פתרון מבחן מועד Y מחשוב מקבילי ומבוזר סמסטר 2022 א

שאלה מספר 1 (סה"כ 30 נקודות)

לפניך שלושה קטעי קוד המבוססים על שימוש ב-MPI, OpenMP, CUDA עם בעיות מימוש כמו למשל deadlock, אי התאמה בין תהליכים ועוד. עבור כל אחד מהסעיפים יש להבין את הבעיות ולתאר את הדרך לפתור אותן.

בכל המקרים החלק של הקוד שכתוב בשפת C לא כולל שגיאות קומפילציה.

שאלה מספר 1.1 (MPI)

יש שלושה תהליכים. כל אחד מהם מחזיק מספר שלם (במשתנה my_num). המטרה של הקוד היא לחבר את שלושת המספרים ואת הסכום לאחסן בכל אחד משלושת התהליכים במשתנה sum.

```
void main(int argc, char *argv[])
{
    int my_rank;
    MPI_INIT(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    int my_num;  // assume this is initialized
                 // in all processes
    MPI_Bcast(&my_num, 1, MPI_INT, my_rank, MPI_COMM_WORLD);

    int other_num1, other_num2;
    MPI_Recv(&other_num1, 1, MPI_INT, MPI_ANY_SOURCE, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    MPI_Recv(&other_num2, 1, MPI_INT, MPI_ANY_SOURCE, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   int sum = my_num + other_num1 + other_num2;
}
```

רשמו כאן את הבעיות שיש בקוד ואיך לפתור אותן

אין צורך לכתוב קטע קוד שלם שעובד – מספיק לציין בקצרה איך ניתן לתקן את הקוד.

פתרון

הקוד הנתון יוצר רושם שגוי כאילו כל אחד משלושת התהליכים עושה broadcast כדי להפיץ את המספר שלו לשני התהליכים האחרים. וכל תהליך עושה פעמיים MPI_Recv כדי לקבל את המספרים שהפיצו שני התהליכים האחרים.

אבל זה לא עובד ככה ב- MPI: קבלת הודעה שמופצת ע"י MPI_Bcast לא נעשית ע"י קריאה ל- MPI_Recv. במקום זה כדי לבצע broadcast, כל התהליכים (גם התהליך שמפיץ את ההודעה וגם התהליכים שיקבלו אותה) צריכים לקרוא ל- MPI_Bcast כאשר כולם מעבירים את אותו "root" כארגומנט (ה- "root" הוא (ה- rank של) התהליך שמפיץ את ההודעה לאחרים).

שימו לב שבקוד הנתון כל תהליך מעביר את my_rank כארגומנט שמציין את ה- "root" בקריאה ל- MPI_Bcast כלומר כל אחד מעביר root אחר. זו שגיאה כי כל התהליכים צריכים לציין את אותו ה- root.

פתרון: אפשר לעשות שלושה broadcast כך שכל תהליך יפיץ את המספר שלו לאחרים.

(כל תהליך יקרא שלוש פעמים ל- MPI_Bcast). לחילופין אפשר לדאוג שכל תהליך יעביר לשניים האחרים את המספר שלו ע"י קריאות ל- MPI_Send ו- MPI_Recv. (אפשר גם להשתמש ב- MPI_Sendrecv).

שאלה מספר 1.2 (OpenMP)

נתון מערך A של מספרים. המטרה היא לכתוב לפלט את מספר הזוגות של מספרים שהם סמוכים זה לזה וזהים. לדוגמא אם איברי המערך הם

8 8 8 7 2 2 7 10 10 5 3 אז מספר הזוגות שמקיימים את התנאים הוא 4

(זוג של 10, זוג של 2 ושני זוגות של 8 (ב- 8 8 8 יש 2 זוגות: 8 8 הראשונים ו- 8 8 האחרונים).

הנה הקוד:

```
#define N 10000
int A[N]; // assume A is initialized
int counter = 0;
#pragma omp parallel default(none)
```

```
{   int i = 0;
    while (i < N-1){
        if (A[i] == A[i+1])
            counter++;
        i++;
    }
}
printf("count is %d\n", counter);
```

<u>רשמו כאן גרסה נכונה של הקוד</u>

<u>פתרון</u>

כפי שזה רשום, כל thread יעשה את כל האיטרציות של לולאת ה- while בעוד שהכוונה היא שעבודת ביצוע האיטרציות תחולק בין ה- threads השונים.

OpenMP לא תומכת במיקבול של לולאות while. לכן יש להשתמש בלולאת for אותה ניתן למקבל. בעיה נוספת: בגלל שרשום default(none) יש לציין במפורש אם A ו- counter הם shared או private.

ההשמה ל- counter (אם הכוונה היא שיהיה shared) יוצרת race condition.

הנה גרסה נכונה של הקוד:

```
#define N 10000
int A[N]; // assume A is initialized

int counter = 0;
#pragma omp parallel for default(none) shared(A) \
                          reduction(+:counter)
  for(int i = 0; i < N-1; i++)
      if (A[i] == A[i+1])
            counter++;
```

```
printf("count is %d\n", counter);
```

שאלה 1.3 (Cuda)

המטרה היא להכפיל בשלוש את כל האיברים של וקטור נתון.

הנה ה- kernel:

```
__global __ void multiply3(int *A)
{
    A[threadIdx.x] *=3;
}
```

הנה ה- main:

```
#define N  1100
int main(int argc, char *argv)
{
    int A[N];
    // assume A is initialized here ... */

    multiply3<<<N/256, 256>>>(A);

    for(int i = 0; i < N; i++)
        printf("A[%d] is %d\n", i, A[i]);

    return 0;
}
```

רשמו כאן את הבעיות שיש בקוד ואיך לפתור אותן

כאן מספר ה- threads אמור להיות לפחות N (גודל המערך) כי כל thread מכפיל איבר בודד של המערך. כדי להבטיח את זה צריך לעגל כלפי מעלה את N/256 כשמחשבים את מספר הבלוקים. כלומר במקום N/256 צריך להיות (N+255)/256.

צריך לוודא שאף thread לא חורג מגבולות המערך. לכן ה- kernel צריך להיות מוגדר כך (הארגומנט size מציין את גודל המערך):

```
__global __ void multiply3(int *A, unsigned int size)
{
    if (threadIdx.x < size)
        A[threadIdx.x] *=3;
}
```

בעיה נוספת: בקוד הנתון בשאלה, הקריאה ל- kernel ב- main מעבירה כתובת במרחב הכתובות של ה- host כארגומנט. במקום זה צריך להעביר כתובת של העתק של המערך A שנמצא על ה- device.

לפני הקריאה ל- kernel צריך להקצות מקום בזיכרון של ה- device ולהעתיק לשם את המערך A. ואז להעביר כארגומנט ל- kernel את הכתובת של ההעתק. לבסוף, צריך להעתיק בחזרה ל- host את המערך שעודכן ע"י ה- kernel ולשחרר את הזיכרון שהוקצה על ה- device.

הנה הקוד המתוקן (הושמטו בדיקות האם יש שגיאות):

```
int *dev_A;

cudaMalloc((void **)&dev_A, N*sizeof(int));
cudaMemcpy(dev_A, A, N*sizeof(int), cudaMemcpyHostToDevice);

multiply3<<<(N+255)/256, 256>>>(dev_A, N);

cudaMemcpy(A, dev_A, N*sizeof(int), cudaMemcpyDeviceToHost);

cudaFree(dev_A);
```

<u>שאלה מספר 2 (בסה"כ 70 נקודות)</u>

כתבו בעזרת MPI ו- OpenMP תכנית הבודקת אם מערך A של מספר שלמים הוא סדרת פיבונאצ'י כלומר כל איבר במערך (החל מהאיבר השלישי) שווה לסכום של שני האיברים הקודמים. לדוגמא זו סדרת פיבונאצ'י: 2, 7, 9, 16, 25, 41

התכנית אמורה לעבוד עם מספר שאינו ידוע מראש של תהליכי MPI.

בהתחלה איברי המערך A ידועים רק לתהליך עם rank אפס. (אין צורך לאתחל את המערך). התהליך עם rank אפס הוא שיכתוב בסופו של דבר "yes" לפלט אם איברי המערך מהווים סדרת פיבונאצ'י ואחרת הוא יכתוב "no".

ניתן להניח שמספר התהליכים p מתחלק ב- N, מספרי האיברים ב- A.

סעיף א: רשמו פתרון בפסאודו קוד

סעיף ב: רשמו את הפתרון בקוד תוך שימוש ב- MPI וב- OpenMP.

<u>פתרון</u>

התהליך עם rank אפס (ה- "master") יפזר את איברי A בין כל התהליכים כך שכל תהליך יקבל חלק רציף של המערך. כל תהליך יבדוק אם החלק שקיבל הוא סידרת פיבונאצ'י ויחשב את התוצאה: true (כן סדרת פיבונאצ'י) או false (לא סידרת סדרת פיבונאצ'י).

ה- AND של כל התוצאות יחושב והתוצאה תשמר אצל ה- master.

(אפשר לעשות את זה ע"י reduce של התוצאות הנ"ל (שכל אחת מהן היא true או false) עם האופרטור AND או reduce עם SUM (ואז כל תוצאה תהיה 1 או 0) ולבדוק שהסכום שווה למספר התהליכים).

כל אחד מהתהליכים בודק את כל איברי חלק המערך שקיבל מלבד השניים הראשונים

כי כדי לבדוק את אלו הוא צריך שני איברים קודמים אותם הוא לא קיבל.

לכן ה- master יבדוק אם האיברים שלא נבדקו ע"י שאר התהליכים עומדים בתנאי של

סדרת פיבונאצ'י. אם כן ובנוסף לכך התוצאה של ה- AND הנ"ל היא true אז ה- master יכתוב yes ואחרת יכתוב no.

פסאודו קוד:

```
if (myrank == 0) {
    send  N/p  (contiguous) elements to each of the
          other processes.
} else
      receive N/P elements into subarray my_A
// all processes do this:
```

```
check if my part of the array is a Fibonacci
sequence:
isFibonacci = true
parallel loop:
for (i = 2; i < N/p; i++)
    if (my_A[i] !=  my_A[i-1] + my_A[i-2])
       isFibonacci = false

Reduce all isFibonacci values using operator AND
and store the result in process with rank 0 (in
variable isFibonacci)

if (myrank == 0)  { // this is done sequentially
    check if first and second elements in all
    subarrays fulfill the Fibonacci condition.
    if not then set isFibonacci = false
if (isFibonacci)
    print("yes");
else
    print("no")
 } // myrank == 0
```

<u>הנה הקוד</u>

```
int main(int argc, char **argv) {
   int myid, nprocs;
   int *A; // the input array. assume it is initialized
   int *my_A; // subarray of A
   int N; // size of the input array. assume it is initialized
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (myid == 0) {
       if ((N % nprocs) != 0) {
          fprintf(stderr, "size of array not divisable by number of
processes\n");
          MPI_Abort(MPI_COMM_WORLD, 2);
       }
    }
    /* The root process first broadcasts chunkSize, the size of the subarray
          of A each process will check.
       This enables the processes to allocate memory for the subarray.
       And then the root scatters the array itself.
    */
   int chunkSize = N/nprocs;
   MPI_Bcast(&chunkSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
   my_A = (int *) Malloc(sizeof(int)*chunkSize);

   MPI_Scatter(A, chunkSize, MPI_INT,
                  my_A, chunkSize, MPI_INT, 0, MPI_COMM_WORLD);

    int result = isFibonacci(my_A, chunkSize);

    int sum_result; // used by process with rank 0
        // we need a logical AND (MPI_LAND) of all results but we can use
        // MPI_SUM  which is what we learned in class
    MPI_Reduce(&result, &sum_result, 1, MPI_INT,
                     MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) {
```

```
        int isFibonacci = (sum_result == nprocs); // all proccess
                                                  // found it is a fibonacci series
        if (!isFibonacci)
            printf("no\n");
        else {
            // check boundary elements:
           for (int rank = 1; rank < nprocs; rank++) {
              // process 'rank' could not check first 2 numbers
              // in its subarray because it didn't know what the
              // previous numbers were so the master does the check here
             int start_index = rank * (chunkSize);
             if (A[start_index] != A[start_index-1]+A[start_index-2]) {
                 isFibonacci = 0;
               break;
             }
            if (A[start_index+1] != A[start_index]+A[start_index-1]) {
              isFibonacci = 0;
             break;
            }
         }
         printf("%s\n", isFibonacci ? "yes" : "no");
        }
    } // myid == 0

    MPI_Finalize();
    return 0;
}
```

```
int isFibonacci(int *array, unsigned int size)
{
    for (int i = 2; i < size; i++)
        if (array[i] != array[i-1]+ array[i-2])
            return 0;
     return 1;
}
```