσει και ποιο είναι το πρώτο στοχείο που χάλασε την ταξινόμηση. Έπειτα θα εμφανιστεί το μενού επιλογών.

2 Κώδικας

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define TAG_T           1
#define TAG_N           2
#define TAG_BUFSIZE     3
#define TAG_MENU        4
#define TAG_SORTED      5
#define TAG_VAL         6
#define TAG_POS         7
#define TAG_PREV        8

int
main(int argc, char *argv[])
{
        MPI_Status status;
        int nproc, rank, rc;
        int *t, n, prev;
```

```
int bufsize, offset, remaining;
int val, sorted;        /* results from each process */
int f_val, f_sorted;    /* final results */
int found = 0;          /* indicates that the first unsorted element has been found */
int i, ch = 1;

/* just in case an error occurs during initialization */
if ((rc = MPI_Init(&argc, &argv)) != 0) {
        fprintf(stderr, "%s: cannot intialize MPI.\n", argv[0]);
        MPI_Abort(MPI_COMM_WORLD, rc);
}

/* count how many processes we have running */
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);

/* main loop */
while (ch != 2) {
        if (rank == 0) {
                printf("Enter N: \n");
                scanf("%d", &n);
                getchar();

                t = malloc(n * sizeof(int));

                /* read whole array */
                for (i = 0; i < n; i++) {
                        printf("t[%d]: \n", i);
                        scanf("%d", &t[i]);
                }
                getchar();

                /* reset flag */
                sorted = 1;
                /* see if there are any remaining elements */
                remaining = n % nproc;

                /* calculate offsets and pass sub-array to each processor */
                for (i = 1; i < nproc; i++) {
                        /* calculate bufsize as if there are remaining == 0 */
                        bufsize = n / nproc;
                        /*
                         * calculate which part of the array each process will get.
                         * the last parts makes sure that if there are any
                         * remaining elements, the offset will adapt properly
                         */
```

```
                        offset = i * bufsize + (i - 1 < remaining ? i : remaining);
                        /* if there are any remaining elements, increment bufsize */
                        if (i < remaining)
                                bufsize++;

                        /* each process gets the last element of the previous one */
                        prev = t[offset - 1];

                        /* send sub-array along with its size and sorted flag */
                        MPI_Send(&bufsize, 1, MPI_INT, i, TAG_BUFSIZE, MPI_COMM_WORLD);
                        MPI_Send(t + offset, bufsize, MPI_INT, i, TAG_T, MPI_COMM_WORLD
);
                        MPI_Send(&prev, 1, MPI_INT, i, TAG_PREV, MPI_COMM_WORLD);
                        MPI_Send(&sorted, 1, MPI_INT, i, TAG_SORTED, MPI_COMM_WORLD);
                }

                /*
                 * calculate bufsize for proc 0 in and increment
                 * in case there are still remaining elements
                 */
                bufsize = n / nproc;
                if (remaining != 0)
                        bufsize++;
        } else {
                /*
                 * receive sub-arrays and sorted flag and allocate memory for each
                 * process' array
                 */
                MPI_Recv(&bufsize, 1, MPI_INT, 0, TAG_BUFSIZE, MPI_COMM_WORLD, &status)
;
                t = malloc(bufsize * sizeof(int));
                MPI_Recv(t, bufsize, MPI_INT, 0, TAG_T, MPI_COMM_WORLD, &status);
                MPI_Recv(&prev, 1, MPI_INT, 0, TAG_PREV, MPI_COMM_WORLD, &status);
                MPI_Recv(&sorted, 1, MPI_INT, 0, TAG_SORTED, MPI_COMM_WORLD, &status);
        }

        /* check if array is unsorted */
        for (i = 0; i < bufsize - 1; i++) {
                if (t[i] > t[i + 1]) {
                        val = t[i];
                        sorted = 0;
                        break;
                /*
                 * we check if the last element of the previous process
                 * is greater than t[i] so that each process doesn't compare
                 * only against its own elements. we make sure that the rank
```

```
                 * is not 0, since it doesn't have a previous rank.
                 */
                } else if (t[i] < prev && rank != 0) {
                        val = prev;
                        sorted = 0;
                        break;
                }
        }

        /* we're done with t, free everything */
        free(t);

        if (rank == 0) {
                /* collect results from proc 0 first */
                f_sorted = sorted;
                f_val = val;
                /* if f_sorted is false already, don't bother searching below */
                found = f_sorted == 0;

                /* receive results from the rest */
                for (i = 1; i < nproc; i++) {
                        MPI_Recv(&val, 1, MPI_INT, i, TAG_VAL, MPI_COMM_WORLD, &status)
;
                        MPI_Recv(&sorted, 1, MPI_INT, i, TAG_SORTED, MPI_COMM_WORLD, &
status);
                        /*
                         * AND all flags to determine whether the array is sorted.
                         * if one of the flags is 0, res will be 0 even if the
                         * rest of the flags are 1
                         */
                        f_sorted &= sorted;
                        /*
                         * get only the first unsorted element, we don't care about
                         * the rest, if any
                         */
                        if (!sorted && !found) {
                                f_val = val;
                                found = 1;
                        }
                }

                if (f_sorted)
                        puts("Array␣is␣sorted.");
                else
                        printf("Array␣is␣not␣sorted:␣first␣unsorted␣element:␣%d\n",
f_val);
```

```
                puts("Press␣[ENTER]␣to␣continue.␣.␣.");
                getchar();
        } else {
                /* send results to processor 0 */
                MPI_Send(&val, 1, MPI_INT, 0, TAG_VAL, MPI_COMM_WORLD);
                MPI_Send(&sorted, 1, MPI_INT, 0, TAG_SORTED, MPI_COMM_WORLD);
        }

        /* menu */
        if (rank == 0) {
                system("clear␣||␣cls");
                printf("1.␣Continue\n2.␣Exit\nYour␣choice:␣\n\n");
                scanf("%d", &ch);
                /* everyone has to know what the choice is */
                for (i = 1; i < nproc; i++)
                        MPI_Send(&ch, 1, MPI_INT, i, TAG_MENU, MPI_COMM_WORLD);
        } else
                MPI_Recv(&ch, 1, MPI_INT, 0, TAG_MENU, MPI_COMM_WORLD, &status);
    }

    MPI_Finalize();

    return 0;
}
```

3 Προβλήματα

Επειδή έκανα την ανάπτυξη του κώδικα κατά βάση σε FreeBSD, αλλά έκανα και tests σε Arch Linux, παρατήρησα ότι στα Linux υπάρχει πιθανότητα η `printf()` να μην βγάζει output, ή αν βγάζει, να μην εμφανίζεται με την σωστή σειρά, ή ακόμα και να κρεμάει όλο το πρόγραμμα. Σε διάφορες αναζητήσεις είδα ότι αρκετοί πρότειναν την συνάρτηση `fflush(stdout)` αλλά δεν είχα ιδιαίτερη τύχη με αυτή. Αυτό που δούλεψε ήτανε να βάλω `getchar()` όπου έβλεπα ότι χρειαζόταν, και να προσθέσω newlines σε όλες τις `printf()` που κανονικά δεν είχανε.

Ένα άλλο πρόβλημα που αντιμετώπισα ήτανε ότι όταν πρωτοέφτιαξα το menu επιλογών, δεν έστελνα στους υπόλοιπους επεξεργαστές την επιλογή που έδινα στον επεξεργαστή 0, το οποίο σήμαινε ότι οι υπόλοιποι επεξεργατές δεν ήξεραν τί είχα όντως απαντήσει, οπότε μπορεί να εκτελούσαν και αυτοί το μενού, και γενικώς υπήρχε περίεργη συμπεριφορά. Το διόρθωσα αυτό βάζοντας ένα απλό `MPI_Send()` και `MPI_Recv()` αφού δώσω την επιλογή στον επεξεργαστή 0, ωστέ όλοι οι υπόλοιποι να γνωρίζουν ότι δώθηκε απάντηση, καθώς και ποια ήτανε αυτή.

Δεν κατάφερα να βρω έναν καλό τρόπο ώστε να μπορώ να υπολογίσω την θέση του στοιχείου που χαλάει την ταξινόμηση στον γενικό πίνακα. Δηλαδή, για παράδειγμα αν τύχαινε στον επεξεργαστή 2 να χαλάσει το στοιχείο 1, θα έπρεπε να υπολογίσω γενικώς στον συνολικό πίνακα ποια θέση έχει αυτό το στοιχείο. Οπότε, αντ'αυτού έβαλα να εμφανίζεται η τιμή του στοιχείου που χαλάει την ταξινόμηση, και όχι την θέση του. Βέβαια, κάτι τέτοιο γνωρίζω ότι δεν έχει ιδιαίτερο νόημα.

4 Ενδεικτικά τρεξίματα

Τα παρακάτω τρεξίματα έγιναν σε Arch Linux, εξ'ού και τα newlines που ανέφερα στα προβλήματα και χαλάνε την εμφάνιση του πρόγραμματος.

```
[christos@archlinux ~/repos/uni/uni-assignments/c-parallel-computing/ex1]$ mpiexec -n 2 ./a.out
Enter N:
7
t[0]:
1
t[1]:
2
t[2]:
3
t[3]:
4
t[4]:
5
t[5]:
6
t[6]:
7
Array is sorted.
Press [ENTER] to continue. . .
```

```
Enter N:
11
t[0]:
1
t[1]:
2
t[2]:
311
t[3]:
223
t[4]:
21
t[5]:
522
t[6]:
1
t[7]:
2
t[8]:
3
t[9]:
51
t[10]:
6
Array is not sorted: first unsorted element: 311
Press [ENTER] to continue. . .
```

```
1. Continue
2. Exit
Your choice:

1
Enter N:
5
t[0]:
3
t[1]:
4
t[2]:
2
t[3]:
6
t[4]:
1
Array is not sorted: first unsorted element: 4
Press [ENTER] to continue. . .
```

```
Enter N:
12
t[0]:
1
t[1]:
2
t[2]:
6
t[3]:
3
t[4]:
4
t[5]:
5
t[6]:
1
t[7]:
2
t[8]:
4
t[9]:
1
t[10]:
2
t[11]:
3
Array is not sorted: first unsorted element: 6
Press [ENTER] to continue. . .
```

Σε αυτό το τρέξιμο έδωσα την λίστα που έφερα ως παράδειγμα πιο πάνω και ανέφερα ότι αν δεν ο κάθε επεξεργαστής δεν ξέρει ποιο είναι το προηγούμενο από αυτόν στοιχείο, δεν μπορούμε να έχουμε σωστό αποτέλεσμα.