Εργαστήριο Παράλληλου Υπολογισμού - Εργασία 2

Χρήστος Μαργιώλης

Ιανουάριος 2020

Περιεχόμενα

1 Τεκμηρίωση

Το πρόγραμμα, προκειμένου να υλοποίησει τα ερωτήματα της άσκησης, ακολουθεί την εξής δομή:

- Δέχεται το διάνυσμα X και το μήκος του στον επεξεργαστή 0.
- Υπολογίζει τα παρακάτω ωστέ να είναι πιο εύκολη και οργανωμένη η υλοποίηση του υπόλοιπου προγράμματος.
  - X_{min}
  - X_{max}
  - m - Μέση τιμή
  - `scounts` και `displs` - Για να χρησιμοποιηθούν από τις `MPI_Scatterv(3)` και `MPI_Gatherv(3)`.
  - Τοπικό n για τον κάθε επεξεργαστή καθώς και το διάνυσμα που του αναλογεί.
- Αφού γίνουν επιτυχώς τα παραπάνω, υλοποιεί τα ερωτήματα.
  - Πόσα στοιχεία είναι μικρότερα ή μεγαλύτερα της μέσης τιμής m.
  - Την διασπορά των στοιχείων του διανύσματος X.
  - Διάνυσμα Δ.
  - Την μεγαλύτερη τιμή του διανύσματος Δ καθώς και την θέση της στο διάνυσμα.
  - Το διάνυσμα προθεμάτων (prefix sums) των στοιχείων του διανύσματος X.
- Εμφανίζει όλα τα ζητούμενα αποτελέσματα στον επεξεργαστή 0.

Αυτό που αξίζει προσοχή είναι οι πίνακες `scounts` και `displs` που ανέφερα παραπάνω. Προκειμένου να μοιράζεται σωστά το διάνυσμα ακόμα και στην περίπτωση που το n δεν είναι ακέραιο πολλαπλάσιο του p, πρέπει να χρησιμοποιηθούνε οι συναρτήσεις `MPI_Scatterv(3)` και `MPI_Gatherv(3)` - αυτές οι συναρτήσεις είναι παρόμοιες με τις `MPI_Scatter(3)` και `MPI_Gather(3)`, με την διαφορά ότι μπορούνε να δεχτούνε μεταβλητό αριθμό στοιχείων προς αποστολή στον κάθε επεξεργαστή. Για παράδειγμα, ας υποθέσουμε ότι είμαστε στην περίπτωση που το n *είναι* πολλαπλάσιο του p και έχουμε το εξής input:

$$p = 4$$
$$n = 8$$
$$X = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

Τότε, μπορούμε να ισομοιράσουμε τα στοιχεία στους $p = 4$ επεξεργαστές ως εξής:

$$p_0 = \{1, 2\}$$
$$p_1 = \{3, 4\}$$
$$p_2 = \{5, 6\}$$
$$p_3 = \{7, 8\}$$

Στην περίπτωση όμως που το n *δεν* είναι ακέραιο πολλαπλάσιο του p, χρειαζόμαστε να μοιράσουμε τα στοιχεία έτσι ώστε όλοι οι επεξεργαστές να έχουνε μέχρι 1 στοιχείο παραπάνω, προκειμένου ο υπολογιστικός φόρτος να είναι όσο το δυνατόν πιο ίσα μοιρασμένος. Οπότε, έχοντας το παρακάτω input ως παράδειγμα:

$$p = 4$$
$$n = 11$$
$$X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Με βάση τα παραπάνω λεγόμενά μου, τα στοιχεία πρέπει να μοιραστούν ως εξής:

$$p_0 = \{1, 2, 3\}$$
$$p_1 = \{4, 5, 6\}$$
$$p_2 = \{7, 8, 9\}$$
$$p_3 = \{10, 11\}$$

Προκειμένου να επιλυθεί αυτό το πρόβλημα, χρησιμοποιούμε, όπως προανέφερα, τις συναρτήσεις `MPI_Scatterv(3)` και `MPI_Gatherv(3)`. Τα νέα ορίσματα που μάς απασχολούνε είναι τα εξής δύο:

- `scounts` - αριθμός στοιχείων που παίρνει ο κάθε επεξεργαστής
- `displs` - offset στο γενικό διάνυσμα

Και τα δύο ορίσματα (πίνακες) έχουνε μήκος p, ωστέ σε κάθε θέση του πίνακα να υπάρχουνε οι κατάλληλες πληροφορίες για το πώς θα μοιραστούν τα στοιχεία στον κάθε επεξεργαστή.

Έπειτα, θέτουμε τις κατάλληλες τιμές για το τοπικό n και διάνυσμα. Στον κώδικα έχω κάνει `localn = scounts[rank]` το οποίο, αν και προφανές, σημαίνει ότι το κάθε τοπικό διάνυσμα έχει μήκος όσο και τα στοιχεία που υπολογίσαμε ότι θα έχει ο επεξεργαστής που θα το δεχτεί. Στην συνέχεια στέλνουμε - επιτέλους - μέσω της `MPI_Scatterv(3)` σε όλους τους επεξεργαστές το διάνυσμα που τούς αναλογεί.

Μετά από τα παραπάνω βήματα, ξεκινάνε οι υπολογισμοί για τα ζητούμενα της άσκησης, οπότε θα εξήγησω στο επόμενος μέρος περιληπτικά τί κάνει η κάθε συνάρτηση που έχω υλοποιήσει με την σειρά που εκτελούνται στο πρόγραμμα.

2 Συναρτήσεις

2.1 `float input(const char *fmt, int i)`

Δίνει την δυνατότητα για formatted input.

2.2 `float find(int flag)`

Βρίσκει το X_{min} και X_{max} ανάλογα με το flag που τής έχει δωθεί. Τα flags που μπορούνε να δωθούν είναι τα εξής

- `FIND_XMIN`
- `FIND_XMAX`

Προκειμένου να υπολογίσει οποιαδήποτε από τις δύο τιμές ακολουθεί τον εξής αλγόριθμο:

- Για κάθε στοιχείο του τοπικού διανύσματος του εκάστοτε επεξεργαστή, βρες το τοπικό μέγιστο ή ελάχιστο.
- Μάζεψε τα αποτέλεσματα από όλους τους επεξεργαστές στον `root`.
- Βρες το ολικό μέγιστο ή ελάχιστο με την ίδια λογική όπως στο πρώτο βήμα.
- Στείλε το αποτέλεσμα σε όλους τους επεξεργαστές.

2.3 `float calcavg(void)`

Υπολογίζει την ολική μέση τιμή. Αρχικά βρίσκει όλα τα τοπικά μέγιστα, τα μαζεύει στον `root` επεξεργαστή, ο οποίος βρίσκει την ολική μέση τιμή και την στέλνει σε όλους τους υπόλοιπους επεξεργαστές.

2.4 `float findavg(float *v, int len)`

Βοηθητική συνάρτηση για υπολογισμό μέσης τιμής. Χρησιμοποιείται από την `calcavg` [2.3].

2.5 `int count(float avg, int flag)`

Υπολογίζει πόσα στοιχεία είναι είτε μεγαλύτερα είτε μικρότερα της ολικής μέσης τιμής m. Το τί από τα δύο θα υπολογίσει εξαρτάται από το flag που θα τής δωθεί. Τα flags που δέχεται είναι τα εξής:

- `COUNT_BELOW_AVG`
- `COUNT_ABOVE_AVG`

Για τους υπολογισμούς ακολουθεί παρόμοια λογική με την `find` [2.2].

2.6 `float calcvar(float avg)`

Υπολογίζει την ολική διασπορά των στοιχείων του διανύσματος X. Ως όρισμα δέχεται την μέση τιμή m που υπολογίζει η `calcavg` [2.3]. Ο τύπος που χρησιμοποιεί για τον υπολογισμό της τοπικής διασποράς είναι:

$$var_{local} = \sum_{i=0}^{n_{local}} (x_i - m)^2$$

Αφού μαζέψει όλα τα τοπικά αποτελέσματα στον επεξεργαστή `root`, τα αθροίζει και τα διαιρεί δια $n-1$, ώστε να ολοκληρωθεί ο τύπος της διασποράς.

$$var = \frac{1}{n-1} \cdot \sum_{i=0}^{n-1} (x_i - m)^2$$

2.7 `float *calcd(float xmin, float xmax)`

Δημιουργεί ένα νέο διάνυσμα Δ, του οποίο το κάθε στοιχείο δ_i είναι ίσο με

$$\delta_i = ((x_i - x_{min})/(x_{max} - x_{min})) \cdot 100$$

Αφού υπολογίσει τον παραπάνω τύπο για κάθε στοιχείο όλων των τοπικών διανυσμάτων, μαζεύει όλα τα διανύσματα στον επεξεργαστή `root` μέσω της `MPI_Gatherv(3)`.

2.8 `Pair findmax(float *d)`

Βρίσκει το ολικό μέγιστο στο διάνυσμα Δ καθώς και την θέση του στο διάνυσμα. Αρχικά, αυτό που επιστρέφει η συνάρτηση είναι ένα `struct Pair`, το οποίο είναι ένα `struct` που δημιούργησα ώστε να αποθηκεύσω τα δύο αποτέλεσματα που θα παράξει η συνάρτηση αυτή (D_{max} και D_{maxloc}).

Ο αλγόριθμος που ακολουθεί η συνάρτηση είναι ο εξής:

- Για κάθε στοιχείο του γενικού διανύσματος Δ, ψάξε το μέγιστο στοιχείο και την θέση του.
- Αφού βρεθεί, με την χρήση της `MPI_Reduce(3)`, βρες την θέση το ολικό μέγιστο καθώς και την θέση του και αποθήκευσέ τα στο `out` στον επεξεργαστή `root`.

2.9 `float *calcpfxsums(void)`

Υπολογίζει το διάνυσμα προθεμάτων (prefix sums) των στοιχείων του διανύσματος X. Προκειμένου να γίνει αυτό χρησιμοποείται η συνάρτηση `MPI_Scan(3)`, η οποία κάνει όλους τους απαραίτητους υπολογισμούς. Εμείς το μόνο που έχουμε να κάνουμε είναι να αποθηκεύσουμε τα αποτελέσματα στον πίνακα `pfxsums`. Σημαντικό να σημειωθεί ότι αυτή η συνάρτηση εκτελείται επιτυχώς *μόνο* στην περίπτωση που $n = p$.

2.10 `void printv(const char *str, const float *v)`

Βοηθτική συνάρτηση για να τυπώνει διανύσματα με πιο όμορφο τρόπο.

2.11 `void *emalloc(size_t nb)`

`malloc(3)` με error checks.

3 Κώδικας

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

/* Flags */
#define FIND_XMIN       1 << 0
#define FIND_XMAX       1 << 1
#define COUNT_BELOW_AVG 1 << 2
#define COUNT_ABOVE_AVG 1 << 3

typedef struct {
        float val;      /* Max value in array */
        int i;          /* Index of max */
} Pair;

/* Function declarations */
static float     input(const char *, int);
static float     find(int);
static float     findavg(float *, int);
static float     calcavg(void);
static int       count(float, int);
static float     calcvar(float);
static float    *calcd(float, float);
static Pair      findmax(float *);
static float    *calcpfxsums(void);
static void      printv(const char *, const float *);
static void     *emalloc(size_t);

/* Global variables */
static int rank, nproc, root = 0;
static int *scounts, *displs;
static float *vec, *localvec;
static int n, localn;

/* Function implementations */
/* Formatted input */
static float
input(const char *fmt, int i)
{
        char buf[48];
        float n;

        sprintf(buf, fmt, i);
        printf("%s", buf);
```

```
        scanf("%f", &n);
        getchar();

        return n;
}

/* Find `xmin' and `xmax' depending on the `flag' argument. */
static float
find(int flag)
{
        float localres = *localvec;
        float finalres = localres;
        float *res;
        int i;

        res = emalloc(nproc * sizeof(float));
        /*
         * Loop through each local vector and assign the local
         * result depending on which of the two flags is set
         */
        for (i = 0; i < localn; i++)
                if ((flag & FIND_XMIN && localvec[i] < localres)
                ||  (flag & FIND_XMAX && localvec[i] > localres))
                        localres = localvec[i];
        /* Send local results to `root' */
        MPI_Gather(&localres, 1, MPI_FLOAT, res, 1, MPI_FLOAT, root, MPI_COMM_WORLD);

        if (rank == root)
                /* Same process as above */
                for (i = 0; i < nproc; i++)
                        if ((flag & FIND_XMIN && res[i] < finalres)
                        ||  (flag & FIND_XMAX && res[i] > finalres))
                                finalres = res[i];

        /* Everyone has to know the final result */
        MPI_Bcast(&finalres, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
        free(res);

        return finalres;
}

/*
 * Small utility function for `calcavg' to avoid code duplication.
 * Calcuates the average for a given vector
 */
static float
```

```
findavg(float *v, int len)
{
        float sum = 0.0f;
        int i = 0;

        for (; i < len; i++)
                sum += v[i];
        return (sum / (float)len);
}

/* Calculate the global average */
static float
calcavg(void)
{
        float *avgs, localavg, finalavg;

        avgs = emalloc(nproc * sizeof(float));
        localavg = findavg(localvec, localn);
        MPI_Gather(&localavg, 1, MPI_FLOAT, avgs, 1, MPI_FLOAT, root, MPI_COMM_WORLD);

        if (rank == root)
                finalavg = findavg(avgs, nproc);
        MPI_Bcast(&finalavg, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
        free(avgs);

        return finalavg;
}

/*
 * Count how many elements are below or above average based on the
 * `flag` argument. Similar logic as with `find` above.
 */
static int
count(float avg, int flag)
{
        int *res, localres = 0, finalres = 0, i;

        res = emalloc(nproc * sizeof(int));
        for (i = 0; i < localn; i++)
                if ((flag & COUNT_BELOW_AVG && localvec[i] < avg)
                ||  (flag & COUNT_ABOVE_AVG && localvec[i] > avg))
                        localres++;
        MPI_Gather(&localres, 1, MPI_INT, res, 1, MPI_INT, root, MPI_COMM_WORLD);

        if (rank == root)
                for (i = 0; i < nproc; i++)
```

```
                        finalres += res[i];
        MPI_Bcast(&finalres, 1, MPI_INT, root, MPI_COMM_WORLD);
        free(res);

        return finalres;
}

/* Calculate the global variance */
static float
calcvar(float avg)
{
        float *vars, localvar = 0.0f, finalvar = 0.0f;
        int i;

        for (i = 0; i < localn; i++)
                localvar += (localvec[i] - avg) * (localvec[i] - avg);

        vars = emalloc(nproc * sizeof(float));
        MPI_Gather(&localvar, 1, MPI_FLOAT, vars, 1, MPI_FLOAT, root, MPI_COMM_WORLD);

        if (rank == root) {
                for (i = 0; i < nproc; i++)
                        finalvar += vars[i];
                finalvar /= (float)n - 1;
        }
        MPI_Bcast(&finalvar, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
        free(vars);

        return finalvar;
}

/* Generate D. A vector where each element is
 * ((xi - xmin) / (xmax - xmin)) * 100.
*/
static float *
calcd(float xmin, float xmax)
{
        float *locald, *finald;
        int i;

        locald = emalloc(localn * sizeof(float));
        finald = emalloc(n * sizeof(float));

        for (i = 0; i < localn; i++)
                locald[i] = ((localvec[i] - xmin) / (xmax - xmin)) * 100;
```

```
        MPI_Gatherv(locald, localn, MPI_FLOAT, finald, scounts, displs,
                MPI_FLOAT, root, MPI_COMM_WORLD);

        free(locald);

        return finald;
}

/* Find global max and MAXLOC */
static Pair
findmax(float *d)
{
        Pair in, out;
        int i = 1;

        in.val = *d;
        in.i = 0;
        for (; i < n; i++) {
                if (in.val < d[i]) {
                        in.val = d[i];
                        in.i = i;
                }
        }
        in.i += rank * localn;
        MPI_Reduce(&in, &out, 1, MPI_FLOAT_INT, MPI_MAXLOC, root, MPI_COMM_WORLD);

        return out;
}

/* Calucate the prefix sums of `vec`. Only world when
 * n == nproc
 */
static float *
calcpfxsums(void)
{
        float *pfxsums;
        float sum = 0.0f;

        pfxsums = emalloc(n * sizeof(float));

        /* Scan each local vector and assign the result to `sum`. */
        MPI_Scan(localvec, &sum, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
        /* Be in sync. */
        MPI_Barrier(MPI_COMM_WORLD);
        /* Get results in `pfxsums` */
        MPI_Gather(&sum, 1, MPI_FLOAT, pfxsums, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
```

```
        return pfxsums;
}

/* Utility function to print a vector in a prettier way. */
static void
printv(const char *str, const float *v)
{
        int i = 0;

        printf("%s␣[", str);
        for (; i < n; i++)
                printf("%.4f%s", v[i], i != n-1 ? ",␣" : "");
        printf("]\n");
}

/* Error checking `malloc(3)`. */
static void *
emalloc(size_t nb)
{
        void *p;

        if ((p = malloc(nb)) == NULL) {
                fputs("cannot␣allocate␣memory", stderr);
                exit(EXIT_FAILURE);
        }
        return p;
}

int
main(int argc, char *argv[])
{
        Pair dmax;
        float avg, var, xmin, xmax;
        float *d, *pfxsums;
        int belowavg, aboveavg;
        int i, rc;

        if ((rc = MPI_Init(&argc, &argv)) != 0) {
                fprintf(stderr, "%s:␣cannot␣initialize␣MPI.\n", argv[0]);
                MPI_Abort(MPI_COMM_WORLD, rc);
        }
        MPI_Comm_size(MPI_COMM_WORLD, &nproc);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        /* Read global vector. */
```

```
if (rank == root) {
        n = input("N:␣", 0);
        vec = emalloc(n * sizeof(float));
        for (i = 0; i < n; i++)
                vec[i] = input("vec[%d]:␣", i);
}

/* Send ‘n‘ to everyone. */
MPI_Bcast(&n, 1, MPI_INT, root, MPI_COMM_WORLD);

/* Will be used for ‘MPI_Scatterv(3)‘ and ‘MPI_Gatherv(3)‘ later on. */
scounts = emalloc(nproc * sizeof(int));
displs = emalloc(nproc * sizeof(int));
for (i = 0; i < nproc; i++) {
        /* make it work even if ‘n‘ is not a multiple of ‘nproc‘. */
        scounts[i] = (i != nproc - 1) ? n / nproc : n / nproc + n % nproc;
        /* take the last ‘scounts‘ so that we don't offset +1 each time. */
        displs[i] = i * scounts[i != 0 ? i-1 : i];
}

/*
 * Each local ‘n‘ is the same as the ‘scounts‘ of each process, so we
 * assign it to ‘localn‘ for readablity.
 */
localn = scounts[rank];
localvec = emalloc(localn * sizeof(float));

/* Scatter the array to each process. */
MPI_Scatterv(vec, scounts, displs, MPI_FLOAT, localvec, localn,
        MPI_FLOAT, root, MPI_COMM_WORLD);

/* Part 0.1 - Calculate global minimum and maximum. */
xmin = find(FIND_XMIN);
xmax = find(FIND_XMAX);

/* Part 0.2 - Calculate average. */
avg = calcavg();

/* Part 1 - Find how many elements are above or below average. */
belowavg = count(avg, COUNT_BELOW_AVG);
aboveavg = count(avg, COUNT_ABOVE_AVG);

/* Part 2 - Calculate variance. */
var = calcvar(avg);

/* Part 3 - Generate D. */
```

```
        d = calcd(xmin, xmax);

        /* Part 4 - Find dmax and dmaxloc. */
        dmax = findmax(d);

        /* Part 5 - Prefix sums of 'vec'. */
        pfxsums = calcpfxsums();

        /* Print all results */
        if (rank == root) {
                printf("\n");
                printv("X:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", vec);
                printf("Average:␣␣␣␣␣␣␣␣␣%.4f\n", avg);
                printf("Xmin:␣␣␣␣␣␣␣␣␣␣␣␣␣%.4f\n", xmin);
                printf("Xmax:␣␣␣␣␣␣␣␣␣␣␣␣␣%.4f\n", xmax);
                printf("Below␣Average:␣␣␣␣%d\n", belowavg);
                printf("Above␣Average:␣␣␣␣%d\n", aboveavg);
                printf("Variance:␣␣␣␣␣␣␣␣%.4f\n", var);
                printv("D:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", d);
                printf("Dmax:␣␣␣␣␣␣␣␣␣␣␣␣%.4f\n", dmax.val);
                printf("Dmaxloc:␣␣␣␣␣␣␣␣␣%d\n", dmax.i);
                printv("Prefix␣Sums:␣␣␣␣", pfxsums);
        }

        free(scounts);
        free(displs);
        free(vec);
        free(localvec);
        free(d);
        free(pfxsums);

        MPI_Finalize();

        return 0;
}
```

4 Προβλήματα

Δεν κατάφερα να βρω πώς να υλοποιηθεί η εύρεση του διανύσματος προθεμάτων στην περίπτωση που το $n \neq p$.

5 Ενδεικτικά τρεξίματα

Input: $p = 4$, $n = 7$, $X = \{1, 2, 3, 4, -1, -2, 6\}$

```
christos@pleb$ mpicc ex2.c && mpiexec -n 4 ./a.out < data
N: vec[0]: vec[1]: vec[2]: vec[3]: vec[4]: vec[5]: vec[6]:
X:              [1.0000, 2.0000, 3.0000, 4.0000, -1.0000, -2.0000, 6.0000]
Average:        1.9375
Xmin:           -2.0000
Xmax:           6.0000
Below Average:  3
Above Average:  4
Variance:       7.8171
D:              [37.5000, 50.0000, 62.5000, 75.0000, 12.5000, 0.0000, 100.0000]
Dmax:           100.0000
Dmaxloc:        6
Prefix Sums:    [1.0000, 3.0000, 6.0000, 10.0000, 0.0000, 0.0000, 0.0000]
```

Input: $p = 4$, $n = 11$, $X = \{1, 2, -5, 21, 13, 6, -10, 8, 9, 4, -6\}$

```
christos@pleb$ mpicc ex2.c && mpiexec -n 4 ./a.out < data
N: vec[0]: vec[1]: vec[2]: vec[3]: vec[4]: vec[5]: vec[6]: vec[7]: vec[8]: vec[9]: vec[10]:
X:              [1.0000, 2.0000, -5.0000, 21.0000, 13.0000, 6.0000, -10.0000, 8.0000, 9.0000, 4.0000, -6.0000]
Average:        5.0000
Xmin:           -10.0000
Xmax:           21.0000
Below Average:  6
Above Average:  5
Variance:       81.8000
D:              [35.4839, 38.7097, 16.1290, 100.0000, 74.1936, 51.6129, 0.0000, 58.0645, 61.2903, 45.1613, 12.9032]
Dmax:           100.0000
Dmaxloc:        3
Prefix Sums:    [1.0000, -4.0000, 9.0000, -1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000]
```

Input: $p = 8$, $n = 8$, $X = \{2, 3, 4, 5, 6, -1, -2\}$

Εφόσον τώρα ισχύει $n = p$, η συνάρτηση για την έυρεση των prefix sums ϑα λειτουργήσει σωστά.

```
christos@pleb$ mpicc ex2.c && mpiexec -n 8 ./a.out < data
N: vec[0]: vec[1]: vec[2]: vec[3]: vec[4]: vec[5]: vec[6]: vec[7]:
X:              [2.0000, 3.0000, 4.0000, 5.0000, 6.0000, -1.0000, -2.0000, -2.0000]
Average:        1.8750
Xmin:           -2.0000
Xmax:           6.0000
Below Average:  3
Above Average:  5
Variance:       10.1250
D:              [50.0000, 62.5000, 75.0000, 87.5000, 100.0000, 12.5000, 0.0000, 0.0000]
Dmax:           100.0000
Dmaxloc:        4
Prefix Sums:    [2.0000, 5.0000, 9.0000, 14.0000, 20.0000, 19.0000, 17.0000, 15.0000]
```