Λογισμικό και προγραμματισμός υψηλής επίδοσης

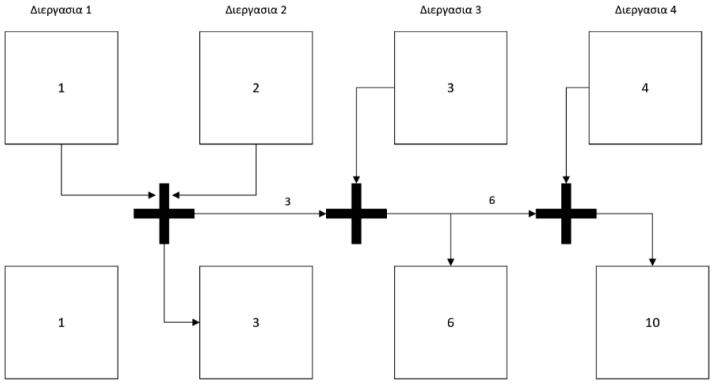
Εργασία 1

Contents

Question 1

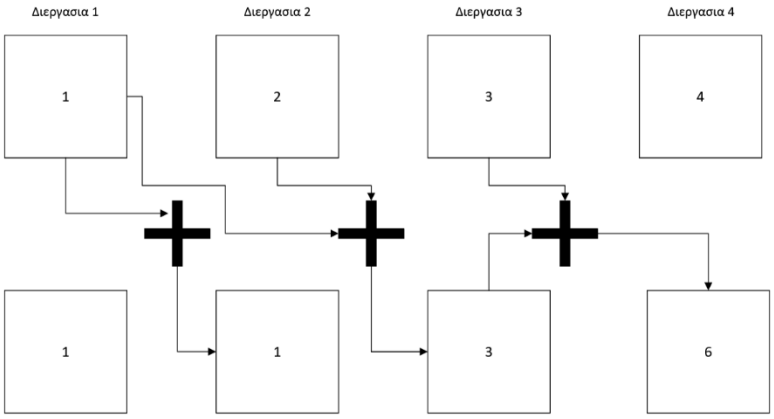
a)

Αρχικά, οφείλουμε να κατανοήσουμε την λειτουργία της exscan. Αυτό γίνεται εύκολα μελετώντας την mpi_scan. Έστω ότι έχουμε 4 διεργασίες και η καθεμιά αντίστοιχα έχει τιμή 1,2,3,4. Αυτό που κάνει είναι να προσθέτει η κάθε διεργασία το άθροισμα των τιμών των προηγουμένων διεργασιών μαζί με αυτή του εαυτού της.



Εικόνα 1: Παράδειγμα mpi_scan

Στην exscan η διαφορά είναι ότι η κάθε διεργασία παίρνει απλά το άθροισμα των προηγουμένων χωρίς να προσθέτει τον εαυτό της.



Εικόνα 2: Παράδειγμα mpi_exscan

Προχωρώντας στην υλοποίηση του κώδικα, θέτουμε τις εξής παραδοχές:

Έχουμε 4 διεργασίες με τιμές ίδιες με αυτές του παραδείγματος, δηλαδή διεργασία 1=1, διεργασία 2=2, διεργασία 3=3, διεργασία 4=4. Τότε, τα αποτελέσματα που περιμένουμε να δούμε είναι 0, 1, 3, 6.

Αρχικά, δημιουργούμε την συνάρτηση όπως ζητείτε και περιλαμβάνουμε τα απαραίτητα ορίσματα. Στο send η κάθε διεργασία αποστέλλει τον αριθμό που περιέχει, στο receive λαμβάνει το άθροισμα και στο count μετράμε τον αριθμό στοιχείων που αυτά περιέχουν.

```
// Setting up the function
int MPI_Exscan_pt2pt(const int* sendbuf, int* recvbuf, int count,
MPI_Comm comm) {
    int rank, size;
    MPI_Status status;
```

Επιπλέον, δεν ξεχνάμε να πάρουμε τον συνολικό αριθμό διεργασιών και τον αριθμό της καθεμίας.

```
// Getting the rank and size of each process
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
```

Παρακάτω, πρέπει να διαχειριστούμε ειδικά την διεργασία 1 (δηλαδή αυτή με αριθμό 0), επειδή δεν έχει κάποια πριν από αυτή άρα θα είναι 0 αφού δεν έχει άθροισμα. Ταυτόχρονα, θέλουμε να στείλει τον αριθμό της ώστε να πάει στην επόμενη.

```
// Making exception for process 1
    if (rank == 0) {
```

```
        *recvbuf = 0;
        if (size > 1) {
            MPI_Send(sendbuf, 1, MPI_INT, 1, 0, comm);
        }
        return MPI_SUCCESS;
    }
```

Τώρα πραγματοποιούμε το άθροισμα μέχρι να φτάσουμε στην τελική διεργασία.

```
// Each process receives the sum of the previous ones
    int temp = 0;
    MPI_Recv(&temp, 1, MPI_INT, rank-1, 0, comm, &status);
    *recvbuf = temp;
    temp += *sendbuf;

// If its not the final one it also sends to the next one
    if (rank < size - 1) {
        MPI_Send(&temp, 1, MPI_INT, rank+1, 0, comm);
    }

    return MPI_SUCCESS;
```

Τέλος, απλά καλούμε την συνάρτηση που δημιουργήσαμε αφού αρχικοποιησουμε καταλληλά τις απαραίτητες τιμές send και receive.

```
// Calling the function in the main
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

// Determine the rank and size of the current process/processes
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);


    int send_value = rank + 1;  // Using this so we have the correct
value as we said in the report
    int recv_value = 0;


    MPI_Exscan_pt2pt(&send_value, &recv_value, 1, MPI_COMM_WORLD);


    printf("Process %d: send = %d, received sum of previous = %d\n",
           rank, send_value, recv_value);
```

```
    MPI_Finalize();
    return 0;
}
```

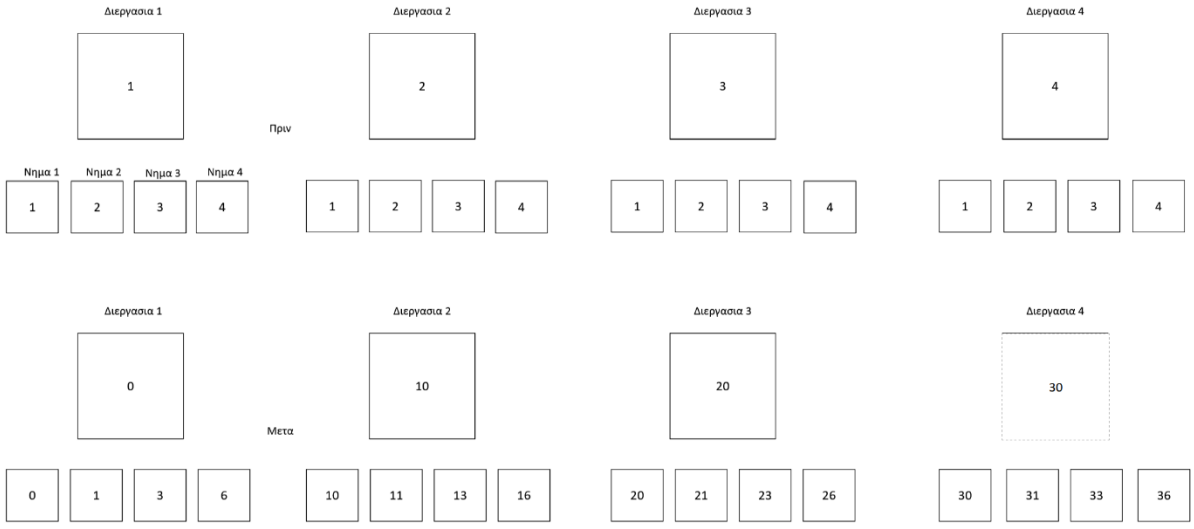
Παρατηρούμε ότι πράγματι τα αποτελέσματα είναι τα αναμενόμενα.

```
hpcgrp25@krylov100:~$ mpirun -np 4 ./exscan
Process 0: send = 1, received sum of previous = 0
Process 1: send = 2, received sum of previous = 1
Process 2: send = 3, received sum of previous = 3
Process 3: send = 4, received sum of previous = 6
```

b)

Για την υλοποίηση μας, θεωρούμε τις εξής παραδοχές:

Οι διεργασίες μας καθώς και ο αριθμός με τον οποίο αρχικοποιουνται παραμένουν ίδια με αυτά του ερωτήματος α. Επιπλέον, δημιουργούμε 4 νήματα για κάθε διεργασία και το καθένα από αυτά έχει τιμές αντίστοιχες με αυτές των διεργασιών. Δηλαδή, η κάθε διεργασία έχει νήμα 1, 2, 3, 4 με αρχικές τιμές 1, 2, 3, 4 αντίστοιχα. Αυτές οι τιμές εντός της διεργασίας προσθέτονται όπως είδαμε στο προηγούμενο παράδειγμα, δηλαδή είναι νήμα 1=1, νήμα 2=2, νήμα 3=3, νήμα 4=4. Επιπλέον, τα νήματα των επομένων διεργασιών λαμβάνουν υπόψιν το άθροισμα των προηγουμένων νημάτων. Συνεπώς, χρειαζόμαστε την omp exscan ώστε να υλοποιήσει την διαδικασία κανονικά για όλα τα νήματα και την προηγούμενη mpi exscan ώστε να μεταδώσει τις τιμές του αθροίσματος των νημάτων. Δηλαδή, οι διεργασίες θα έχουν την τιμή 0, 10, 20 ,30 αντίστοιχα.



Εικόνα 3: Παράδειγμα omp_exscan

Ας προχωρήσουμε στην ανάλυση του κώδικα. Αρχικά, ορίζουμε τον μέγιστο δυνατό αριθμό thread σε 8 (αν και χρησιμοποιούμε 4, υπάρχει η δυνατότητα και για 8) και αρχικοποιουμε τις απαραίτητες μεταβλητές μας. Ορίζουμε τα array που περιέχουν τις αρχικές τιμές των νημάτων και τα αποτελέσματα της exscan. Επιπλέον, συγκρατούμε κάθε φορά για κάθε διεργασία το άθροισμα των τιμών των νημάτων της (για όλες είναι 10) και μετα θέτουμε ως νέα τιμή το άθροισμα όλων των προηγουμένων νημάτων.

```
// Set 8 threads in case later the teams decide to do it with more
that 4
#define MAX_THREADS 8

//initialize variables
int thread_vals[MAX_THREADS];    // Stores initial values of threads
we set as said in the report
int thread_exscan[MAX_THREADS];  // Stores the result of running
exscan on each process on the threads
int sum_per_process = 0;         // Sum of all the thread initial
values each time (they are all 10)
int base_offset = 0;             // Sum of threads sum for previous
processes (they go 0,10,20,30)
```

Συνεχίζοντας, υλοποιούμε την συνάρτηση MPI_Exscan_omp. Αρχικά, όπως είπαμε κάθε νήμα πρέπει να αποθηκεύει την τιμή του (όπως και ζητείτε από την άσκηση) και φροντίζουμε με barrier ότι όλα αποθήκευσαν την τιμή τους πριν προχωρήσουμε. Παρακάτω, οφείλουμε να εξασφαλίσουμε ότι δεν θα έχουμε race conditions, άρα 1 νήμα την φορά εισέρχεται στο τμήμα υπολογισμών. Μέσα σε αυτό, θέτουμε ότι το νήμα 0 είναι πάντα 0 και μετα απλά προσθέτουμε την αρχική τιμή του προηγούμενο νήματος με την τιμή της διαδικασίας exscan (δηλαδή με ότι έχει βρει το άθροισμα έως τότε) και επιπλέον υπολογίζουμε το άθροισμα όλων των αρχικών τιμών των νημάτων της διεργασίας (σε όλες είναι 10). Παρακάτω θα δούμε πως παρακάμπτουμε ότι έχουμε θέσει σε κάθε αρχικό thread την τιμή 0.

```
void MPI_Exscan_omp(int my_val, int thread_id, int num_threads,
MPI_Comm comm, int* result) {
    // Store initial values of each thread as we said before
    thread_vals[thread_id] = my_val;

    // Cannot skip this, otherwise the program may continue without
saving initial values
    #pragma omp barrier

    // One thread computes each time so no race conditions
    #pragma omp single
    {
        // Calculating logic
        thread_exscan[0] = 0; // We set the value of thread 0 to 0 as
```

```
said in the report.
        for(int t = 1; t < num_threads; t++) {
            thread_exscan[t] = thread_exscan[t-1] + thread_vals[t-1];
        }

        // Total sum of initial values of threads in the process (its
always 10)
        sum_per_process = 0;
        for(int t = 0; t < num_threads; t++) {
            sum_per_process += thread_vals[t];
        }
    }
```

Προσθέτοντας, εξασφαλίζουμε με barrier ότι όλα τα thread έχουν πραγματοποιήσει τους υπολογισμούς τους πριν προχωρήσουμε στην αποστολή δεδομένων με Mpi. Τώρα, είναι αναγκαίο με την Mpi_exscan να μεταβιβάσουμε το υπολογισμένο άθροισμα των τιμών των νημάτων στην επόμενη διεργασία ώστε να έχουμε σωστά αποτελέσματα κάθε φορά και ταυτόχρονα θέτουμε ότι διεργασία 1=0.

```
/ Hold on until all threads have done their calculations
    #pragma omp barrier

    // One thread executes the MPI exclusive scan to get the
base_offset to counteract that thread 0=0 and make sure the values
pass on to the next process
    #pragma omp single
    {
        MPI_Exscan(&sum_per_process, &base_offset, 1, MPI_INT,
MPI_SUM, comm);

        // For process 0 (or 1 as we have used in the report) we make
sure the sum is 0
        int rank;
        MPI_Comm_rank(comm, &rank);
        if(rank == 0) {
            base_offset = 0;
        }
    }
```

Επιπλέον, αφού εξασφαλίσαμε ότι όλα τα νήματα έχουν μεταβιβάσει τα αποτελέσματα τους προσθέτουμε τις προηγούμενες τιμές που λάβαμε στο άθροισμα ώστε να πάρουμε και το σωστό αποτέλεσμα.

// Ensure that the calculations are finished

#pragma omp barrier

```
     // We actually add the previous results of the threads to get the
correct answer
    *result = base_offset + thread_exscan[thread_id];
}
```

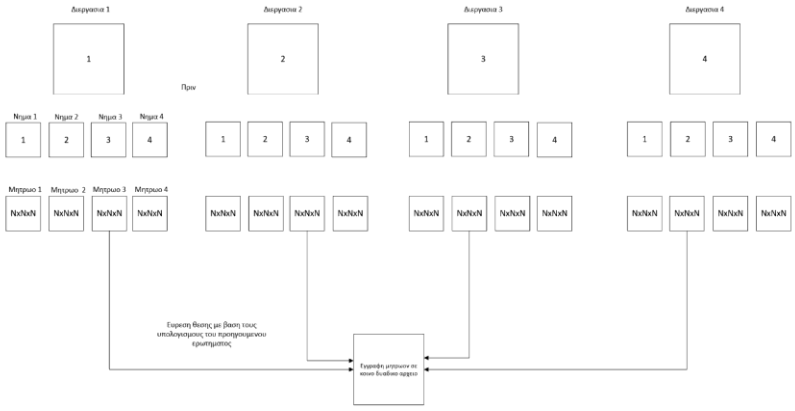
Προχωρώντας στην main, αρχικοποιουμε την Mpi και θέτουμε τις αρχικές τιμές των νημάτων. Επιπροσθέτως, αποθηκεύουμε τα αποτελέσματα του κάθε νήματος. Τέλος, αρχικοποιουμε την omp και καλούμε την νέα συνάρτηση exscan_omp για κάθε νήμα.

```
int main(int argc, char *argv[]) {
    // Initialize MPI
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);

    if(provided < MPI_THREAD_FUNNELED) {
        fprintf(stderr, "Error: MPI does not provide needed threading
level\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Define the number of threads per MPI process
    const int num_threads = 4;

    // put the initial values of the threads, they are the same for
everyone
    int initial_values[4] = {1, 2, 3, 4};

    // Final result of each thread
    int results[4] = {0};

    // Start OMP
    #pragma omp parallel num_threads(num_threads)
    {
        int thread_id = omp_get_thread_num();
        int my_val = initial_values[thread_id];
        int thread_result = 0;

        // everyone calls, all threads
        MPI_Exscan_omp(my_val, thread_id, num_threads, MPI_COMM_WORLD,
&thread_result);

        // Store the result
        results[thread_id] = thread_result;
    }
```

```
    // Synchronize all processes
    MPI_Barrier(MPI_COMM_WORLD);


    #pragma omp parallel num_threads(1)
    {
        #pragma omp single
        {
            printf("Process %d results:\n", rank);
            for(int t = 0; t < num_threads; t++) {
                printf("  Thread %d: %d\n", t, results[t]);
            }
            fflush(stdout);
        }
    }

    MPI_Finalize();
    return 0;
}
```

c)

Στη συγκεκριμένη υλοποίηση, χρειάζεται να αλλάξουμε την main. Χρειαζόμαστε επιπλέον ένα array που θα αποθηκεύει την τιμή των μητρώων και ουσιαστικά η exscan χρησιμοποιείτε για να δείξει στο κάθε νήμα σε ποια θέση πρέπει να αποθηκεύσει το αποτέλεσμα του στο κοινόχρηστο δυαδικό αρχείο.



Εικόνα 4: Παράδειγμα αποθήκευσης

Συγκεκριμένα, το τμήμα πριν από την main μένει ίδιο άρα αναλύουμε τις επιπλέον αλλαγές μας. Αναθέτουμε τις αρχικές τιμές των νημάτων με την διαφορά ότι τώρα το κάνουμε με την χρήση loop αντί για ένα array, για τον απλό λόγο ότι τώρα δεν χρειάζεται να θέσουμε χειροκίνητα τις επιθυμητές

τιμές. Επιπλέον, δημιουργούμε το αρχείο με όνομα matrix_data.bin και το ανοίγουμε.

```
// setting initial values as shown in the pic in the report
    int initial_values[MAX_THREADS];
    for(int i = 0; i < num_threads; i++) {
        initial_values[i] = i + 1;
    }

    // Result of the thread exscan
    int results[MAX_THREADS];
    memset(results, 0, sizeof(results));

    // Matrix size and data size
    const int matrix_size = N * N * N;
    const int data_size = matrix_size * sizeof(double);

    // Name of the binary file
    const char *filename = "matrix_data.bin";

    // Open the binary file
    MPI_File fh;
    MPI_Status status;
    int mpi_error;

    mpi_error = MPI_File_open(MPI_COMM_WORLD, filename,
MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
    if(mpi_error != MPI_SUCCESS) {
        fprintf(stderr, "Process %d: Error opening file for writing
and reading.\n", rank);
        MPI_Abort(MPI_COMM_WORLD, mpi_error);
    }
```

Συνεχίζοντας, υλοποιούμε την λογική των νημάτων. Φροντίζουμε ότι όλα καλούν την συνάρτηση όπως και στο προηγούμενο ερώτημα και αποθηκεύουμε τα αποτελέσματα στο array results, κοιτώντας το αντίστοιχο thread id. Δεν ξεχνάμε να αναθέσουμε το τυχαίο μητρώο σε όλα τα thread. Την θέση εγγραφής την παίρνουμε από την εξίσωση base_offset+thread_exscan[thread_id])) * data_size, όπου λαμβάνουμε υπόψιν το άθροισμα των thread των προηγούμενων διεργασιών και το προσθέτουμε με το τοπικό αποτέλεσμα της exscan όπως κάναμε και για το προηγούμενο ερώτημα, με την διαφορά ότι επηρεάζεται από το μέγεθος των δεδομένων.

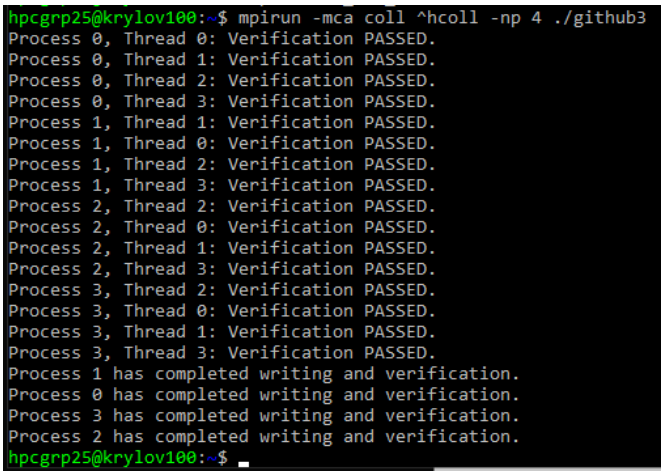
```
pragma omp parallel num_threads(num_threads)
    {
        int thread_id = omp_get_thread_num();
        int my_val = initial_values[thread_id];
```

```
        int thread_result = 0;

        // Each thread calls the MPI_Exscan_omp
        MPI_Exscan_omp(my_val, thread_id, num_threads, MPI_COMM_WORLD,
&thread_result);

        // getting the sum of everyone
        results[thread_id] = thread_result;

        // Calculate the write offset in bytes
        MPI_Offset write_offset = ((MPI_Offset)(base_offset +
thread_exscan[thread_id])) * data_size;

        // random matrix assigned to everyone
        double *matrix = (double*) malloc(data_size);
        if(matrix == NULL) {
            fprintf(stderr, "Process %d, Thread %d: Error allocating
memory for matrix.\n", rank, thread_id);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
unsigned int seed = (unsigned int)(rank * num_threads + thread_id +
1);
        for(int i = 0; i < matrix_size; i++) {
            matrix[i] = ((double)rand_r(&seed)) / RAND_MAX;
        }
```

Παρακάτω, εξασφαλίζουμε με χρήση Mpi ότι τα νήματα αποθηκεύουν τα μητρώα στην σωστή τοποθεσία και δημιουργούμε την σύγκριση αποτελεσμάτων. Άρα συγκρίνουμε τα δεδομένα του μητρώου με αυτά που διαβάζουμε από το αντίστοιχο νήμα στην τοποθεσία εγγραφής (read buffer). Αν το σφάλμα ξεπερνάει μια πολύ μικρή τιμή τότε θεωρούμε ότι η εγγραφή δεν είναι σωστή.

```
// Compare

        double *read_buffer = (double*) malloc(data_size);
        if(read_buffer == NULL) {
            fprintf(stderr, "Process %d, Thread %d: Error allocating
memory for read buffer.\n", rank, thread_id);
            free(matrix);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }


        mpi_error = MPI_File_read_at(fh, write_offset, read_buffer,
matrix_size, MPI_DOUBLE, &status);
```

```
        if(mpi_error != MPI_SUCCESS) {
            fprintf(stderr, "Process %d, Thread %d: Error reading from
file.\n", rank, thread_id);
            free(matrix);
            free(read_buffer);
            MPI_Abort(MPI_COMM_WORLD, mpi_error);
        }

        // Compare the read data with the original
        int verification_passed = 1;
        for(int i = 0; i < matrix_size; i++) {

            if(fabs(matrix[i] - read_buffer[i]) > 1e-9) {
                verification_passed = 0;
                break;
            }
        }

        if(verification_passed) {
            printf("Process %d, Thread %d: Verification PASSED.\n",
rank, thread_id);
        }
        else {
            printf("Process %d, Thread %d: Verification FAILED.\n",
rank, thread_id);
        }
```



Μπορούμε να διακρίνουμε ότι πράγματι όλα τα νήματα αποθηκεύουν με επιτυχία τις τιμές των μητρώων στον φάκελο.

d)

Ο κώδικας μας αλλάζει μόνο στα σημεία εγγραφής των μητρώων στο αρχείο και στο σημείο ελέγχου των δεδομένων, τα υπόλοιπα σημεία παραμένουν ίδια

χωρίς αλλαγή. Επιλέγουμε την συμπίεση ZLIB, όπου με αυτή υπολογίζουμε το μέγιστο δυνατό μέγεθος των συμπιεσμένων δεδομένων, συμπιέζει τα δεδομένα των μητρώων και ενημερώνει το array των αρχικών τιμών με τις νέες συμπιεσμένες τιμές για να τις χρησιμοποιήσει το exscan με σκοπό να γράφουμε σε διαδοχικές θέσεις μνήμης του φακέλου.

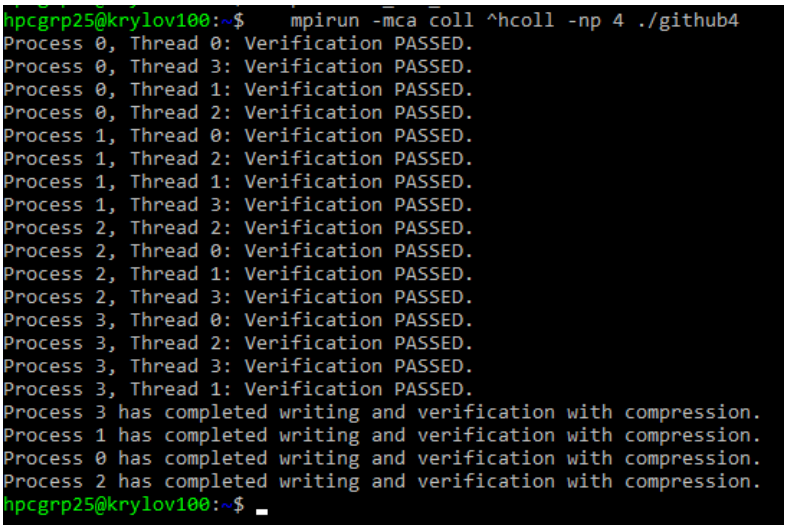
```
// Compress the matrix using ZLIB
        uLongf compressed_bound = compressBound(data_size);
        Bytef *compressed_data = (Bytef*) malloc(compressed_bound);
        if(compressed_data == NULL) {
            fprintf(stderr, "Process %d, Thread %d: Error allocating
memory for compressed data.\n", rank, thread id);
            free(matrix);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }

        int zlib_status = compress(compressed_data, &compressed_bound,
(Bytef*)matrix, data_size);
        if(zlib_status != Z_OK) {
            fprintf(stderr, "Process %d, Thread %d: Error during
compression. ZLIB status: %d\n", rank, thread_id, zlib_status);
            free(matrix);
            free(compressed_data);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }

        // Overwrite initial_values with compressed sizes for exscan
        initial_values[thread_id] = compressed_bound; // Compressed
size in bytes

free(matrix);

        // hold on to zip all the data
        #pragma omp barrier


        MPI_Exscan_omp(initial_values[thread_id], thread_id,
num_threads, MPI_COMM_WORLD, &results[thread_id]);

        // Getting the write position
        MPI_Offset write_offset = (MPI_Offset)(base_offset +
thread_exscan[thread_id]);

        // Write the compressed data to the file
        mpi_error = MPI_File_write_at(fh, write_offset,
compressed_data, compressed_bound, MPI_BYTE, &status);
        if(mpi_error != MPI_SUCCESS) {
            fprintf(stderr, "Process %d, Thread %d: Error writing
compressed data to file.\n", rank, thread_id);
```

```
            free(compressed_data);
            MPI_Abort(MPI_COMM_WORLD, mpi_error);
        }
```

Τέλος, για να πραγματοποιήσουμε τον έλεγχο τώρα χρειάζεται να αποσυμπιέσουμε τα αρχεία που αποθηκεύσαμε στο αρχείο και να τα συγκρίνουμε με τα αρχικά.

```
// Compare
        Bytef *read_compressed_data = (Bytef*)
malloc(compressed_bound);
        if(read_compressed_data == NULL) {
            fprintf(stderr, "Process %d, Thread %d: Error allocating
memory for read compressed data.\n", rank, thread_id);
            free(compressed_data);
            MPI Abort(MPI COMM WORLD, 1);
        }

        // Compare the read data with the original
        mpi_error = MPI_File_read_at(fh, write_offset,
read_compressed_data, compressed_bound, MPI_BYTE, &status);
        if(mpi_error != MPI_SUCCESS) {
            fprintf(stderr, "Process %d, Thread %d: Error reading
compressed data from file.\n", rank, thread_id);
            free(compressed_data);
            free(read_compressed_data);
            MPI_Abort(MPI_COMM_WORLD, mpi_error);
        }

        // Decompress
        Bytef *decompressed_data = (Bytef*) malloc(data_size);
        if(decompressed_data == NULL) {
            fprintf(stderr, "Process %d, Thread %d: Error allocating
memory for decompressed data.\n", rank, thread_id);
            free(compressed_data);
            free(read_compressed_data);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }

        uLongf decompressed_size = data_size;
        zlib_status = uncompress(decompressed_data,
&decompressed_size, read_compressed_data, compressed_bound);
        if(zlib_status != Z_OK) {
            fprintf(stderr, "Process %d, Thread %d: Error during
decompression. ZLIB status: %d\n", rank, thread_id, zlib_status);
            free(compressed_data);
            free(read_compressed_data);
```

```
            free(decompressed_data);
            MPI_Abort(MPI_COMM_WORLD, mpi_error);
        }


        double *original_matrix = (double*) malloc(data_size);
        if(original_matrix == NULL) {
            fprintf(stderr, "Process %d, Thread %d: Error allocating
memory for original matrix.\n", rank, thread_id);
            free(compressed_data);
            free(read_compressed_data);
            free(decompressed_data);
            MPI_Abort(MPI_COMM_WORLD, mpi_error);
        }


        seed = (unsigned int)(rank * num_threads + thread_id + 1);
        for(int i = 0; i < matrix_size; i++) {
            original_matrix[i] = ((double)rand_r(&seed)) / RAND_MAX;
        }


        int verification_passed = 1;
        for(int i = 0; i < matrix_size; i++) {

            if(fabs(original_matrix[i] -
((double*)decompressed_data)[i]) > 1e-9) {
                verification_passed = 0;
                break;
            }
        }

        if(verification_passed) {
            printf("Process %d, Thread %d: Verification PASSED.\n",
rank, thread_id);
        }
        else {
            printf("Process %d, Thread %d: Verification FAILED.\n",
rank, thread_id);
        }
```



Μπορούμε να διακρίνουμε ότι πράγματι η εγγραφή έγινε με επιτυχία ακόμα και με την χρήση συμπίεσης.

Question 2

Παραδοχές: Αφήνουμε τις τιμές των παραμέτρων του κώδικα ακριβώς όπως δίνονται χωρίς καμία αλλαγή. Χωρίζουμε τις ζητούμενες υλοποιήσεις σε 3 διαφορετικούς κώδικες, όπου σε όλες από αυτές χρησιμοποιούμε 12 διεργασίες. Ο κώδικας υλοποιείτε στο σύστημα της σχολής δημιουργώντας virtual environment με τις εντολές python3 -m venv ~/myenv, source ~/myenv/bin/activate.

Πριν ξεκινήσουμε, προσθέτουμε στον σειριακό κώδικα την time και βλέπουμε ότι χρειάζεται 27 δευτερόλεπτα κατά μέσο όρο για την εκτέλεση του. Όσο αφορά το speedup, ο χρόνος λαμβάνεται μέσα από τον τύπο $T = \frac{Ts}{Tp}$, όπου Ts=χρόνος σειριακής εκτέλεσης και Tp= χρόνος παράλληλης εκτέλεσης. Σύμφωνα με αυτό τον τύπο, ο βέλτιστος χρόνος είναι $T = \frac{27}{12} = 2.25$ sec η' σε ποσοστό $\left(1 - \frac{1}{T}\right) * 100$=66%. Για τον πραγματικό χρόνο κανονικά είναι κατάλληλος ο νομός του Amdahl, αλλά αποφασίσαμε ότι δεν χρειάζεται για την συγκεκριμένη άσκηση.

```
(myenv) hpcgrp25@krylov100:~$ python gs_serial.py
{'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 16}
0 0.9042424242424243
{'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 32}
1 0.9021212121212121
{'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 16}
2 0.9030303030303031
{'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 32}
3 0.9012121212121212
{'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 16}
4 0.9030303030303031
{'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 32}
5 0.9021212121212121
{'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 16}
6 0.9030303030303031
{'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 32}
7 0.9027272727272727
Total runtime with serial execution: 27.51 seconds
(0, {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.9042424242424243)
(1, {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.9021212121212121)
(2, {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.9030303030303031)
(3, {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.9012121212121212)
(4, {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.9030303030303031)
(5, {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.9021212121212121)
(6, {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.9030303030303031)
(7, {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.9027272727272727)
(myenv) hpcgrp25@krylov100:~$
```

Multiprocessing pool

Με την συγκεκριμένη υλοποίηση, στόχος μας είναι να ξεπεράσουμε το thread lock της Python δημιουργώντας απλά διεργασίες αντί για thread. Τότε, ορίζουμε τον αριθμό διεργασιών που θέλουμε να δημιουργήσουμε (επιλέξαμε 12 για όλες τις υλοποιήσεις καθώς παρατηρήσαμε ότι είχε τον καλύτερο χρόνο για όλα) με την εντολή

```
pool = multiprocessing.Pool(processes=12)
```

διαμοιράζουμε σε όλες τις διεργασίες τα δεδομένα που θέλουμε να υπολογίζουν (δηλαδή δίνουμε στην καθεμιά subset των αρχικών δεδομένων να υπολογίσει)

```
results = pool.map(evaluate_params, pg)
```

και τέλος εξασφαλίζουμε πως καμία διεργασία δεν θα προστεθεί αφού διαμοιράσαμε τα task και τοποθετούμε το join στο τέλος της παραλληλοποιησης.

```
pool.close()
pool.join()
```

```
(myenv) hpcgrp25@krylov100:~$ python gs_parallel.py
Run 0: Parameters: {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 16}, Accuracy: 0.9048484848484849
Run 1: Parameters: {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 32}, Accuracy: 0.9027272727272727
Run 2: Parameters: {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 16}, Accuracy: 0.8978787878787878
Run 3: Parameters: {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 32}, Accuracy: 0.9045454545454545
Run 4: Parameters: {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 16}, Accuracy: 0.9045454545454545
Run 5: Parameters: {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 32}, Accuracy: 0.9033333333333333
Run 6: Parameters: {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 16}, Accuracy: 0.9015151515151515
Run 7: Parameters: {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 32}, Accuracy: 0.9033333333333333
Total runtime with multiprocessing: 4.73 seconds
(myenv) hpcgrp25@krylov100:~$
```

Ο χρόνος εκτέλεσης μειώθηκε σε 4.73 sec, άρα με τον τύπο που αναφέραμε έχουμε $T = \frac{27}{4.73} = 5.7$ φορές γρηγορότερο από την σειριακή έκδοση. Όπως

βλέπουμε, απέχει αρκετά από το ιδανικό, συγκεκριμένα πετυχαίνουμε περίπου 47% της βέλτιστης απόδοσης. Για όλες τις υλοποιήσεις μας λαμβάνουμε παρόμοια αποτελέσματα, άρα θεωρούμε ότι οι παραπάνω υπολογισμοί ισχύουν για όλες. Δεν γίνεται να πέτυχουμε το βέλτιστο αποτέλεσμα με python, καθώς δεν έχουμε νήματα και με χρήση mpi έχουμε καθυστερήσεις. Αν το κάναμε χωρίς pool, θα έπρεπε να γράψουμε με το χέρι μια μια τις διεργασίες, τι θέλουμε να εκτελέσουν και να τις αρχίζουμε (με start) και να τις τερματίζουμε (με join).

Mpi futures

Σε αυτή την υλοποίηση, επαναλαμβάνουμε την προηγούμενη διαδικασία, αλλά με την χρήση Mpi για τον διαμοιρασμό task στις διεργασίες και την μεταφορά δεδομένων μεταξύ τους. Όλη η διαδικασία διαχειρίζεται αυτόματα από το MPICommExecutor, δηλαδή αναλαμβάνει την ανάθεση task καθώς και την αποστολή δεδομένων. Είναι η ίδια λογική με απλή mpi, απλά αλλάζουμε το format να είναι για python.

```
# Using MPI futures
    with MPICommExecutor() as executor:
        if executor is not None:  # Main process
            comp_start_time = time.time()

            # Map parameter sets to worker processes
            results = list(executor.map(evaluate_parameters,
param_sets))
```

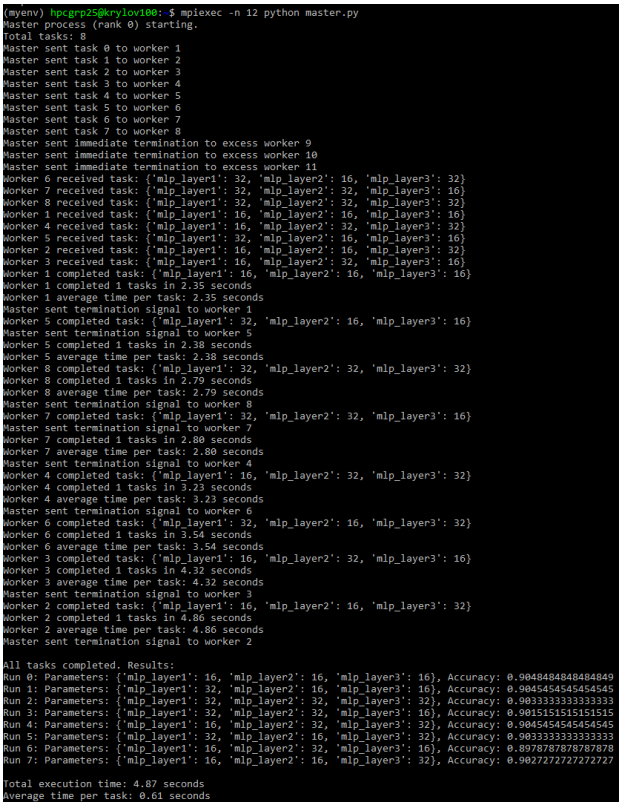
```
(myenv) hpcgrp25@krylov100:~$ mpiexec -n 12 python futures.py

Results:
(0, {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.9048484848484849)
(1, {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.9027272727272727)
(2, {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8978787878787878)
(3, {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.9045454545454545)
(4, {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.9045454545454545)
(5, {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.9033333333333333)
(6, {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.9015151515151515)
(7, {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.9033333333333333)

Timing Information:
Total execution time: 4.63 seconds
Computation time: 4.63 seconds
```

Master-Worker

Σε αντίθεση με τα future, δεν εκτελούν όλες οι διεργασίες ισότιμα την δουλειά. Συγκεκριμένα, ορίζουμε ότι η διεργασία 0 παράγει τα δεδομένα και τα μοιράζει στις υπόλοιπες. Επιπλέον, αυτή (διεργασία 0) συλλεγεί τα δεδομένα από τους worker και τους αναθέτει εκ νέου task εφόσον τελείωσαν με αυτό που εκτελούσαν, δηλαδή η ανάθεση είναι δυναμική.

```
    if rank == 0:
        print(f"Master process (rank {rank}) starting.")
        # Generate data
        X, y = make_classification(
            n_samples=10000,
            random_state=42,
            n_features=2,
            n_informative=2,
            n_redundant=0,
            class_sep=0.8
        )

        # Split sets
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=0.33, random_state=42
        )

        # Define hyperparameter grid
        params = [{
            'mlp_layer1': [16, 32],
            'mlp_layer2': [16, 32],
            'mlp_layer3': [16, 32]
        }]
        pg = list(ParameterGrid(params))
        num_tasks = len(pg)
        results = []
        print(f"Total tasks: {num_tasks}")

        # Initialize task index
        task_index = 0

        # Distribute initial tasks to workers
        for worker in range(1, min(size, num_tasks + 1)):
            p = pg[task_index]
            comm.send((p, X_train, X_test, y_train, y_test),
dest=worker, tag=11)
            print(f"Master sent task {task_index} to worker {worker}")
            task_index += 1

        # Assign remaining tasks dynamically
        for _ in range(num_tasks):
            # Receive result from any worker
            status = MPI.Status()
            result = comm.recv(source=MPI.ANY_SOURCE, tag=22,
status=status)
            source = status.Get_source()
            results.append(result)
            print(f"Master received result from worker {source}:
```

```
{result}")

            # Assign next task if available
            if task_index < num_tasks:
                p = pg[task_index]
                comm.send((p, X_train, X_test, y_train, y_test),
dest=source, tag=11)
                print(f"Master sent task {task_index} to worker
{source}")
                task_index += 1
            else:
                # No more tasks; send termination signal
                comm.send(None, dest=source, tag=11)
                print(f"Master sent termination signal to worker
{source}")

        # After all tasks are completed, print the results
        print("\nAll tasks completed. Printing results:")
        for idx, (p, acc) in enumerate(results):
            print(f"Run {idx}: Parameters: {p}, Accuracy: {acc}")

    else:
        # Worker Processes
        while True:
            data = comm.recv(source=0, tag=11)
            if data is None:
                # Termination signal received
                print(f"Worker {rank} received termination signal.")
                break
            p, X_train, X_test, y_train, y_test = data
            print(f"Worker {rank} received task: {p}")
            result = evaluate_params(p, X_train, X_test, y_train,
y_test)
            comm.send(result, dest=0, tag=22)
            print(f"Worker {rank} completed task: {p}")
```



Question 3

a)

Υποθέτουμε ότι έχουμε διάσταση N=100 και 4 νήματα. Αν η υλοποίηση μας ήταν static, οι επαναλήψεις μοιράζονται ως εξής:

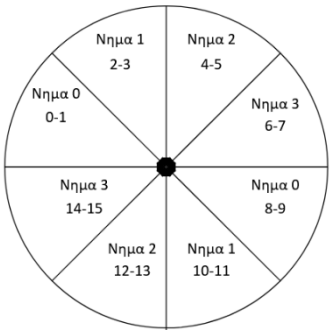
Νήμα 0=0-24

Νήμα 1=24-49

Νήμα 2=50-74

Νήμα 3=75-99

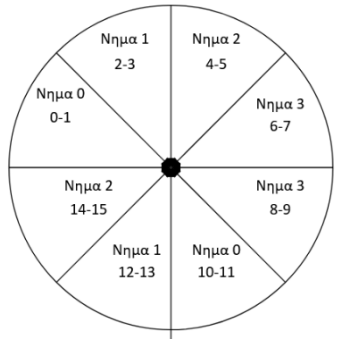
Παρατηρούμε ότι έχουν ισότιμη κατανομή, δηλαδή όλα εκτελούν τον ίδιο φόρτο εργασίας. Με chunk size=2 που στην συγκεκριμένη υλοποίηση, η κατανομή στην πραγματικότητα είναι νήμα 0=0-1, νήμα 1=2-3, νήμα 2=4-5, νήμα 3=6-7, νήμα 0=8-9 κλπ.



Εικόνα 5: Παράδειγμα static parallel for

Το πρόβλημα μας είναι ότι σε περίπτωση που κάποιο από τα νήματα τελειώσει γρηγορότερα από τα υπόλοιπα, πρέπει να περιμένει την σειρά του για την ανάθεση επαναλήψεων, δηλαδή μένει ανενεργό χωρίς να εκτελεί κάτι.

Για αυτό τον λόγο στην συγκεκριμένη υλοποίηση χρησιμοποιούμε dynamic scheduling που είναι μια μορφή first come first served αλγορίθμου. Επεξηγηματικά, οι επαναλήψεις δεν μοιράζονται με την σειρά, αλλά ανάλογα με το ποιο νήμα τελειώνει πιο γρήγορα. Για παράδειγμα, αν το νήμα 3 τελειώσει τις επαναλήψεις του 6-7 γρηγορότερα από ότι το νήμα 0 τις 0-1, τότε σε αυτό ανατρέπονται οι επαναλήψεις 8-9 αντί για το νήμα 0 ώστε να μην μένει ανενεργό.



Εικόνα 6: Παράδειγμα dynamic parallel for

b)

Για να γράψουμε ισοδύναμη υλοποίηση, οφείλουμε να κατανοήσουμε πρώτα την λειτουργία του parallel for. Όπως είδαμε, διαμοιράζει βρόχους στα νήματα. Κάθε βρόγχος είναι ένα task, άρα εμείς πρέπει αρχικά να τα δημιουργήσουμε και μετα να τα αναθέσουμε προς εκτέλεση. Τότε, το μόνο που έχουμε να κάνουμε είναι να δημιουργούμε task από ένα μόνο thread, αλλιώς στο παράδειγμα μας θα τρέχει ο κώδικας δημιουργίας 4 φορές και άρα θα έχουμε 4 φορές τον αριθμό των task που θέλουμε και μετα η dynamic διαμοίραση γίνεται αυτόματα από το omp περιβάλλον στα task.

```
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);
                }
            }
        }
    }
}
```

Μέσα στην παράλληλη περιοχή, ένας δημιουργεί τα task, αυτά τοποθετούνται στην ουρά και αφού έχουν δημιουργηθεί όλα (δηλαδή να τελειώσει αναγκαστικά πρώτα το single, έχει implicit barrier) και μετα όλα μαζί εκτελούν τα task με δυναμικό τρόπο (first come first served).