Αλγοριθμικές Τεχνικές και Εφαρμογές

Φοιτητης: Παναγιωτης Κοντος

ΑΜ: mpsp2215

Η παρακάτω εργασία αφορά την υλοποίηση ενός αλγορίθμου εύρεσης της μέγιστης τιμής μιας συνάρτησης δύο μεταβλητών με χρήση του προγραμματιστικού μοντέλου MPI. Η εργασία εγινε με την χρήση της **Python**. Αξίζει να σημειωθεί ότι γράψαμε εκ νέου έναν hill climbing αλγόριθμο τον οποίον χρησιμοποιήσαμε για να φτιάξουμε έναν parallel hill climbing αλγόριθμο.

> *Βηματα*

Πιο συγκεκριμένα, αρχικά θα πρέπει να κάνουμε generate έναν 2D array(n x m διαστάσεων) όπου οι αριθμοί κάθε φορά θα είναι τυχαίοι.

```
import numpy as np
import random

def initilize_array(n,m):
    arr = np.random.randint(0,n*m,size=(n,m))
    return arr;
```

Στον οποίο πίνακα θα εφαρμόσουμε ένα αλγόριθμο hill climbing όπου θα περιοριοριζεται σε `συγκεκριμένο αριθμό βημάτων.`

```
def hill_climb(arr, max_iter = MAX_STEPS_EACH_CLIMB):
```

οπου θα δημιουργούμαι εσωτερικά ενα `visited array` απο πολλά 0 και κάθε φορά που θα επισκέπτεται κάποιο σημείο, το σημείο αυτό θα γίνεται 1.

```
 # Create visited array
 visited = np.zeros(arr.shape, dtype=bool)
```

Επιπλέον, το `σημείο εκκίνησης` μας θα ειναι κάθε φορά `τυχαίο`

```
# find rows and columns of the array
rows = len(arr)
cols = len(arr[0])

# generate random starting point
current = [random.randint(0,rows-2),random.randint(0,cols-2)]

# set the initial starting position to visited
visited[current[0]][current[1]] = True
```

Ο αλγόριθμος θα ψάχνει στους κοντινότερους γείτονες για την υψηλότερη κορυφή και κάθε φορά που μετακινείται σε κάποιο γείτονα θα θέτει το σημείο σαν visited μέχρι να μην έχει κορυφή να κινηθεί ή να έχει φτάσει τον max αριθμό βημάτων.

```
if best_neighbor is None:
      return current

if best_neighbor_val <= current_val:
      return current

# Update current and mark as visited
current = best_neighbor

# update the visited table
visited[current[0], current[1]] = True
```

> *Πως μπορεί αυτό να γίνει παράλληλα με την χρήση του MPI?*

Θα δημιουργήσουμε έναν αλγόριθμο parallel_hill_climb που θα χρησιμοποιεί το MPI, το αλγόριθμο για να να κάνει generate έναν 2D array και το αλγόριθμο hill_climbing.

```
def parallel_hill_climbing():
 ...
```

Αρχικα, θα πρεπει να δημιουργηθει ο 2D πινακας και να μοιραστει σε ολα τα processes.

Αυτο θα γινει απο το process με rank 0 και υστερα θα το κανει Broadcast σε ολα τα αλλα.

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

if rank == 0:
     arr = initilize_array(ROWS,COLS)
     # print(arr)
else:
     arr = None


# broadcast array
arr = comm.bcast(arr, root=0)
```

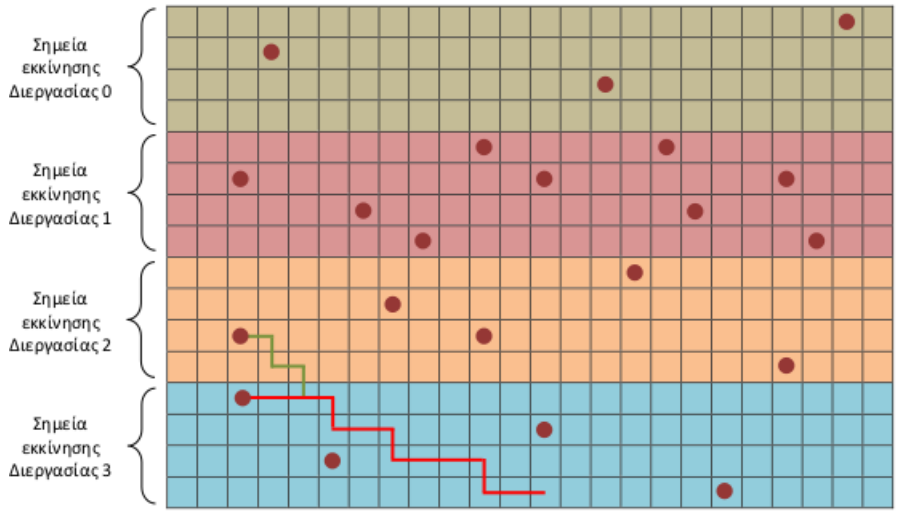
Επειτα θα χωρισουμε τον πινακα σε ισοποσα κομματια ανα γραμμη για ολα τα processes.

```
# Divide array into chunks for each process
process_arr = np.array_split(arr, size)[rank]
```

Δηλαδή αν υποθέσουμε οτι έχουμε έναν πίνακα 4x4 και έχουμε 4 processes. Θα καταλήξουμε με 4 πίνακες 1x4, η κάθε διεργασία θα αναλάβει συγκεκριμένο κομμάτι του πινάκα.

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

[ 0,  1,  2,  3],

[ 4,  5,  6,  7],

[ 8,  9, 10, 11],

[12, 13, 14, 15]
```

Ύστερα, θέλουμε τα restarts του κάθε σημείου να γίνονται τυχαία σε όλο το μήκος του πίνακα αλλά θέλουμε και να διασφαλίζεται οτι κάποιο σημείο δεν θα ακολουθήσει παρόμοια διαδρομή με κάποιο άλλο



Για να συμβεί αυτό θα ξαναχωρίσουμε τον πίνακα αλλά αυτήν την φορά θα είναι ανά στήλη

Ο αριθμός των στηλών θα είναι ο αριθμός των restart που θέλουμε να γίνονται, έτσι το κάθε restart θα έχει 1 τετράγωνο υπό ευθύνη του και θα δημιουργούνται σύνορα τα οποία δεν μπορεί να περάσει.

Στην ουσία ο πίνακας διαιρείται σε κομμάτια ανά γραμμή και μετά αυτά τα κομμάτια διαιρούνται ανά στήλη

```
    # split in n grids by column
    local_arr = np.hsplit(process_arr, RESTARTS)
```

```
[ 0,  1,  2,  3,  5,  9,  2,  23],

[0,  1]    [2,  3]  [5,  9]  [2,  23]


. . .
```

Και το κάθε restart αφού τρέξει τον αρχικό hill_climbing αλγόριθμο θα βρίσκει ένα τοπικό μέγιστο, το οποίο θα συγκρίνει με τα τοπικά μέγιστα απο τα άλλα restarts για να βγάλει το μέγιστο όλου του process.

```
 # Compute max of local array with hill climbing
        local_max = hill_climb(local_arr[i],MAX_STEPS_EACH_CLIMB)

# get max value
        local_max = local_arr[i][local_max[0],local_max[1]]

# get the largest from all splits
        if(local_max>process_max):
            process_max=local_max
        i+=1
```

αφού βγουν τα `process_max` θα γίνονται reduce από όλα τα processes στο rank 0 το οποίο θα καταλήγει με το εκτιμώμενο `global_max`

```
# Reduce process max to get global max
global_max = comm.reduce(process_max, op=MPI.MAX, root=0)


 if rank == 0:
        print("Global max:", global_max)

MPI.Finalize()
```

> *Προβλήματα που αντιμετωπίστηκαν*

Πως θα καταλάβει ο αλγόριθμος 1 σημείο αν έχει εξεταστεί?

- Με την εφαρμογή του visited array στον αλγόριθμο hill climbing που θα χρησιμοποιείται από όλα
- Με το να χωρίσουμε ανά στήλη τον πίνακα για κάθε restart τα σημεία δεν θα έχουν πρόσβαση στον πίνακα του κάθε restart

Πως θα χειριστεί ο αλγόριθμος το πρόβλημα του Load Balancing?

- Ο αλγόριθμος hill climbing έχει max αριθμό βημάτων που μπορεί να είναι πχ 4-5 επομένως όλες οι διεργασίες θα έχουν παρόμοιο αριθμό βημάτων που μπορούν να κάνουν

- Επιπλέον, δεν γίνεται να δημιουργηθούν οι ίδιοι τυχαίοι αριθμοί διότι η κάθε διεργασία θα αναλαμβάνει 1 συγκεκριμένο κομμάτι του πίνακα και δεν θα έχει πρόσβαση στα άλλα
- Ούτε γίνεται να κάνει restart στο ίδιο τυχαίο σημείο διότι το κάθε restart έχει δικό του πινάκα

- Γίνεται ο ίδιος αριθμός restart για ολες τις διεργασίες. πχ 5.

> *Μετρήσεις και Αποτελέσματα*

Παρακάτω παρουσιάζονται τα αποτελέσματα στον αλγόριθμο parallel_hill_climbing για διάφορα **μεγέθη πινάκων** και **διεργασιών** και **restarts.** Επίσης, κάθε φορά παρουσιάζεται ο χρόνος εκτέλεσης σε **seconds**,η απόσταση που είχε το εκτιμώμενο ολικό μέγιστο από το πραγματικό ολικό μέγιστο και η εκτίμηση του για το ολικό μέγιστο.

margin = αποσταση απο το πραγματικο ολικο μεγιστο.

- *Για πίνακες 10x10 και 100x100 τα αποτελέσματα ειναι τα παρακάτω.*

RESTARTS=5 N=10			
Processors	Time	Margin	Global Estimate
1	0.0003	0	99
2	0.0006	0	99
4	0.0008	0	99
5	0.0006	0	99

RESTARTS=5 N=100			
Processors	Time	Margin	Global Estimate
1	0.0005	66	9933
2	0.0009	71	9928
4	0.0017	26	9973
8	0.0008	0	9999

- *Για πίνακες 1000x1000 και 10000x10000 τα αποτελέσματα ειναι τα εξης:*

RESTARTS=5 N=1000 Processors	Time	Margin	Global Estimate
1	0.01	9710	990289
2	0.011	7030	992969
4	0.015	2513	997486
8	0.028	469	999530

RESTARTS=5 N=10000 Processors	Time	Margin	Global Estimate
1	0.59	2260940	97739059
2	0.89	1228155	98771844
4	1.37	1583831	98416168
8	40	56786	99943213

Συμπερασματικά, ο αλγόριθμος φαίνεται να λειτουργεί αρκετά γρήγορα και με ακρίβεια για πινάκες μεγέθους 10x10, 100x100. Όταν όμως αυξήσουμε το μέγεθος στα 1000 η απόδοση του παραμένει καλή άλλα για να αυξηθεί η ακρίβεια του πρέπει να αυξήσουμε τον αριθμό των διεργασιών. Αντίθετα, όταν αυξήσουμε σε μεγέθη 10000x10000 εκεί για να έχουμε ακρίβεια πρέπει να θυσιάσουμε σημαντικά την ταχύτητα του αλγορίθμου. Παρόλα, αυτά μέχρι τα 10000 φαίνεται να δίνει ένα κάλο estimate του μεγίστου σημείου και σε ικανοποιητικό χρόνο