Παντελεήμων Πρώιος

ice 18390023

5ο ‘Εξάμηνο

Μηχανικών Πληροφορικής και Υπολογιστών

ice18390023@uniwa.gr

ΕΙΣΑΓΩΓΗ ΣΤΟΝ ΠΑΡΑΛΛΗΛΟ ΥΠΟΛΟΓΙΣΜΟ

ΕΡΓΑΣΙΑ 2

ΥΠΕΥΘΗΝΟΙ ΚΑΘΗΓΗΤΕΣ

ΜΑΜΑΛΗΣ ΒΑΣΙΛΗΣ

ΙΟΡΔΑΝΑΚΗΣ ΜΙΧΑΛΗΣ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
UNIVERSITY OF WEST ATTICA

PREFIX SUM ΓΙΑ N > P

Η υλοποίηση του prefix sum για N > P μπορεί να γίνει με τουλάχιστον 2 τρόπους.

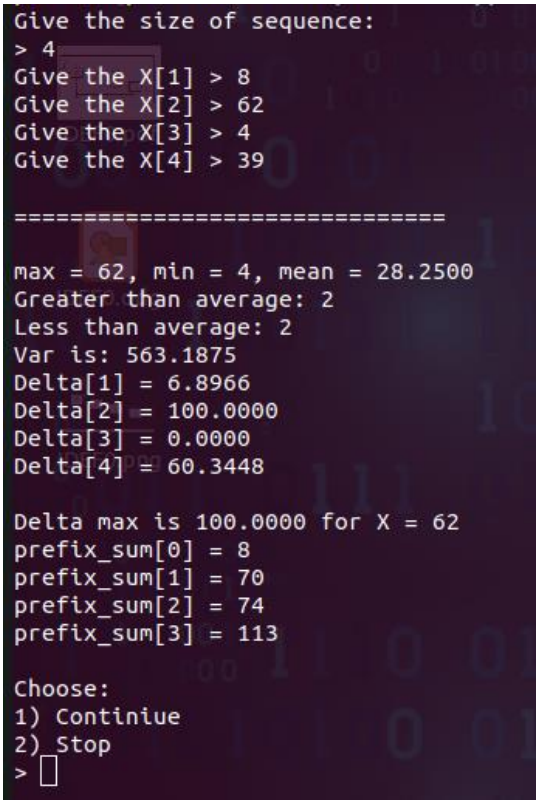
Α' ΤΡΟΠΟΣ

Αφού, ο κάθε επεξεργαστής υπολογίση το τοπικό prefix_sum, να τοποθετηθεί η συνάρτηση MPI_Barrier. Έπιτα ο κάθε επεξεργαστής να υπολογίση την τιμή $\lceil \log_2 P \rceil$, όπου P ο αριθμός των επεξεργαστών που πρέπει να γίνει η επικοινωνία. Έστω οτι το αποτέλεσμα του υπολογισμού της τιμής βρίσκεται στην μεταβλητή times. Στην συνέχεια θα πρέπει να υπάρχει ένας βρόγχος επανάληψης, όπου θα επαναληφθή για times φορές ξεκινώντας τον μετρητή απο το 0, έστω μετρητής είναι ο i. Μέσα στον βρόγχο, θα γίνεται ο υπολογισμός του 2^i και έστω οτι αποθηκεύευταί στην μεταβλητή offset. Έπιτα, θα υπάρχουν τρείς συνθήκες if. Η πρώτη συνθήκη ελέγχει αν το rank-offset είναι αρνιτικος αριθμός, αν δεν είναι τότε θα λαμβάνει μια τιμή μέσω της MPI_Irecv με το source να είναι ίσο με το rank-offset και έστω ότι αποθηκεύει την τιμή στο rcv_buf επίσης σε μια μεταβλητή έστω με το όνομα flag, θα εκχωρεί true η οποία θα πρέπει να έχει αρχικοποιηθεί με false πριν τον βρόγχο. Έπιτα ακολουθεί η επόμενη συνθήκη, όπου θα ελέγχει αν το rank+offset είναι μικρότερο απο το πλήθος των επεξεργαστών P, άν είναι μικρότερο, τότε θα στέλνει μέσο της MPI_Isend με destination το rank+offset το τελευταίο στοιχείο του τοπικού πίνακα prefix_sum. Μετά απο τις δύο συνθήκες, θα υπάρχει η τελευταία συνθήκη θα ελέγχει αν η μεταβλητή flag είναι true, άν είανι τότε θα προσθέτει σε κάθε στοιχείο του τοπικού πίνακα prefix_sum το περιεχόμενο που παρέλαβε και βρίσκεται στην μεταβλητή rcv_buf και τέλος θα αρχικοποιεί την μεταβλητή flag με false. Μετά την τελευταία συνθήκη θα χρησιμοποιηθεί η συνάρτηση MPI_Barrier.

Note: ενδεχομένος να είναι ποίο efficient, πρώτα η συνθήκη που περιέχει την MPI_Isend και μετά της MPI_Ircv. Επίσης, καλύτερα η MPI_Ircv να αντικατασταθεί με MPI_Rcv και να αφαιρεθεί η συνάρτηση MPI_Barrier, εκτός και επειδή κάποιος φορές η MPI_Rcv λειτουργή σαν την MPI_Ircv είναι καλύτερο να μείνει.

Β' ΤΡΟΠΟΣ

Αφού, κάθε επεξεργαστής υπολογίση το τοπικό prefix_sum, να ενημερώνετε η ROOT μέσο της MPI_Gather, και κάθε επεξεργαστης να της στέλνει το τελευταίο στοιχείο του τοπικού πίνακα prefix_sum. Έπιτα, η ROOT βάση αυτου του πίνακα, να υπολογίζει αυτό το prefix_sum, να μετατοπίζει όλα τα στοιχεία κατά ένα δεξιότερο και στο πρώτο στοιχείο να εκχωρεί την τιμή 0. Στην συνέχεια, να γίνει αποστολή αυτού του πίνακα σε όλους τους επεξεργαστές που συμμετέχουν και ο κάθε επεξεργαστής να προσθέσει αυτόν τον αριθμό σε όλα τα στοιχεία του τοπικού πίνακα prefix_sum.



Σχήμα 1 Για 4 επεξεργαστές

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

#define ROOT 0

int menu();
int read_size(int );
int *read_sequence(int );
int *create_malloc(int );
int minimum( int *, int);
int maximum( int *, int);
int summary( int *, int);
void find_greater_less( int *, int , double , int *);
double var_calc( int *, int, double, int);
double delta_calc( int , int , int );
double *create_malloc_double(int );
int index_max( double *, int );


int main(int argc, char** argv){

    int rank;
    int menu_response;
    int N, p;
    int *X;
    int *send_rcv_size;
    int tmp;
    int i;
    int extra;
    int loc_sizeX;
    int *displs;
    int offset;
    int *loc_X;
    int loc_min, loc_max, loc_sum;
    double loc_mean, mean;
    int min, max;
    int *temp;
    int loc_gl[2], total_gl[2];
    double loc_var, total_var;
    double *loc_delta, *delta;
    int result;
    int *prefix_sum;

    // ========================= MPI Initalized ========================

    // Η MPI_Init, επιστρέφει MPI_SUCCESS, στην επιτυχεία.
    // Στην μεταβλητή tmp, εκχωρείται η επιστρεφόμενη τιμή της MPI_Init και
    // η συνθήκη, ελέγχει αν η tmp δεν είναι MPI_SUCCESS, έτσι ώστε να πράξει
    // καταλλήλος, αλλιώς να συνεχίση κανονικά.

    tmp = MPI_Init( &argc, &argv);

    if (tmp != MPI_SUCCESS){
        perror("MPI initalization");
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }

    MPI_Status status;
```

```
    // κάθε processor μαθαίνει τον αριθμό του, για τον Communicator
MPI_COMM_WORLD
    // και εκχωρείται στην rank.
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);      // Πόσοι υπάρχουν, υπο τον
MPI_COMM_WORLD
                                            // θα είναι ο ίδιος αριθμός για όλους
του MPI_COMM_WORLD




    // Επανάληψη, για πολλαπλές φορές ελέγχων.
    while(1){




        // Η συνθήκη θα είναι αληθής μόνο για τον processor που έχει rank == ROOT
== 0.
        if ( rank == ROOT){

            // Καλεί την συνάρτηση και επιστρέφει τον αριθμό των ακεραίων που θα
διαβάσει.
            N = read_size(p);

            // Επιστρέφει την διεύθηνση των ακεραίων που διαβάστηκαν.
            X = read_sequence(N);


            // Γίνεται allocation p ακεραίων στην μεταβλητή.
            send_rcv_size = create_malloc(p);

            // Γίνεται υπολογισμός, πόσοι επεξεργαστές θα έχουν +1 στοιχείο.
            extra = N % p;

            // Στις δύο επόμενες for, εκχωρείτε στην send_rcv_size το μέγεθος τον
στοιχείων
            // που θα παραλάβουν ή σταλθουν για το διάνυσμα X.
            // Στην πρώτη for εκχωρούντε όσα θα πάρουν συν ένα παραπάνω, ενώ στην
αλλή,
            // είναι απλά N/p.
            for ( i = 0; i < extra; send_rcv_size[i++] = N/p+1 );

            for (; i < p; send_rcv_size[i++] = N/p );


            // Γίνεται allocation p ακεραίων στην μεταβλητή
            displs = create_malloc(p);


            // Γίνεται υπολογισμός της displs. Πρώτα εκχωρείται στην displs[i] το
offset απο
            // το όποίο θα δώσει τα στοιχεία στον i επεξεργαστή και μετά στο
offset, εκχωρείτε
            // το offset που υπήρχε + τον αριθμό των δεδομένων που θα
στελνόντουσαν στον i
```

```
            // επεξεργαστή, τα οποία περιέχονται στο send_rcv_size[i].
            for ( offset = 0, i = 0; i < p; offset = (displs[i] = offset) +
send_rcv_size[i++]);


            printf("\n==============================\n\n");
        }


        // Γίνεται διαμοιρασμός σε κάθε επεξεργαστή του μεγέθους των δεδομένουν
που θα τους
        // σταλθούν. Θα μπορούσε αυτή η ενημέρωση να είχε αποφεχθεί, επειδή θα
τους ενημερώσουμε
        // με την ακόλουθη MPI_Bcast, με το μέγεθος του N, οπότε θα μπορούσαν να
το υπολογίσουν.
        MPI_Scatter( send_rcv_size, 1, MPI_INT, &loc_sizeX, 1, MPI_INT, ROOT,
MPI_COMM_WORLD);


        // Γίνεται ενημέρωση του μεγέθους του διανύσματος X.
        MPI_Bcast( &N, 1, MPI_INT, ROOT, MPI_COMM_WORLD);



        // Κάθε επεξεργαστής δεσμεύει μνήμη μεγέθους loc_sizeX στην loc_X.
        loc_X = create_malloc( loc_sizeX);


        // Ο κάθε επεξεργαστής λαμβάνη το μέρος του διανύσματος που του
αντιστοιχεί και εκχωρείται
        // στην loc_X.
        MPI_Scatterv( X, send_rcv_size, displs, MPI_INT, loc_X, loc_sizeX,
MPI_INT, ROOT, MPI_COMM_WORLD);


        // Η ROOT αποδεσμεύη την δεσμευμένη μνήμη της διεύθησης X.
        if (rank == ROOT){
            free(X);
        }


        // Στις μεταβλητές loc_min, loc_max και loc_sum εκχωρούντε η μικρότερη
τιμή, η μεγαλύτερή τιμή
        // και το άθροισμα αντίστοιχα, του πίνακα loc_X μεγέθους loc_sizeX κάθε
τοπικού επεξεργαστή.
        loc_min = minimum( loc_X, loc_sizeX);
        loc_max = maximum( loc_X, loc_sizeX);
        loc_sum = summary( loc_X, loc_sizeX);

        // Στην μεταβλητή loc_mean δεν είναι η τοπική μέση τιμή, αλλά ένα κομμάτι
της συνολικής μέσης τιμής.
        loc_mean = (double)loc_sum/(double)N;



        // Με τις ακόλουθες MPI_Reduce, βρίσκονται η τιμές max, min και mean
αντίστοιχα.
        MPI_Reduce( &loc_max, &max, 1, MPI_INT, MPI_MAX, ROOT, MPI_COMM_WORLD);

        MPI_Reduce( &loc_min, &min, 1, MPI_INT, MPI_MIN, ROOT, MPI_COMM_WORLD);
```

```
        MPI_Reduce( &loc_mean, &mean, 1, MPI_DOUBLE, MPI_SUM, ROOT,
MPI_COMM_WORLD);



        // Η ROOT εκτυπώνει στην οθόνη τα αποτελέσματα max, min και mean
αντίστοιχα.
        if( rank == ROOT){
            printf("max = %d, min = %d, mean = %.4lf\n", max, min, mean);
        }


        // Γίνεται ενημέρωση σε όλους τους επεξεργαστές για το min, max και mean
αντίστοιχα.
        MPI_Bcast( &min, 1, MPI_INT, ROOT, MPI_COMM_WORLD);

        MPI_Bcast( &max, 1, MPI_INT, ROOT, MPI_COMM_WORLD);

        MPI_Bcast( &mean, 1, MPI_DOUBLE, ROOT, MPI_COMM_WORLD);



        // Η συνάρτηση find_greater_less, επιστρέφει στον πίνακα loc_gl[2], το
πλήθος
        // των στοιχείων που είναι μεγαλύτερα του mean και το πλήθος των
στοιχείων που είναι
        // μικρότερα απο το mean στην loc_gl[0] και loc_gl[1] αντίστοιχα.
        find_greater_less( loc_X, loc_sizeX, mean, loc_gl);

        // Με την MPI_Reduce γίνεται ενημέρωση των αποτελεσμάτων και το άθροισμα
αυτόν,
        // εκχωρείτε στην total_gl.
        MPI_Reduce( loc_gl, total_gl, 2, MPI_INT, MPI_SUM, ROOT, MPI_COMM_WORLD);


        // Η ROOT εκτυπώνει στην οθόνη τα αποτελέσματα με κατάλληλο μήνυμα.
        if( rank == ROOT){
            printf("Greater than average: %d\nLess than average: %d\n",
total_gl[0], total_gl[1]);
        }



        // Γίνεται τοπικός υπολογισμός της διασποράς και εκχωρείτε στην loc_var.
        loc_var = var_calc( loc_X, loc_sizeX, mean, N);


        // Με την MPI_Reduce, γίνεται ενημέρωση των τοπικών υπολογισμών και
παράλληλα και
        // ο συνολικός υπολογισμός της διασποράς.
        MPI_Reduce( &loc_var, &total_var, 1, MPI_DOUBLE, MPI_SUM, ROOT,
MPI_COMM_WORLD);


        // Ο ROOT εκτυπώνει μήνυμα στην οθόνη με την διασπορά και κατάλληλο
μήνυμα.
        if( rank == ROOT){
            printf("Var is: %.4lf\n", total_var);
        }
```

```
        // Κάθε επεξεργαστής δεσμεύει μνήμη για double μεγέθους loc_sizeX
        loc_delta = create_malloc_double(loc_sizeX);


        // Ο ROOT δεσμεύει μνήμη για δεδομένα τύπου double, μεγέθους N, διότι θα
πρέπει
        // να τα παραλάβει για να τα εκτυπώσει.
        if ( rank == ROOT){
            delta = create_malloc_double(N);
        }


        // Κάθε επεξεργαστής υπολογίζει την τοπική loc_delta
        for( i = 0; i < loc_sizeX; loc_delta[i++] = delta_calc( loc_X[i], min,
max));


        // Ο ROOT μαζεύει όλα τα loc_delta
        MPI_Gatherv( loc_delta, loc_sizeX, MPI_DOUBLE, delta, send_rcv_size,
displs, MPI_DOUBLE, ROOT, MPI_COMM_WORLD);

        // Ο ROOT εκτυπώνει όλα τα delta στοιχεία με κατάλληλο μήνυμα και έπιτα
απελευθερώνει τον δεσμευμένο χώρο.
        if( rank == ROOT){
            for( i = 0; i < N; i++){
                printf("Delta[%d] = %.4lf\n", i+1, delta[i]);
            }
            free(delta);
        }


        // Το struct θα μας επιτρέψει να περάσουμε 2 τιμές στην MPI_Reduce
        struct{
            double delta;
            int xi;
        } in, out;


        // Κάθε επεξεργαστής βρίσκεί τον δείκτη του max loc_delta, μέσα απο την
συνάρτηση index_max και
        // εκχωρείτε στην μεταβλητή tmp. Έπιτα, στην in τοπόθετούνται τα στοιχεία
του loc_delta και loc_X,
        // με δείκτη tmp.
        tmp = index_max( loc_delta, loc_sizeX);

        in.delta = loc_delta[tmp];
        in.xi = loc_X[tmp];

        // Γίνεται χρήση του op MPI_MAXLOC και η χρήση MPI_Datatype
MPI_DOUBLE_INT
        MPI_Reduce( &in, &out, 1, MPI_DOUBLE_INT, MPI_MAXLOC, ROOT,
MPI_COMM_WORLD);

        // Ο ROOT εκτυπώνει μήνυμα στην οθόνη με την μεγαλύτερη τιμή delta και
την τιμή του
        // διανύσματος X απο την οποία προήλθε.
        if( rank == ROOT){
            printf("\nDelta max is %.4lf for X = %d\n", out.delta, out.xi);
        }
```

```
        // Με την MPI_Scan, κάθε επεξεργαστής στέλνει το πρώτο στοιχείο του loc_X
δίοτι είναι υλοποίηση για N=p
        // και το result περιέχη το άθροισμα όλων των προηγούμενων επεξεργαστών
συν τον δικό του.
        MPI_Scan( loc_X, &result, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);



        // Ο ROOT δεσμύει μνήμη μεγέθους p για να παραλάβει τα αποτελέσματα όλων
των επεξεργαστών.
        if( rank == ROOT){
            prefix_sum = create_malloc(p);
        }



        // Ο ROOT παραλαμβάνει όλα τα αποτελέσματα.
        MPI_Gather( &result, 1, MPI_INT, prefix_sum, 1, MPI_INT, ROOT,
MPI_COMM_WORLD);


        // Ο ROOT εκτυπώνει στην οθόνη τα αποτελέσματα με κατάλληλο μήνυμα.
        if( rank == ROOT){

            for( i = 0; i < p; printf("prefix_sum[%d] = %d\n", i++,
prefix_sum[i]));

        }


        // Όλοι οι επεξεργαστές αποδεσμεύουν τα loc_X και loc_delta.
        free(loc_X);
        free(loc_delta);


        // Ο ROOT αποδευσμεύει την μνήμη που έχει δεσμεύσει και εν συνεχεία καλή
την
        // συνάρτηση menu και η επιστρεφόμενη τιμή εκχωρείτε στην menu_response.
        if (rank == ROOT){

            free(send_rcv_size);
            free(displs);
            free(prefix_sum);

            // Στην menu_response, εκχωρείται η επιστρεφόμενη τιμή, της
συνάρτησης menu().
            // Η πιστρεφόμενη τιμή, είναι ο αριθμός που δώθηκε απο το stdin, 1
για συνέχεια
            // και 2 για έξοδο. Σε περίπτωση διαφορετικής τιμής του 1 ή 2, πάλι
θα συνεχήσει.
            menu_response = menu();
        }

        // Γίνεται ενημέρωση σε όλους (αν υπάρχουν), με την τιμή menu_response
που δώθηκε.
        MPI_Bcast( &menu_response, 1, MPI_INT, ROOT, MPI_COMM_WORLD);

        // Αν η τιμή menu_response είναι, τότε σπάει ο ατέρμονος βρόγχος.
        if (menu_response == 2) break;
```

```
    } /* while */

    // Τέλος της παραλληλίας και της χρήσης των συναρτήσεων MPI.
    MPI_Finalize();

    return 0;
}


// Επιστρέφει την ελάχιστη τιμή του πίνακα arr, μεγέθους size.
int minimum( int *arr, int size){

    int min = arr[0];
    int i = 1;

    for (; i < size; ++i){
        if (arr[i] < min) min = arr[i];
    }

    return min;
}


// Επιστρέφει την μέγιστη τιμή του πίνακα arr, μεγέθους size.
int maximum( int *arr, int size){

    int max = arr[0];
    int i = 1;

    for (; i < size; ++i){
        if (arr[i] > max) max = arr[i];
    }

    return max;
}


// Επιστρέφει το άθροισμα του πίνακα arr, μεγέθους size.
int summary( int *arr, int size){

    int sum = 0;
    int i = 0;

    for (; i < size; sum += arr[i++] );

    return sum;
}


// Επιστρέφει τον δείκτη του μέγιστου στοιχείου του πίνακα arr, μεγέθους size.
int index_max( double *arr, int size){

    int max_index = 0;
    int i = 1;

    for (; i < size; ++i){
        if (arr[i] > arr[max_index]) max_index = i;
    }

    return max_index;
```

```
}


// Εκχωρεί στον πίνακα return_arr[0], το πλήθος του πίνακα arr μεγαλύτερου του
avg
// και στον πίνακα return_arr[1], το πλήθος του πίνακα arr μικρότερου του avg.
void find_greater_less( int *arr, int size, double avg, int *return_arr){

    int i = 0;

    return_arr[0] = return_arr[1] = 0;

    for(; i < size; ++i){

        if(arr[i] > avg)
            return_arr[0]++;
        else if(arr[i] < avg)
            return_arr[1]++;
    }

}


// Γίνεται υπολογισμός της διασποράς των στοιχείων του πίνακα arr με μέγεθος
size,
// με μ.ο. avg και συνολικό πλήθος στοιχείων N.
double var_calc( int *arr, int size, double avg, int N){

    int i = 0;
    double sum = 0.0;

    for (; i < size; ++i){
        sum += (arr[i]-avg) * (arr[i]-avg);
    }

    return sum/N;
}



// Επιστρέφει μια τιμή delta, για xi με min και max, βάση του δοθέντος
υπολογισμού.
double delta_calc( int xi, int min, int max){

    return (double)(xi-min)/(double)(max-min)*100;

}



// Η συνάρτηση read_size, έχει ως παράμετρο το μέγεθος των processor. Εκτυπώνει
στο stdout
// κατάλληλο μήνυμα και περιμένει για έναν ακέραιο αριθμό. Ο αριθμός πρέπει να
είναι
// μεγαλύτερος ή ίσος με τους επεξεργαστές, αλλιώς θα εκτυπωθεί κατάλληλο μήνυμα
και θα
// ξανά ζητηθεί αριθμός, μέχρις ότου να είναι μεγαλύτερος ή ισος απο τον αριθμό
των
// επεξεργαστών. Τέλος, θα επιστραφεί αυτός ο αριθμός.
int read_size(int p){
    int N;
```

```c
    do{
        printf("Give the size of sequence:\n> ");
        scanf("%d", &N);
        if ( N < p){
            printf("Size MUST be greater or equal to procecces\n");
        }
    }while(N < p);

    return N;
}


// Η συνάρτηση read_sequence, δέχεται ως παράμετρο, το μέγεθος των στοιχείον προς
ανάγνωση
// από το stdin. Επιστρέφει την διεύθηνση.
int *read_sequence(int N){

    int i, tmp;

    // Εκχωρεί στην ptr την επιστρεφόμενη διεύθηνση απο την create_malloc
    int *ptr = create_malloc(N);

    // Κάνει N επαναλήψης για την ανάγνωση όλων των δεδομένων απο το stdin.
    for ( i = 0; i < N; i++){
        printf("Give the X[%d] > ", i+1);
        scanf("%d", &tmp);
        ptr[i] = tmp;
    }
    return ptr;
}


// Η συνάρτηση create_malloc, δέχεται ως παράμετρο το πλήθος των ακεραίων που θα
γίνουν
// allocate, κάνει έλεγχο και επιστρέφει την διεύθυνση.
int *create_malloc(int len){

    int *addr = NULL;

    addr = (int *) malloc( len * sizeof(int) );

    if (addr == NULL){
        perror("Malloc error");
        exit(EXIT_FAILURE);
    }

    return addr;

}


// Η συνάρτηση create_malloc_double, δέχεται ως παράμετρο το πλήθος των double
// που θα γίνουν allocate, κάνει έλεγχο και επιστρέφει την διεύθυνση.
double *create_malloc_double(int len){

    double *addr = NULL;

    addr = (double *) malloc( len * sizeof(double) );

    if (addr == NULL){
```

```
        perror("Malloc error");
        exit(EXIT_FAILURE);
    }

    return addr;

}


// Η συνάρτηση menu, εμφανίζει κατάλληλο μήνυμα στο stdout και περιμένει έναν ακέραιο απο το
// stdin, το οποίο επιστρέφει. Θα μπορούσε να γίνεται έλεγχος ορθότητας του αριθμού, μέσα
// σε έναν βρόγχο, δηλαδή do{...}while(option > 0 && option < 3);
//
int menu(){
    int option;
    printf("\nChoose:\n1) Continiue\n2) Stop\n> ");
    scanf("%d", &option);
    return option;
}
```