Software and Programming of High Performance Systems

Exercise 1

Frountzos Maximos

Papageorgiou Spyros

(All codes can also be found in the zip)

Question 1

a)

```c
void MPI_Exscan_pt2pt(const int *sendbuf, int *recvbuf, int count, MPI_Op op, MPI_Comm comm)
{

    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    int *tempbuf = (int *)malloc(count * sizeof(int));
    int *partial = (int *)malloc(count * sizeof(int));

    //se periptwsh poy den exei arxikopoihthei se 0 h se 1 (oudeteres times gia sum kai prod)
    for (int i = 0; i < count; i++)
    {
        recvbuf[i] = 0;
    }

    // indata
    for (int i = 0; i < count; i++)
    {
        partial[i] = sendbuf[i];
    }

    if (rank > 0) {

        MPI_Recv(tempbuf, count, MPI_INT, rank - 1, 0, comm, MPI_STATUS_IGNORE);

        for (int i = 0; i < count; i++)
        {
            recvbuf[i] = tempbuf[i];
            if (op == MPI_SUM)
                partial[i] += tempbuf[i];
            else if (op == MPI_PROD)
                partial[i] *= tempbuf[i];
        }
    }

    if (rank < size - 1)
        MPI_Send(partial, count, MPI_INT, rank + 1, 0, comm);

    free(tempbuf);
    free(partial);
}
```

The implementation of the requested function is shown above. The function has been implemented to calculate partial sum and partial product. We also assume that the user is running the program with a number of processes >= 2.

Function Explanation:

All processes execute the function, and all but rank0 "bloxk" on the MPI_Recv command waiting to receive a message. rank0 is the only one that sends partial initially (partial is the variable that contains the partial sum/product of the previous processes and is therefore 0 for rank0). The partial is sent each time only to the next process, which uses it to make the new partial. The process continues for all processes in order. We notice that the MPI_Send command is executed for rank < size – 1 which means that the last process does not send to someone (obviously, since there is no next one).

By executing the function (where the indata for each process is 10, 20 ,30...):

```c
int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int indata = 10*(rank+1);
    int outdata = 0;

    MPI_Exscan_pt2pt(&indata, &outdata, 1, MPI_SUM, MPI_COMM_WORLD);
    printf("process %d: %d -> %d\n", rank, indata, outdata);

    MPI_Finalize();
    return 0;
}
```

We have the following results for 4 processes:

```
(base) (base) max@DESKTOP-548569G:~/hpc$ mpirun -np 4 ./1a
process 0: 10 -> 0
process 1: 20 -> 10
process 2: 30 -> 30
process 3: 40 -> 60
```

Respectively for 6:

```
(base) (base) max@DESKTOP-548569G:~/hpc$ mpirun -np 6 ./1a
process 0: 10 -> 0
process 1: 20 -> 10
process 2: 30 -> 30
process 3: 40 -> 60
process 4: 50 -> 100
process 5: 60 -> 150
```

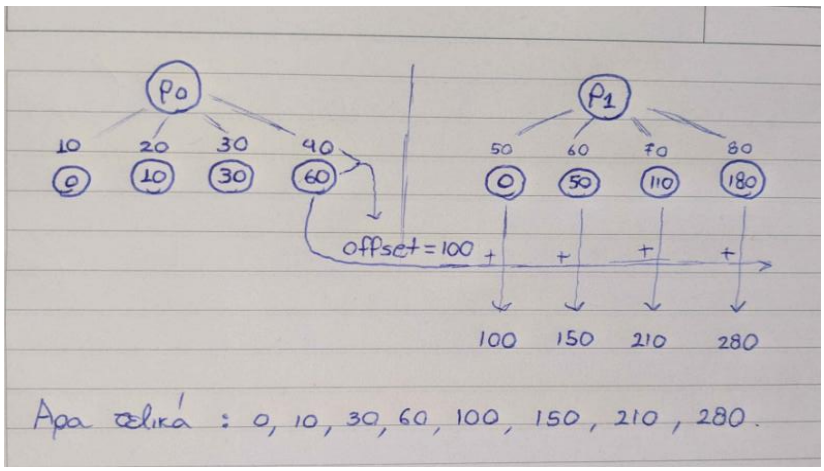
b)

```
void MPI_Exscan_omp(const int *sendbuf, int *recvbuf, MPI_Comm comm, int *offset)
{
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    int id = omp_get_thread_num();

    //se periptwsh poy den exei arxikopoihthei se 0
    *recvbuf = 0;

    if (rank > 0)
    {
        int first_thread = 0; //tha mporousa kai pragma omp single alla thelo explicit
        if (id == first_thread)
            MPI_Recv(offset, 1, MPI_INT, rank - 1, 0, comm, MPI_STATUS_IGNORE);
        #pragma omp barrier
    }

    for (int i = 0; i < id; i++)
    {
        *recvbuf += sendbuf[i];
    }

    //prosthetw to offset to opoio einai 0 gia rank = 0
    *recvbuf += *offset;

    if (rank < size - 1)
    {
        int last_thread = omp_get_num_threads() - 1;
        if (id == last_thread)
        {
            int offset_value = *recvbuf + sendbuf[last_thread];
            MPI_Send(&offset_value, 1, MPI_INT, rank + 1, 0, comm);
        }
    }
}
```

To extend the function to the hybrid model, we removed the possibility of partial products and the count parameter, considering that it is always 1 for simplicity and as they will not be needed for subsequent queries.

Also, indata is now a shared table between threads so that each thread can calculate the partial sum of the previous ones in parallel. (There is no need for thread1 to wait for thread0, and so on.). Of course, we must ensure that the table has been filled in by all threads before the MPI_Exscan_omp() starts, so that we do not have race conditions (with a barrier).

We added the offset parameter which is the sum of the indata of all threads in a process. That is, it is the partial sum of the first thread of the next process. This is calculated on the last thread of each process (int offset_value = *recvbuf + sendbuf[last_thread];) and sent from the same thread to the next thread. The threads of the next process normally calculate the partial sums and then the offset (*recvbuf += *offset;) is added to calculate the final partial sum of the thread of a process taking into account the threads of the previous process.



Shown above is an example of 2 processes with 4 threads each. Above each thread is its indata and within the circle the result of exscan() i.e. the partial sum of the previous ones.

By running the program (barrier makes sure that the indata table is fully populated

```c
int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int num_threads = 2;
    int indata[num_threads];
    int offset = 0; //to offset den einai to offset se arxeio alla to meriko athroisma toy prwtoy thread toy epomenoy process

    #pragma omp parallel num_threads(num_threads) shared(indata, offset)
    {
        int id = omp_get_thread_num();
        indata[id] = 10 * (rank * num_threads + id + 1);
        int outdata = 0;
        #pragma omp barrier // sigoureuw oti to indata[id] exei parei oles tis times

        MPI_Exscan_omp(indata, &outdata, MPI_COMM_WORLD, &offset);
        #pragma omp critical
        {
            printf("MPI process %d: Thread %d, indata=%d, outdata=%d\n", rank, id, indata[id], outdata);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Result with indata 10,20,30... for 4 and 6 processes respectively:

```
(base) (base) max@DESKTOP-548569G:~/hpc$ mpirun -np 4 ./1b
MPI process 0: Thread 0, indata=10, outdata=0
MPI process 0: Thread 1, indata=20, outdata=10
MPI process 1: Thread 0, indata=30, outdata=30
MPI process 1: Thread 1, indata=40, outdata=60
MPI process 2: Thread 0, indata=50, outdata=100
MPI process 2: Thread 1, indata=60, outdata=150
MPI process 3: Thread 0, indata=70, outdata=210
MPI process 3: Thread 1, indata=80, outdata=280
```

```
(base) (base) max@DESKTOP-548569G:~/hpc$ mpirun -np 6 ./1b
MPI process 0: Thread 0, indata=10, outdata=0
MPI process 0: Thread 1, indata=20, outdata=10
MPI process 1: Thread 0, indata=30, outdata=30
MPI process 1: Thread 1, indata=40, outdata=60
MPI process 2: Thread 0, indata=50, outdata=100
MPI process 2: Thread 1, indata=60, outdata=150
MPI process 3: Thread 0, indata=70, outdata=210
MPI process 3: Thread 1, indata=80, outdata=280
MPI process 4: Thread 0, indata=90, outdata=360
MPI process 4: Thread 1, indata=100, outdata=450
MPI process 5: Thread 1, indata=120, outdata=660
MPI process 5: Thread 0, indata=110, outdata=550
```

c)

Code Part 1

```
int main(int argc, char *argv[])
{
    int rank, size;
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

    if (provided < MPI_THREAD_MULTIPLE)
    {
        printf("Error: MPI does not support required multithreading level.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

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

    int num_threads = 4;
    int indata[num_threads];
    int offset = 0; //to offset den einai to offset se arxeio alla to meriko athroisma toy prwtoy thread toy epomenoy process

    int step = 420;
    char filename[256];
    sprintf(filename, "mydata_%05d.bin", step);

    MPI_File f;
    MPI_File_open(MPI_COMM_WORLD, filename , MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &f);

    MPI_File_set_size (f, 0);
    MPI_Offset base;
    MPI_File_get_position(f, &base);

    const int nlocal = N*N*N;
    int len = nlocal*sizeof(int);
```

For this particular question we will need to make MPI calls through many threads (in the previous questions we had mpi_recv and mpi_send through threads but it was always done by one thread, the last of each process sending and the first of the next receive). Therefore, we need to do the necessary configuration in the MPI:

MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

Then each process using MPI I/O opens or creates if there is no binary file in read-write mode and we define the parallel region as shown below.

Code Part 2

```
#pragma omp parallel num_threads(num_threads) shared(indata, offset)
{
    int id = omp_get_thread_num();
    int matrix[N][N][N];
    int seed = 10 * (rank * num_threads + id + 1);
    initialize_matrix(matrix, seed);

    // #pragma omp critical
    // {
    //     print_matrix(matrix, rank, id);
    // }

    indata[id] = len;
    int fileOffset = 0;
    #pragma omp barrier // sigoureuw oti to indata[id] exei parei oles tis times

    MPI_Exscan_omp(indata, &fileOffset, MPI_COMM_WORLD, &offset);

    MPI_File_write_at_all(f, base + ((MPI_Offset)fileOffset), matrix, nlocal, MPI_INT, MPI_STATUS_IGNORE);
    #pragma omp barrier //sigoyreuw oti exei teleiwsei h eggrafi toy arxeio prin ksekinisw thn anagnwsh

    // elegxos swsths eggrafhs
    int read_matrix[N][N][N];
    MPI_File_read_at_all(f, base + ((MPI_Offset)fileOffset), read_matrix, nlocal, MPI_INT, MPI_STATUS_IGNORE);

    #pragma omp critical
    {
        if(!equal_matrices(read_matrix, matrix))
        {
            printf("Process: %d, Thread: %d -> Error in writing!\n", rank, id);
        }
        else
        {
            printf("Process: %d, Thread: %d -> Correct!\n", rank, id);
        }
    }
}

MPI_File_close(&f);
MPI_Finalize();
return 0;
```

Each thread initializes a random matrix using the same always-separate seed and thread safe rand_r (we limit the numbers from 0-99 for simplicity):

```
void initialize_matrix(int (*matrix)[N][N], int seed)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            for (int k = 0; k < N; k++)
            {
                matrix[i][j][k] = rand_r(&seed) % 100; //0-99
            }
        }
    }
}
```

We set indata = len which is the length of the text that each thread will write and the exScan_omp function is called to calculate the offset in which each thread should write. Then the write_at_all function is called to make the parallel write and we use a barrier because then we want to read and thus make sure that there will be no parallel read and write in one process. We do the reading and compare the original matrix with what we read.

It should be emphasized that the comparison is made in each thread and therefore each thread compares the matrix it created with the one it read. So the data in the binary is only correct if all threads return success.

Results of 2 threads with 4 and 6 processes respectively:

```
(base) (base) max@DESKTOP-548569G:~/hpc$ mpirun -np 4 ./1c
Process: 0, Thread: 1 -> Correct!
Process: 0, Thread: 0 -> Correct!
Process: 1, Thread: 1 -> Correct!
Process: 1, Thread: 0 -> Correct!
Process: 2, Thread: 0 -> Correct!
Process: 2, Thread: 1 -> Correct!
Process: 3, Thread: 1 -> Correct!
Process: 3, Thread: 0 -> Correct!
```

```
(base) (base) max@DESKTOP-548569G:~/hpc$ mpirun -np 6 ./1c
Process: 0, Thread: 1 -> Correct!
Process: 1, Thread: 1 -> Correct!
Process: 2, Thread: 1 -> Correct!
Process: 3, Thread: 1 -> Correct!
Process: 4, Thread: 1 -> Correct!
Process: 4, Thread: 0 -> Correct!
Process: 5, Thread: 0 -> Correct!
Process: 5, Thread: 1 -> Correct!
Process: 0, Thread: 0 -> Correct!
Process: 1, Thread: 0 -> Correct!
Process: 2, Thread: 0 -> Correct!
Process: 3, Thread: 0 -> Correct!
```

The correct registration is also shown by using extensions where the hexadecimal digits match the initials. (Every 2 lines writes each thread since it writes for N=2, 4bytes*8 = 256bit = 64Hex)

mydata_00420.bin

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  3D 00 00 00 56 00 00 00 0B 00 00 00 44 00 00 00
00000010  4B 00 00 00 32 00 00 00 63 00 00 00 0B 00 00 00
00000020  0E 00 00 00 41 00 00 00 3F 00 00 00 28 00 00 00
00000030  18 00 00 00 13 00 00 00 0A 00 00 00 5B 00 00 00
00000040  44 00 00 00 49 00 00 00 44 00 00 00 0C 00 00 00
00000050  14 00 00 00 24 00 00 00 45 00 00 00 2F 00 00 00
00000060  2D 00 00 00 1C 00 00 00 61 00 00 00 24 00 00 00
00000070  11 00 00 00 36 00 00 00 04 00 00 00 4F 00 00 00
00000080  33 00 00 00 3C 00 00 00 31 00 00 00 20 00 00 00
00000090  2A 00 00 00 63 00 00 00 28 00 00 00 3B 00 00 00
000000A0  04 00 00 00 0F 00 00 00 02 00 00 00 50 00 00 00
000000B0  26 00 00 00 28 00 00 00 1B 00 00 00 27 00 00 00
000000C0  51 00 00 00 17 00 00 00 52 00 00 00 33 00 00 00
000000D0  23 00 00 00 21 00 00 00 56 00 00 00 2F 00 00 00
000000E0  23 00 00 00 4E 00 00 00 23 00 00 00 63 00 00 00
000000F0  1F 00 00 00 37 00 00 00 49 00 00 00 1C 00 00 00
00000100  +
```

d)

```
#pragma omp parallel num_threads(num_threads) shared(indata, offset)
{
    int id = omp_get_thread_num();
    int matrix[N][N][N];
    int seed = 10 * (rank * num_threads + id + 1);
    initialize_matrix(matrix, seed);

    void *compressed_matrix = NULL;
    size_t compressed_size = compress_matrix(matrix, sizeof(matrix), &compressed_matrix);
    indata[id] = (int)compressed_size;

    //indata[id] = len;
    int fileOffset = 0;
    #pragma omp barrier // sigoureuw oti to indata[id] exei parei oles tis times

    MPI_Exscan_omp(indata, &fileOffset, MPI_COMM_WORLD, &offset);

    MPI_File_write_at_all(f, base + ((MPI_Offset)fileOffset), compressed_matrix, compressed_size, MPI_BYTE, MPI_STATUS_IGNORE);
    #pragma omp barrier //sigoyreuw oti exei teleiwsei h eggrafi toy arxeio prin ksekinisw thn anagnwsh

    void *read_compressed_matrix = malloc(compressed_size);
    MPI_File_read_at_all(f, base + ((MPI_Offset)fileOffset), read_compressed_matrix, compressed_size, MPI_BYTE, MPI_STATUS_IGNORE);

    int decompressed_matrix[N][N][N];
    decompress_matrix(read_compressed_matrix, compressed_size, decompressed_matrix, sizeof(decompressed_matrix));


    free(compressed_matrix);
    free(read_compressed_matrix);

    #pragma omp critical
    {
        if(!equal_matrices(decompressed_matrix, matrix))
        {
            printf("Process: %d, Thread: %d -> Error in writing!\n", rank, id);
        }
        else
        {
            printf("Process: %d, Thread: %d -> Correct!\n", rank, id);
        }
    }
}
```

We show only the parallel area as the rest of the program is the same. In this query we compress the original matrix with the compress_matrix (ZLIB):

```
size_t compress_matrix(const void *src, size_t src_size, void **dest)
{
    uLongf dest_size = compressBound(src_size);
    *dest = malloc(dest_size);
    if (compress(*dest, &dest_size, src, src_size) != Z_OK)
    {
        fprintf(stderr, "Compression fail\n");
        free(*dest);
        *dest = NULL;
        return 0;
    }
    return dest_size;
}
```

The function uses compressBound to return the size of the compressed registry and uses compress to do the compression on the matrix by reference.

The size of the compressed matrix initializes the indata table and exscan calculates the write offsets. We write, read and decompress the matrix which we compare with the original one to verify correct registration.

Decompression:

```c
int decompress_matrix(const void *src, size_t src_size, void *dest, size_t dest_size)
{
    uLongf decompressed_size = dest_size;
    if (uncompress(dest, &decompressed_size, src, src_size) != Z_OK)
    {
        fprintf(stderr, "Decompression fail\n");
        return 0;
    }

    // prepei na exoyn idio size
    if (decompressed_size != dest_size)
    {
        fprintf(stderr, "size mismatch!");
        return 0;
    }
    return 1;  // Success
}
```

We see that the process is done correctly:

```
(base) (base) max@DESKTOP-548569G:~/hpc$ mpirun -np 4 ./1d
Process: 0, Thread: 0 -> Correct!
Process: 0, Thread: 1 -> Correct!
Process: 1, Thread: 1 -> Correct!
Process: 1, Thread: 0 -> Correct!
Process: 2, Thread: 1 -> Correct!
Process: 2, Thread: 0 -> Correct!
Process: 3, Thread: 0 -> Correct!
Process: 3, Thread: 1 -> Correct!
```

```
(base) (base) max@DESKTOP-548569G:~/hpc$ mpirun -np 6 ./1d
Process: 2, Thread: 1 -> Correct!
Process: 3, Thread: 1 -> Correct!
Process: 4, Thread: 0 -> Correct!
Process: 5, Thread: 0 -> Correct!
Process: 0, Thread: 1 -> Correct!
Process: 1, Thread: 0 -> Correct!
Process: 5, Thread: 1 -> Correct!
Process: 0, Thread: 0 -> Correct!
Process: 2, Thread: 0 -> Correct!
Process: 3, Thread: 0 -> Correct!
Process: 4, Thread: 1 -> Correct!
Process: 1, Thread: 1 -> Correct!
```

We can see the same from the extension where the number of registration data has been reduced:

mydata_00420.bin

```
         00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  78 9C B3 65 60 60 08 03 62 6E 20 76 01 62 6F 20
00000010  36 02 E2 64 A8 18 00 21 CC 01 CE 78 9C E3 63 60
00000020  60 70 04 62 7B 20 D6 00 62 09 20 16 06 62 2E 20
00000030  8E 06 62 00 16 24 01 47 78 9C 73 61 60 60 F0 04
00000040  62 17 20 E6 01 62 11 20 56 01 62 57 20 D6 07 62
00000050  00 1D C0 01 8A 78 9C D3 65 60 60 90 01 E2 44 20
00000060  56 01 62 41 20 36 03 62 16 20 F6 07 62 00 19 AC
00000070  01 69 78 9C 33 66 60 60 B0 01 62 43 20 56 00 62
00000080  2D 20 4E 06 62 0D 20 B6 06 62 00 1D 98 01 B1 78
00000090  9C 63 61 60 60 E0 07 62 26 20 0E 00 62 35 20 D6
000000A0  00 62 69 20 56 07 62 00 0E 68 00 F6 78 9C 0B 64
000000B0  60 60 10 07 E2 20 20 36 06 62 65 20 56 04 E2 30
000000C0  20 D6 07 62 00 1F 98 01 B7 78 9C 53 66 60 60 F0
000000D0  03 62 65 20 4E 06 62 79 20 36 07 62 4F 20 96 01
000000E0  62 00 1F 48 01 B3 +
```

Question 2

Multiprocessing pool:

```python
import time
from multiprocessing import Pool
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import ParameterGrid
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

X, y = make_classification(n_samples=100000, random_state=42, n_features=2, n_informative=2, n_redundant=0, class_sep=0.8)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

params = [{'mlp_layer1': [16, 32],
           'mlp_layer2': [16, 32],
           'mlp_layer3': [16, 32]}]

pg = list(ParameterGrid(params))

def work(p):
    l1 = p['mlp_layer1']
    l2 = p['mlp_layer2']
    l3 = p['mlp_layer3']
    m = MLPClassifier(hidden_layer_sizes=(l1, l2, l3))
    m.fit(X_train, y_train)
    y_pred = m.predict(X_test)
    ac = accuracy_score(y_pred, y_test)
    return (p, ac)

if __name__ == "__main__":
    start_time = time.time()
    p = Pool(4)
    results = list(p.map(work, pg))

    for r in results:
        print(r)

    end_time = time.time()
    execution_time = end_time - start_time
    print(f"Total execution time: {execution_time:.2f} seconds")
```

By using Pool(4) we use 4 processes to divide the work.

Mpi futures:

```python
import time
from mpi4py.futures import MPIPoolExecutor
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import ParameterGrid
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

X, y = make_classification(n_samples=100000, random_state=42, n_features=2, n_informative=2, n_redundant=0, class_sep=0.8)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

params = [{'mlp_layer1': [16, 32],
           'mlp_layer2': [16, 32],
           'mlp_layer3': [16, 32]}]

pg = list(ParameterGrid(params))

def work(p):
    l1 = p['mlp_layer1']
    l2 = p['mlp_layer2']
    l3 = p['mlp_layer3']
    m = MLPClassifier(hidden_layer_sizes=(l1, l2, l3))
    m.fit(X_train, y_train)
    y_pred = m.predict(X_test)
    ac = accuracy_score(y_pred, y_test)
    return (p, ac)

if __name__ == "__main__":
    start_time = time.time()
    p = MPIPoolExecutor()
    if p is not None:
        results = list(p.map(work, pg))

    for r in results:
        print(r)

    end_time = time.time()
    execution_time = end_time - start_time
    print(f"Total execution time: {execution_time:.2f} seconds")
```

Similarly, we use MPIPoolExecutor to distribute the load to as many processes as we identify during execution.

Master-Worker:

Part 1 (Master)

```python
import time
import numpy as np
from mpi4py import MPI
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import ParameterGrid
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

X, y = make_classification(n_samples=100000, random_state=42, n_features=2, n_informative=2, n_redundant=0, class_sep=0.8)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

params = [{'mlp_layer1': [16, 32],
           'mlp_layer2': [16, 32],
           'mlp_layer3': [16, 32]}]

pg = list(ParameterGrid(params))

if rank == 0:
    start_time = time.time()

    # o master xwrizei ta tasks kai ta stelnei se kathe worker
    worker_tasks = np.array_split(pg, size - 1)

    for i, tasks in enumerate(worker_tasks):
        comm.send(tasks, dest=i+1, tag=0)

    # dexetai ta work(tasks) apo olous toys workers
    results = []
    for i in range(1, size):
        worker_results = comm.recv(source=i, tag=0)
        results.extend(worker_results)

    for r in results:
        print(r)

    end_time = time.time()
    execution_time = end_time - start_time
    print(f"Total execution time: {execution_time:.2f} seconds")
```

The master divides the pg table with the array_split function into as many pieces as the workers. (e.g. if we put 5 processes it will split it into 4). It then sends a piece to each worker and waits until it receives the results from each worker which it collects with extend() and prints them.

Part 2 (Worker)

```
else:

    def work(p):
        l1 = p['mlp_layer1']
        l2 = p['mlp_layer2']
        l3 = p['mlp_layer3']
        m = MLPClassifier(hidden_layer_sizes=(l1, l2, l3))
        m.fit(X_train, y_train)
        y_pred = m.predict(X_test)
        ac = accuracy_score(y_pred, y_test)
        return (rank, p, ac)

    tasks = comm.recv(source=0, tag=0)
    worker_results = list(map(work, tasks))
    comm.send(worker_results, dest=0, tag=0)
```

The worker defines the work function which executes and waits for the master to receive the piece with the tasks (an array with pg values). With map(), the work function runs with each value from the tasks panel and sends the results back to the master.

We run the programs for 100,000 samples at a time to better see the improvement with the parallelization. We use a machine with 6 cores.

Serial Code:

```
(0, {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.824939393939394)
(1, {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8236060606060606)
(2, {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8251818181818181)
(3, {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.8237878787878787)
(4, {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.8242121212121212)
(5, {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8246363636363636)
(6, {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8244242424242424)
(7, {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.8247575757575758)
Total execution time: 28.63 seconds
```

Multiprocessing pool 2 processes:

```
(gs_env) max@DESKTOP-548569G:~/hpc$ python multiprocessingPool.py
({'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.8254545454545454)
({'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8243939393939393)
({'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8247272727272728)
({'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.8247575757575758)
({'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.8245454545454546)
({'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8248181818181818)
({'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8248181818181818)
({'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.825)
Total execution time: 15.24 seconds
```

MPI futures 3 processes (2 workers):

```
(gs_env) max@DESKTOP-548569G:~/hpc$ mpirun -n 3 python3 -m mpi4py.futures mpiFutures.py
({'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.8252121212121212)
({'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8251515151515152)
({'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8248787878787879)
({'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.8246969696969697)
({'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.8244242424242424)
({'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8251818181818181)
({'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8253030303030303)
({'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.8247878787878787)
Total execution time: 14.65 seconds
```

Master-Worker 3 processes (2 workers)

```
(gs_env) max@DESKTOP-548569G:~/hpc$ mpirun -n 3 python3 masterWorker.py
(1, {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.8255454545454546)
(1, {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8255454545454546)
(1, {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8247575757575758)
(1, {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.8248484848484848)
(2, {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.8245757575757576)
(2, {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8243030303030303)
(2, {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.825030303030303)
(2, {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.8245757575757576)
Total execution time: 15.31 seconds
```

Increasing the number of processes to 5:

Multiprocessing pool 5 processes:

```
(gs_env) max@DESKTOP-548569G:~/hpc$ python multiprocessingPool.py
({'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.825060606060606)
({'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.825)
({'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8248787878787879)
({'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.8243636363636364)
({'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.8254848484848485)
({'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8246060606060606)
({'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8246060606060606)
({'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.8252121212121212)
Total execution time: 8.88 seconds
```

MPI futures 6 processes (5 workers):

```
(gs_env) max@DESKTOP-548569G:~/hpc$ mpirun -n 6 python3 -m mpi4py.futures mpiFutures.py
({'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.8252121212121212)
({'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8248181818181818)
({'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8243030303030303)
({'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.8247272727272728)
({'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.824969696969697)
({'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8248787878787879)
({'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8242121212121212)
({'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.825)
Total execution time: 9.80 seconds
```

Master-Worker 6 processes (5 workers)

```
(gs_env) max@DESKTOP-548569G:~/hpc$ mpirun -n 6 python3 masterWorker.py
(1, {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.8251212121212121)
(1, {'mlp_layer1': 16, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.8248181818181818)
(2, {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8242424242424242)
(2, {'mlp_layer1': 16, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.8243333333333334)
(3, {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 16}, 0.8245757575757576)
(3, {'mlp_layer1': 32, 'mlp_layer2': 16, 'mlp_layer3': 32}, 0.825030303030303)
(4, {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 16}, 0.8236060606060606)
(5, {'mlp_layer1': 32, 'mlp_layer2': 32, 'mlp_layer3': 32}, 0.825030303030303)
Total execution time: 9.40 seconds
```

All times have a deviation of +- 2 seconds.

Question 3

a)

```
extern double work(int i);
void initialize(double *A, int N)
{
    #pragma omp parallel for schedule(dynamic, 2)
    for (int i = 0; i < N; i++) {
        A[i] = work(i);
    }
}
```

In the above code we use dynamic scheduling with argument 2. So each thread will undertake 2 iterations of the loop (so it will make 2 entries in A) and once it executes them it will take over 2 more and so on. That is, the total number of iterations that each thread will undertake has not been predetermined, but it is dynamically determined during the runtime. In this way the iterations are distributed more fairly, in the that their execution time increases with the increase of i.

b) (Not a screenshot)

```
void initialize ( double  *A, int  N)
{
    #pragma omp parallel
{
        #pragma omp single
{
            for (int i = 0; to < N; i += 2)
{
                #pragma omp task firstprivate(i)
{
                    A[i] = work(i);
                    if (i + 1 < N)
                        A[i+1] = work(i+1);;
}
}
}
}
}
```

The above code is equivalent to that of question a. In particular, we use tasks that are dynamic by nature. That is, they do not have a predefined load during execution, it is also calculated at runtime. So we just have to have each task make 2 entries at a time. We do exactly this (A[i]=work(i), A[i+1]=work(i+1)) and therefore we need to cut the number of iterations in half. This is done by increasing i by 2 at a time. Pragma omp single is needed for a single thread to create the tasks.