Homework 3 Dry

מגישים :
דמיטרי קולטונוב 320645401
עדי מזוז 305769036

חלק ראשון: זיהוי כשלי סנכרון

תזכורת : תכונות הקטע הקריטי

תכונות הכרחיות:

1. **Mutual Exclusion – מניעה הדדית** – בכל רגע נתון, לא יכול להיות יותר מחוט אחד בתוך הקטע הקריטי (הדבר שקול לכך שהקטע הקריטי הופך לנקודת **סריאליזציה** במסלולי הביצוע).
2. **Progress – התקדמות** - אם יש חוטים שרוצים לבצע את הקטע הקריטי, לבסוף חוט **כלשהו** יצליח להיכנס. ישנה התקדמות – אין Deadlock/Livelock.

תכונות רצויות:

3. **Fairness – הוגנות** – אם יש חוט **שרוצה** לבצע את הקטע הקריטי, **הוא** לבסוף יצליח. אין הרעבה.
הרחבות על הדרישה :
   - **Bounded Waiting** – הגדרת **חסם** למספר הפעמים שחוטים אחרים ייכנסו לקטע הקריטי לפני החוט הנוכחי.
   - **Order** – יש סדר ברור וידוע לזמני הכניסה של החוטים הנכנסים לקטע הקריטי. דוגמה לסדר אפשרי : FIFO.

1.

א. אילו תכונות של הקטע הקריטי מפר המימוש הבא כאשר משתמשים בו במערכת עם נפילות חוטים, ולמה? הניחו שהקוד רץ על מעבד יחיד.
נפילת חוטים: חוט יכול ליפול באופן פתאומי, כתוצאה מחריגה למשל.

ב. במימוש קיימת בעיית Performance, הגורמת לחוסר יעילות של זמן המעבד. ניתן להניח שהקטע הקריטי עליו המנעול מגן הינו קטע ארוך וכבד חישובית. היכן היא? האם הבעיה עדיין קימת אם הקטע היה קצר ומהיר?

```
class lock {
        bool lockVal;
public:
        lock(bool initVal) { lockVal = initVal;}
        void lock(){
                while(AtomicCompareAndSwap(&lockVal,0)==0){}
        }
        void unlock(){
                lockVal = 1;
        }
```

1א. Progress: נשים לב שאם יש נפילת חוטים, עלול לקרות מצב, שחוט נכנס לקטע הקריטי, נעל את המנעול ואז נפל בתוך הקטע הקריטי, כעת המנעול נעול ואין מי שיפתח אותו, כלומר אף אחד לא יצליח להכנס לקטע הקריטי ולהתקדם.
Fariness: אין אצלנו שום מעקב על מי מחכה לחכות ומתי הוא התחי לחכות, ולכן אין דרך לדעת מי יכנס ומתי, ועלול לקרות מצב שחוט יחכה המון זמן (הרעבה) ולא יתקדם כי תמיד חוט אחר "יעקוף" אותו בתור.

1ב. כאשר חוטים מחכים להכנס לקטע הקריטי הם בלולאת while מבצעים busy-wait כלומר הם "מבזבזים" זמן על המעבד רק על לחכות, גם אם הקטע הקריטי היה קצר, יכול להיות שיש הרבה חוטים שמחכים להכנס לקטע והם יבזבזו המון זמן בלולאה

2. ממשים מנעול חדש שעובד כדלקמן. בזמן ניסיון נעילה, המנעול תומך ב-Timeout אותו ממשים על ידי מונה בצורה הבאה: במידה והחוטים במערכת מנסים לתפוס את המנעול MAX_ITER פעמים, אך המנעול אינו שוחרר במהלך ניסיונות אלו, המנעול ישוחרר. שימו לב ש-MAX_ITER הינו define גלובלי הידוע לכל החוטים. תיאור בפסודו קוד של מימוש ה-Timeout נתון בקטע הקוד הבא:

```
while( mutex is locked ) {
        if (mutex wasn't released yet)
                count++;
        else
                count =0;
        if( cnt == MAX_ITER)
                unlock mutex
}
```

הניחו מערכת עם מעבד יחיד ואפשרות לנפילת חוטים פתאומית. הניחו שה-Mutex מומש **<u>בעזרת תור</u>** ושומר על סדר הכניסות אליו (FIFO). זהו אגב, נקרא מנעול "הוגן". אילו תכונות של הקטע הקריטי מופרות פה?

כעת יש בעיה של mutual excultion: במידה וקטע הקריטי הינו ארוך, חוט שמחכה הרבה זמן על המנעול יכול פשוט לפתוח את המענול ולהכנס, ואז יהיו 2 חוטים בתוך הקטע הקריטי באותו הזמן

3. בהנחה שהקוד מורץ על מעבד יחיד, הסבר מה ידפיס הקוד הבא, ולמה ?

```
int sum=0;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
int ids[10]={1,2,3,4,5,6,7,8,9,10};

void* thread_workload(void *threadID){
          int* p_val = (int*) threadID;
          pthread_mutex_lock(&mutex);
          sum += *p_val;
          pthread_mutex_unlock(&mutex);
}
```

```
int main(){
        pthread_t t;
        int i;
        for(i=0;i<10;++i)
                pthread_create(&t,NULL, thread_workload,(void*)(ids+i));
        pthread_join(t,(void**(&i));
        printf("%d\n", sum);
        return 0;
}
```

נשים לב שבmain יש לולאה שיוצרת 10 חוטים ושומרת את המזהה שלהם באותו משתנה t לאחר יצירת החוטים אנו מחכים לחוט t, שהוא יהיה חוט מספר 10 , אך למרות זאת אין הבטחה שחוט 10 יהיה האחרון שסיים לרוץ בעקבות החלפת הקשר, ולכן מה שודפס יהיה סכום החוטים שסיימו לרוץ לפני חוט 10 , + 10 (של החוט האחרון)

4. בהנחה שהקוד מורץ על מעבד יחיד, הסבר מה ידפיס הקוד הבא, ולמה ? התשובה צריכה להיות מורכבת מערך מקסימלי אפשרי וערך מינימלי אפשרי, עם תרחיש אפשרי לכל אחד. ניתן להניח שפעולות store ו-load מתבצעות באופן אטומי (זוהי הנחה בקורס בנוגע לכל פקודות האסמבלר למיניהן)

```
int result;
void* do_calc();
        int i;
        for(i=0; i<100 ; ++i)
                result=result+1;
int main(){
        pthread_t threads[2];
        int i;
        result =0;
        for(i=0;i<2;++i)
                pthread_create(&threads[i],NULL,do_calc,NULL);
        for(i=0;i<2;++i)
                pthread_join(threads[i],NULL);
        printf("%d\n", result); return 0;
}
```

המצב המינמלי יתקבל כאשר חוט אחד יכנס יטען את המשתנה הגלובלי result=0 ואז תתבצע החלפת הקשר לפני שנעלה את result ב1, כעת תתחיל הלולאה של החוט השני, והיא תתבצע עד הסוף . כלומר הוא יעדכן את המשתנה ל100 , ואז תתבצע שוב החלפת הקשר, נזכור שבחוט החוט הראשון המשתנה שמור על 0, ולכן כאשר נבצע את השורה Result=result+1 בעצם נכניס למשתנה הגלובלי 1. ואז נמשיך עד סוף הלולאה ונקבל את הערך המינימלי של 100

במצב המקסימלי החוטים לא יפריעו אחד לשני ונקבל 200 בתוצאה.

5. הסבירו למה אין צורך להגן על sum בעזרת משתנה סנכרון כמו Mutex או Semaphore.

```
int sum = 0;

if(fork()) {
    sum = sum + 5;
```

```
} else {
    sum = sum + 1;
}
```

כיוון כשאנו עושים fork אנו מעתיקים את כל הזכרון של האב ויוצרים שכפול שלו בתהליך הבן כלומר הבן והאב ניגשים למשתנים שונים, ולא לאותו sum.

חלק שני: Singlephore

לרוב מנגנוני הסנכרון עליהם למדתם, קיימים לפחות שתי פעולות. מנעולים פשוטים תומכים ב-lock ו-unlock. משתני תנאי תומכים ב-wait ו-signal, וסמפורים ב-up ו-down או בשמם המקורי בספרות, P ו-V.
בתרגיל זה תעבדו עם מנגנון סנכרון שלו **תמיכה רק בפעולה אחת ויחידה**, ונקרא – **singlephore**.

הגדרת פעולות של המנגנון:

```
typedef struct singlephore {
    int value;
} singlephore;

// Initialize the singlephore to value 0.
void singlephore_init(singlephore * h) {
    h->value = 0;
}

// Block until the singlephore has value >= bound, then atomically increment its value by
delta.
void H(singlephore * h, int bound, int delta) {
// This is pseudocode; a real singlephore implementation would block, not
// spin, and would ensure that the test and the increment happen in one
// atomic step.
    while (h->value < bound) {
        sched_yield();
    }
    h->value += delta;
}
```

ברגע שה-singlephore אותחל, קוד אפליקציה יגש אליו רק דרך הפעולה H.

א. ממש מנעול למניעה הדדית בעזרת singlephore. מלא את תבניות הקוד הבאות:

```
typedef struct mutex {
    singlephore h;
} mutex;

void mutex_init(mutex* m) {
    singlephore_init(&m->h);
}

void mutex_lock(mutex* m) {
    H(&m->h, 0,-1);
}

void mutex_unlock(mutex* m) {
    H(&m->h, 0,1);
}
```

ב. **<u>סעיף בונוס</u> (7 נקודות)**: ממש משתנה תנאי בעזרת singlephore ו-mutex (שכבר מימשתם). מלא את תבניות הקוד הבאות: (שימו לב, הסעיף הבא **<u>אינו</u>** סעיף בונוס, אך יכול לעזור לפתרון סעיף זה).

```
typedef struct condvar {
    mutex m;
    singlephore h;
    int counter;
} condvar;

// Initilize the condition variable
void cond_init(condvar* c) {
   mutex_init(&c->m);
   singlephore_init(&c->h);
    H(&c->h, INT_MIN, -1);

   c->counter=0;
}

// Signal the condition variable
void cond_signal(condvar* c) {
```

```
 if(counter==0) return;
    H(&c->h, INT_MIN, 1);

}
// Block until the condition variable is signaled. The mutex m must be locked by the
// current thread. It is unlocked before the wait begins and re-locked after the wait
// ends. There are no sleep-wakeup race conditions: if thread 1 has m locked and
// executes cond_wait(c,m), no other thread is waiting on c, and thread 2 executes
// mutex_lock(m); cond_signal(c); mutex_unlock(m), then thread 1 will always recieve the
// signal (i.e., wake up).





void cond_wait(condvar* c, mutex* m) {
    mutex_unlock(m);
    mutex_lock(c->m);
    c->counter++;
    mutex_unlock(c->m);
    H(&c->h, 0, -1);
    mutex_lock(c->m);
    C->counter--;
    mutex_unlock(c->m);
    mutex_lock(m);
}
```

בהסתמך על שיפור הפתרון בסעיף הבא: הוספנו counter שיספור כמה ממתינים כרגע על הסיגנל, יש mutex פנימי שמגן על העדכון שלו, אם אין חוטים ממתינים signal לא עושה דבר, בנוסף הורדנו באתחול את הsinglaphor ב1, כך שגם בפעם הראשונה החוט יחכה לsignal

רמזים:

1. אם אין חוט שמחכה על משתנה התנאי c, אז cond_signal(c) לא יעשה דבר.
2. הנח ש-N חוטים ממתינים על משתנה התנאי c. אז N קריאות ל- cond_signal(c) הם תנאי הכרחי ומספיק על מנת להעיר את כולם.
3. יתכן ותוכל להיעזר בסעיף הבא כדי למצוא את הפתרון הנכון
4. ניתן ורצוי להשתמש בקבוע INT_MIN, הערך הנמוך ביותר ש-integer יכול לקבל.

ג. ירמיהו החרוץ מתלמידי הקורס, סיפק את הפתרון הבא לסעיף ב':

```
typedef struct condvar {
    singlephore h;
} condvar;

void cond_init(condvar* c) {
    singlephore_init(&c->h);
```

```
}

void cond_signal(condvar* c) {
    H(&c->h, INT_MIN, 1);
}

void cond_wait(condvar* c, mutex* m) {
    mutex_unlock(m);
    H(&c->h, 0, -1);
    mutex_lock(m);
}
```

מה לא תקין בפתרון? הראו תרחיש אפשרי בו פתרון זה לא עומד בתנאים של סעיף ב'.

בפעם הראשונה שנעשה cond_wait החוט לא יחכה כלל עבור cond_signal כיוון ש value=0=bound
בנוסף אם נעשה cond_signal כאשר אין חוטים בהמתנה אנו נשנה את ערך הvalue של הsinglephore וזה יגרום
לכך שעוד חוטים לא יחכו כאשר אנחנו נעשה cond_wait וימשיכו ישר (כיוון שאנחנו מעלים את value מעל 0)

חלק שלישי: ניתוח של החלק הרטוב

חלק זה מבוסס על חלקו הרטוב של תרגיל בית 3, ומיועד לפתרון לאחר סיום חלק זה. במידה והסתבכתם, ניתן גם להיעזר בחלק זה לשם פתרון החלק הרטוב.

1. פיראס החרוץ מתלמידי הקורס ביסס מנגנון סנכרון בין Producer-Consumer שלו הוא קרא "Barrier":

```
class Barrier {
private:
        int working;
public:
        Barrier(){
                working =0;
        }
        increase(){
                working++;
        decrease(){
                working--;
        }
        wait(){
                while(working!=0){}
        }
} ;
```

השימוש במנגנון היה כדלקמן:

Consumer (One of N)
while(1)
    job j = p.pop() // *blocked here if queue is empty*
    execute j
    b.decrease();

Producer:

1. Init Barrier b
2. Init PCQueue p
3. Init fields $curr, next$
4. for t=0 $\rightarrow$ t=n_generations

```
    for i = 0 → i = M
        p.push(job);
        b.increase();
    b.wait();
    swap(curr, next);
```

הנחות:

1. חלק מהפסודו קוד שניתן לכם במסגרת התרגיל הרטוב הושמט. השאלה מתייחסת רק למנגנון הסנכרון.
2. התור מעלה הינו אותו תור יצרן-צרכן שהתבקשתם לממש בתרגיל הרטוב.
3. job הינו struct אשר מתאר לחוט כלשהו את גבולות הגזרה עליהם עליו לרוץ.
4. M הינו מספר העבודות הכולל שיש לחשב ב-t generation.

א. הסבירו את כוונותיו של פיראס – איך היה אמור המנגנון לעבוד?

הכוונה של פירוס היא שהbarrier יחכה עד סיום עבודת כל החוטים לפני שנמשיך ל generation הבא

ב. במימוש זה מספר בעיות Correctness.

a. מצאו בעיה אחת של Race Condition בפתרון. הסבירו.

שינוי המונה של העובדים (int working) לא מוגן לכן יתכן ששני חוטים יגשו לעדכן את השדה וכתואה נקבל ערך שונה בworking מאשר שהחוטים היו ניגשים לשדה בנפרד.

b. תארו תרחיש שבו מופר ה-Mutual Exclusion. דהיינו, חישוב הלוח curr טרם הסתיים, וה-Producer מבצע למרות זאת את ה-swap של הלוחות.

הכנסנו עבודה והגדנו את הworking ב1 מיד אחר כך התחיל החוט לבצע את העבודה וסיים אותה (לפני שנכנסה עוד עבודה)

בחוט היוצר (producer) נכניס עבודה אחת ונעשה incrice מיד החוט יתחיל לטפל בעבודה וכשיגיע לחלק של הdecrice הוא ישלוף את המספר ששמור ב working (1) כידי לעדכנו אבל אז תתבצעה החלפת הקשר והחוט הראשי ימשיך להכניס את כל שאר המשימות ויגיע לwait כעט נחזור לחוט הראשון כזכור שמר את working ב1 ונשיך את הdecrice שלו ככה שהworking יתעדכן ל0 ונעבור את הwait למרות שישנם חוטים שעוד לא סיימו את עבודתם.

c. תארו שני תרחישים שונים בהם יתכן Deadlock בפתרון.

1. החוטים מקבלים את המשימות שלהן, החוט הראשון שמסיים בא לעדכן את השדה working שומר אצלו את הערך שאותו צריך לעדכן (N כל החוטים) ואז מתבצעה החלפת הקשר ושאר החוטים רצים מסיימים את עבודתם כל אחד בתורו מעדכן את working ואז חוזר החוט שהובקע קודם שלו יש את working=N הוא מעדכן את השדה ל N-1 ואנו נשארים בdeadlock אין מי שיוריד את הערך של working .

2. כאשר החוט יוצר ניגש לעשות increce שומר אצלו את ערך הworking ואז אחד החוטים שסיים את עבודתו ניגש גם כן לעשות decrese ולוקח את השליטה על המעבד ומעדכן את working בהצלחה, ואז החוט היוצר חוזר למעבד וממשיך בincrese אך כעט יש אצלו ערך לא תקים של working ואותו הוא יעדכן וישמור. לכן מעבדים את המספור וכשניהי בwait לא נגיע לworking=0.

ג. תקנו את class Barrier ואת הפסודו קוד של היצרן-צרכן כך שכל בעיות ה-Correctness יפתרו. אין לשנות את מתווה הפתרון של פיראס באופן מהותי ואין להשתמש בפעולות אטומיות.

```
class Barrier {
private:
        int working;
        mutex m;
public:
        Barrier(){
                working =0;
        }
        increase(){
                m.lock();
                working++;
                m.unlock();
        decrease(){
                m.lock();
                working--;
                m.unlock();
        }
        wait(){
                while(working!=0){}
        }
} ;
```

הוספנו הגנה על השדה working כך שכעט הפסדו קוד של צרכן יצרן יעבוד טוב.

ד. בהתייחסות לסעיף ג: מנגנון סנכרון זה נקרא "מונה משותף", ואינו מוצלח במיוחד ממבט של בביצועים – Performance. הסבר מדוע.
במימוש זה יש busy-wait לכן הרבה זמן מעבד ילך לשווא.

ה. **בונוס (7 נקודות):** בנו Barrier אחר, המבוסס על שני מערכים בינריים של באורך N, הפותרים יחדיו את בעיית הסנכרון. על המימוש **לא** להשתמש במנעולים מכל סוג, אך יכול לנצל את הפעולה **האטומית** CompareAndSwap, הנתונה לכם מטה. הניחו שפעולה זו ממומשת **בחומרה**, ולא בתוכנה, כיאה לפעולות אטומיות אחרות. הסבירו למה פתרון זה עדיף מבחינת Performance.

```
int CAS(int *ptr,int oldvalue,int newvalue)
{
   int temp = *ptr;
   if(*ptr == oldvalue)
       *ptr = newvalue
   return temp;
}
```

2. **ביצועים וחוק אמדל**

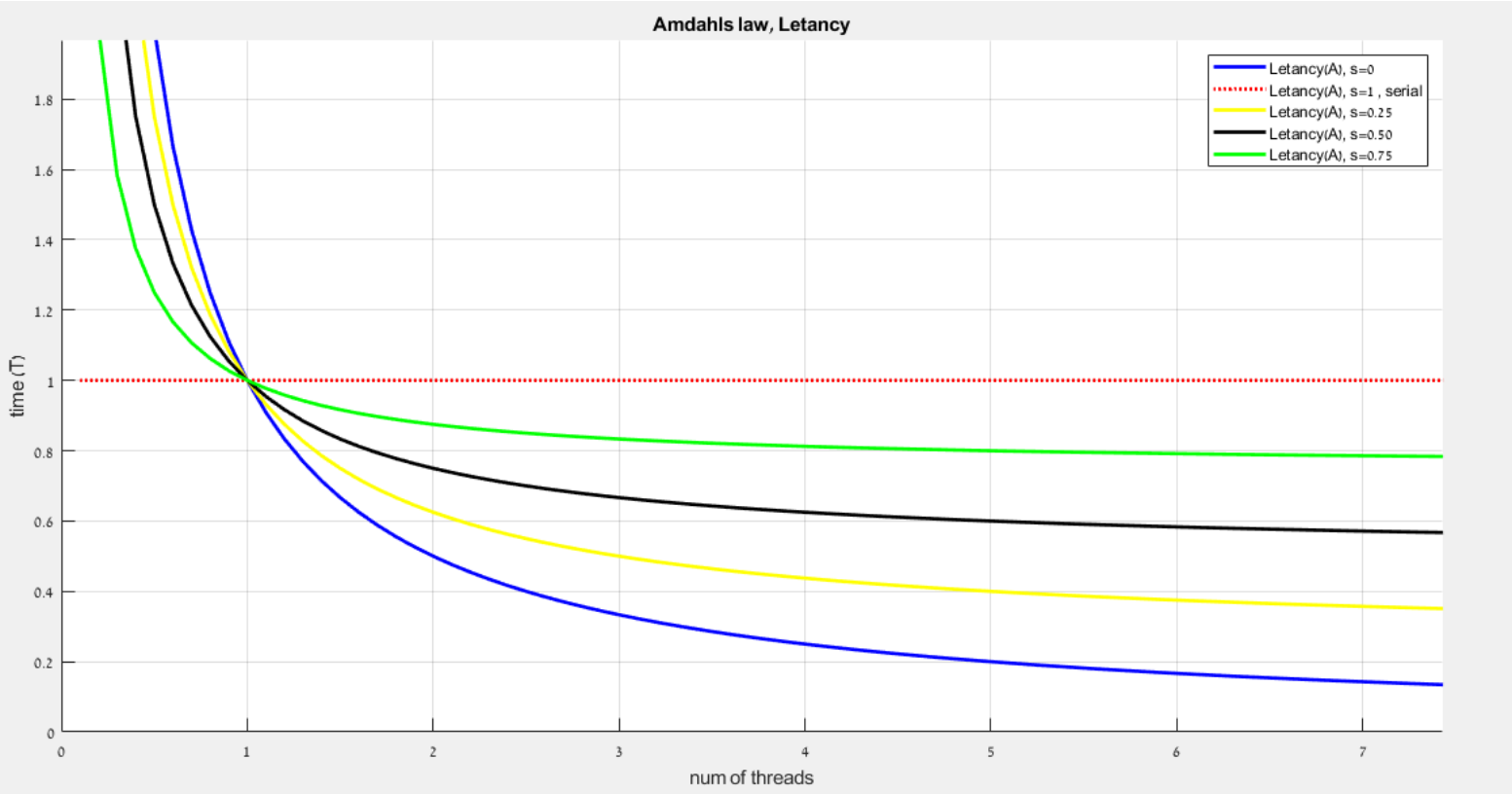
נניח אלגוריתם A המורכב ממספר רב של עבודות $j_1, j_2, \ldots, j_M$
לנוחיתכם, מושגים נפוצים לניתוח Performance של האלגוריתם:

- **Latency(A)** – זמן החישוב הכולל של האלגוריתם A
- **Latency(j)** – זמן החישוב הכולל של עבודה j
- **Turnaround Time(j)** – זמן החישוב + זמן ההמתנה בתור של עבודה j
- **Throughput** – **תפוקה** – מספר העבודות המסתיימות ביחידת זמן

בשאלה זו יש לצייר מספר גרפים, ולהסבירם. על הגרפים לכלול כותרת, מקרא, שמות צירים והסבר קצר על מה התקבל בגרף ולמה.

א. הניחו ש-A ניתן למקבול באופן מלא. ציירו גרף **איכותי** של ה-Latency(A) כתלות במספר החוטים N, עפ"י חזונו של אמדל (Amdahl)

ב. הוסיפו לגרף בסעיף (א) מעלה עקומים המתארים גם A סריאלי לחלוטין, ו- A המורכב מחלק של $s=\{0.25,0.5,0.75\}$ שניתן למקבול.



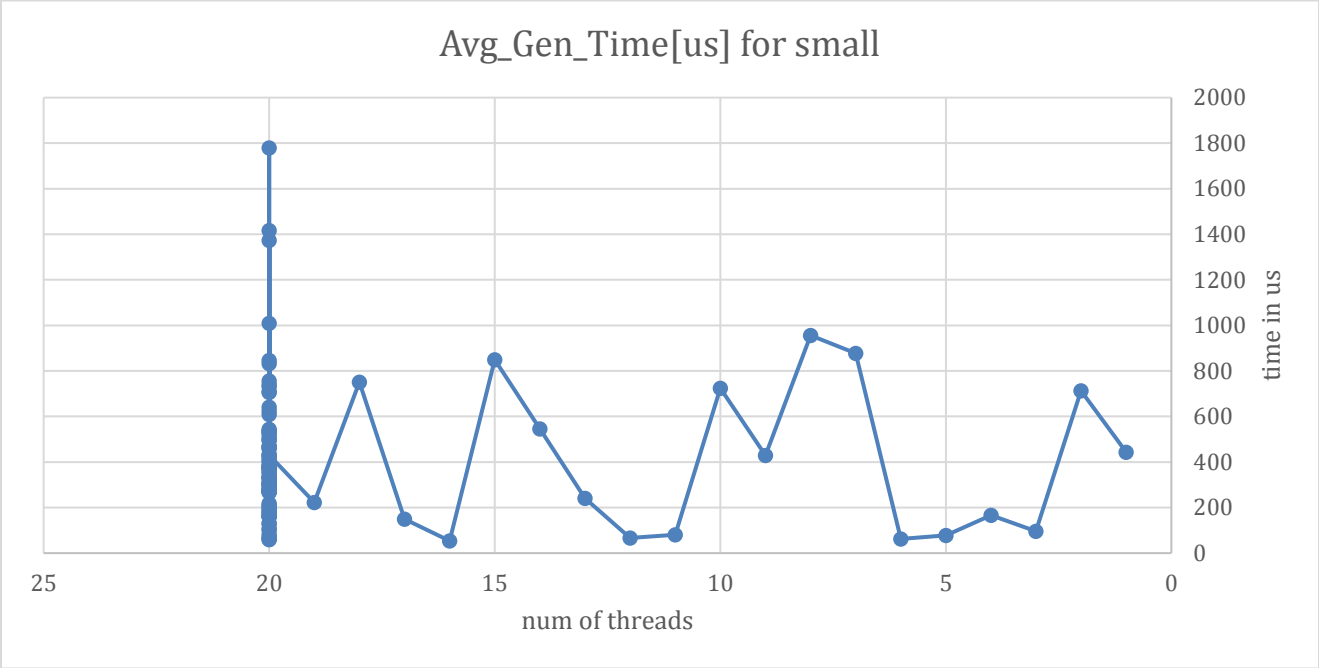
ג. עתה נתפנה לנתח את התוצאות המתקבלות מהאלגוריתם שכתבתם בחלקו הרטוב של התרגיל.
נגדיר את A כחישוב לוח משחק יחיד, פעולה הנעשת ע״י N חוטים במקביל. נגדיר חישוב כל Tile כעבודה j.
הריצו שלושה עומסים שונים על המערכת: small.txt, mid.txt, big.txt וציירו לכל אחד שני גרפים:

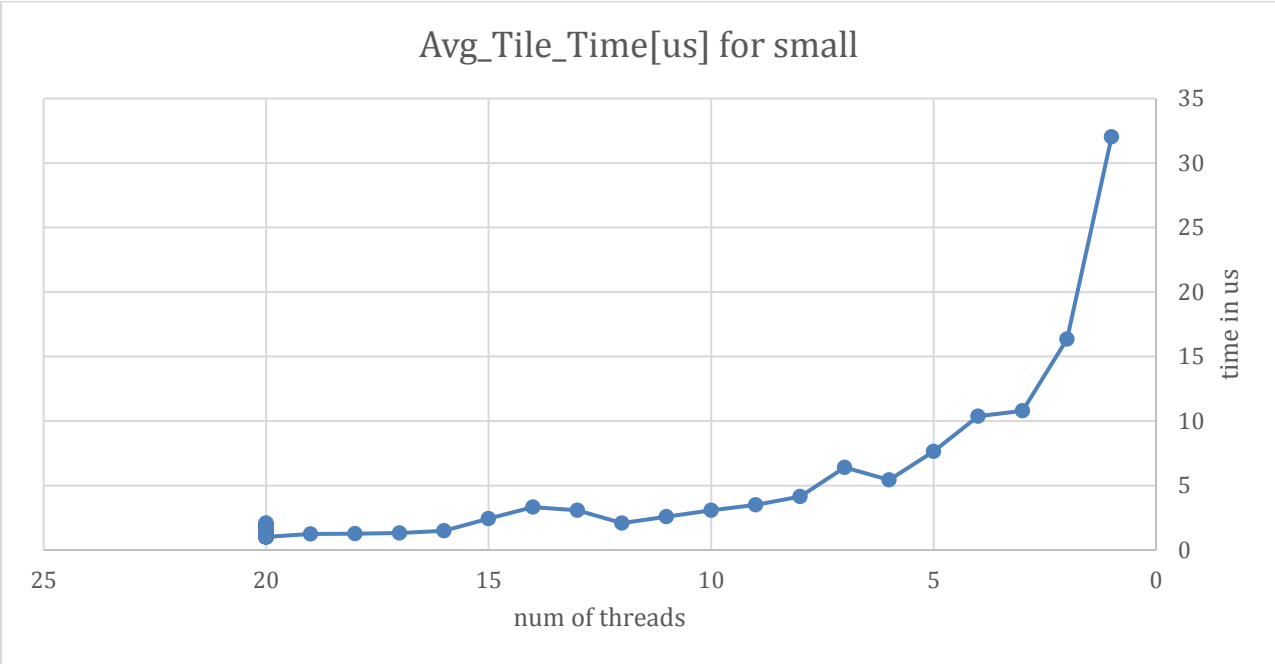
i. גרף של Average Latency(A) כתלות במספר החוטים N.

ii. גרף של Average Latency(j) כתלות במספר החוטים N.

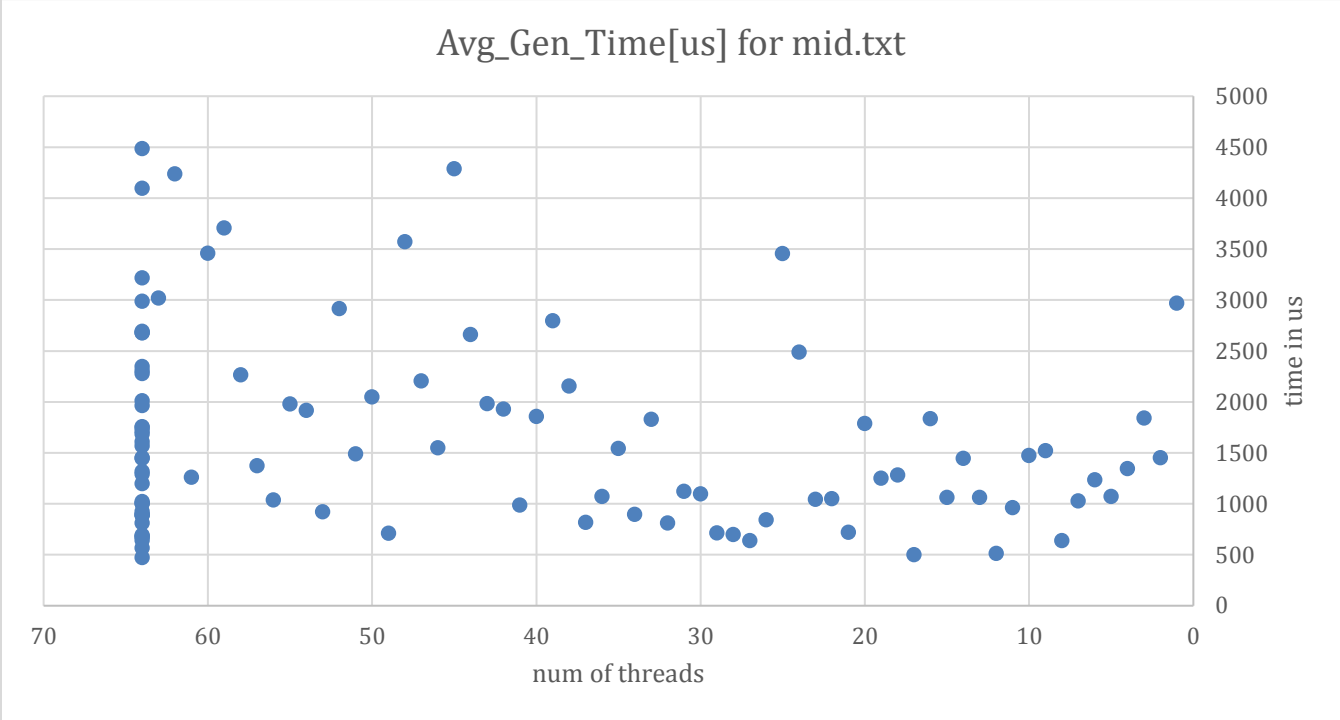
- יש תחילה לכבות את דגל ההדפסה print_on על מנת שהזמנים השונים לא יושפעו מתהליך ההדפסה. יש להעביר לארגומנט האחרון ל-main N במקום Y.
- יש להריץ כל קוניפגורציה ל-Generations 100, עם מספר חוטים משתנה: $N=1,2,3,\ldots,100$, ולהשתמש ב-Avg Gen Time ו-Avg Tile Time המתקבל ב-results.csv. ניתן לכתוב סקריפט bash קצר לביצוע דבר זה.
- ניתן לטעון קבצי CSV (Comma Seperated File) ישירות לאקסל באופן פשוט וקל ע״י פקודת Import.

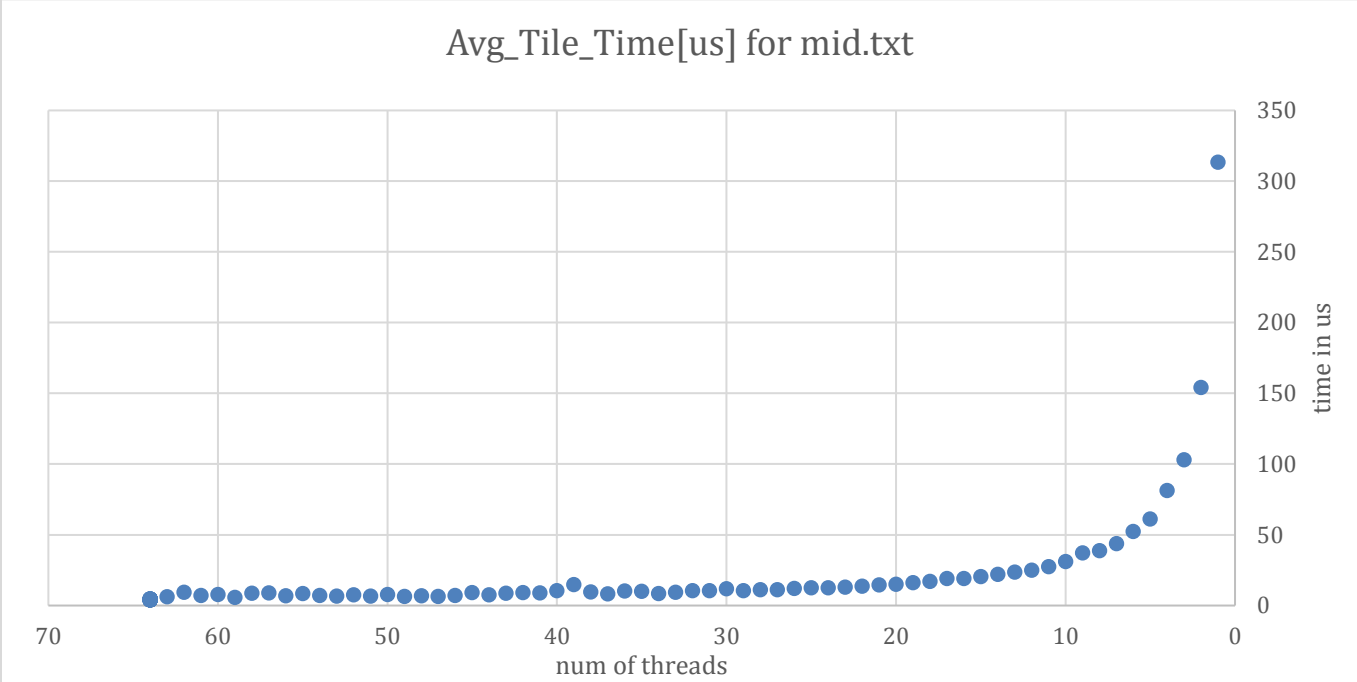
תוצאות של SMALL



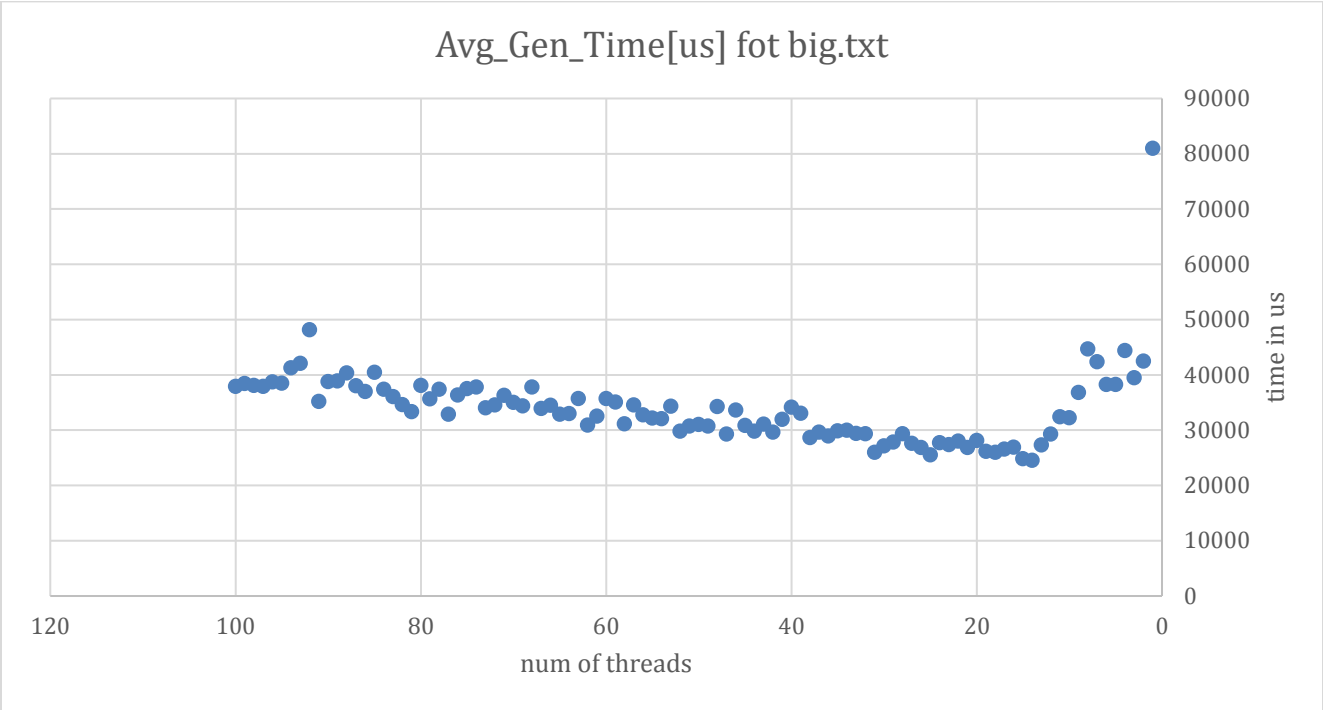


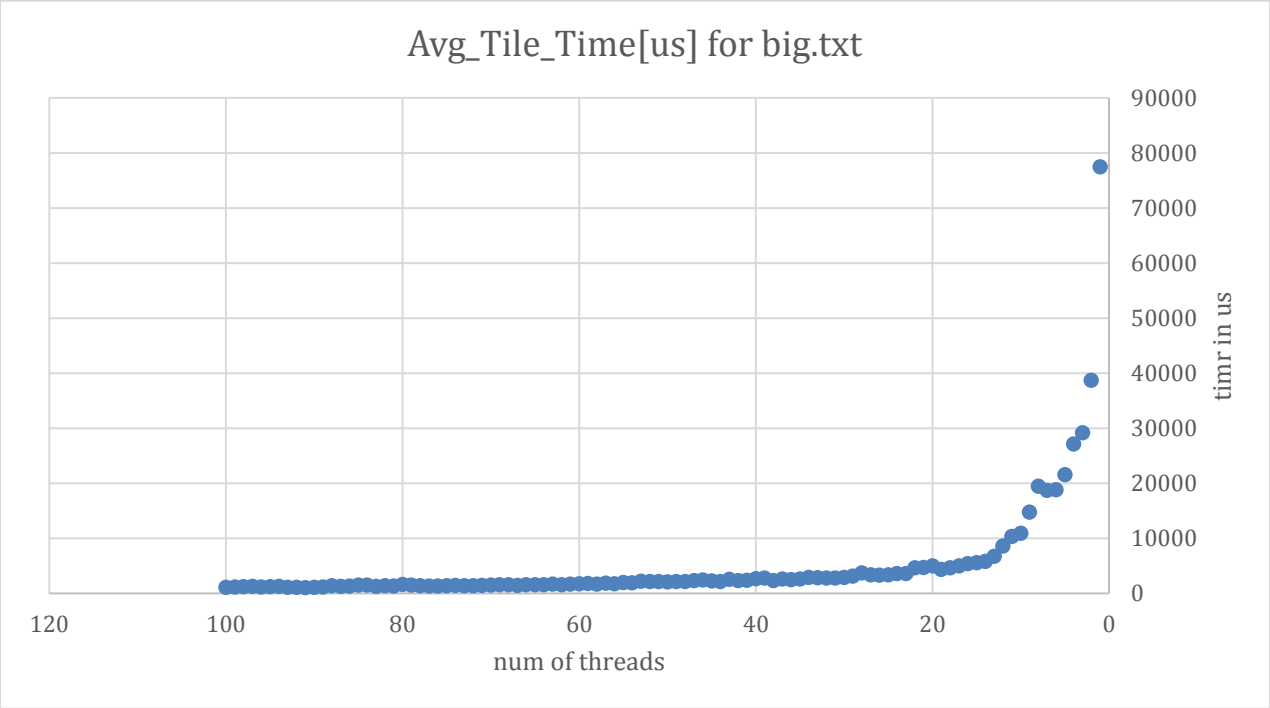
תוצאות של MID:





תוצאות של BIG:


Avg_Gen_Time[us] fot big.txt
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
time in us
120
100
80
60
40
20
0
num of threads


Avg_Tile_Time[us] for big.txt
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
timr in us
120
100
80
60
40
20
0
num of threads

ד. נתחו את הגרפים שהתקבלו באופן **מעמיק.** הניחו בניתוח שהחישוב מבוצע על מעבד יחיד.
שאלות מכוונות אליהם התייחסו בניתוח:

a. האם בחלקים מסוימים ניתן לראות מגמה כזאת או אחרת? עליה/ירידה/קו שטוח ? ממה הדבר נובע לדעתכם?

בכל העומסים בגרפי הavg tile time ניתן לראות ירידה ככה שמשתמשים ביותר חוטים הדבר נוסע מכך שכל חוט מתפל בחלק קטן יותר של המטריצה.

בavg gen time הזמן בתחילה יורד ואז מתחילה עליה נובע מכך שאחרי מספר מסויים של חוטים זמני החלפת ההקשר משמעותיים יותר ופוגעים בביצועים,בנוסף ישנם הרבה פיקים שונים שנובעים מהחלפות הקשר של המעבד לתוכניות אחרות. אפשר לראות זאת בבירור יותר בעומסים הקטנים יותר ששם הזמנים האלה משמעותיים יותר.

b. מהו מספר החוטים האידיאלי לכל עומס ? הסבירו סיבות לכך.

עבור small : רואים שמעל 15 חוטים אין שיפור גדול בavg tile time וקשב לראות את המגמה ב avg gen time בגלל הסיבות מסעיף קודם.

עבור mid : גם באזור ה15 חוטים רואים ש avg tile time מתייצב ובavg gen time רואים שמתחילה מגמת עליה שוב מהסיבות מסעיף קודם.

עבור big : סביבות 17 ניתן לראות בברור בגרף avg gen time ירידה ואז עליה כמו שהוסבר סעיף קודם.

c. האם בהכרח זמן החישוב היה משתפר אם היינו מוסיפים **מעבדים** נוספים?

עבור עומסים גדולים היינו רואים שיפור משמעותי כי שם רוב הקוד יהיה ממוקבל (חישוב גדול על כל tile) ובעומס קטן כאשר הקטע שניתן למקבל קטן לא בטוח שהיינו רואים שיפור משמעותי.

d. השוו בין הגרפים של העומס הקטן, גדול ובינוני. במידה ויש שוני, ממה נובע השוני בין הגרפים של העומסים השונים?

השוני נובע מאורך הקוד שאנו ממקבילים. ומכך שבעומס קטן הפרעות של ההחלפות הקשר פוגעות בנו יותר.

e. האם הגרף מתנהג כמו אחד הגרפים שהתקבלו בסעיפים א',ב' באופן גס? אם כן, כמה מקבילי אתם מעריכים שהקוד שלכם?

גרפי הavg tile time נראים כמו בסעיף א כי העבודה מתחלקת למספר רק של חוטים.

גרפי ה avg gen time לא נראים דומה כי אנו עובדים על מעבד אחד וסך הכל הגרף יהיה מושפע מכמות החלפות ההקשר ואורך העבודה הכללית שיש לבצע.

שימו לב: הגרפים שיתקבלו בסעיף זה יכולים להיות שונים ומגוונים. לא בהכרח שהגרפים יסתדרו עם הציפיות שלכם. במידה ומתקבלים גרפים המתארים התנהגות לא "מקבילית" – בדקו את מימושכם עד שהשתכנעתם שהוא סביר. בכל מצב, הצדיקו את הגרפים שהתקבלו עם טיעונים איכותיים. הניקוד בחלק זה ינתן עבור הסברים משכנעים של התוצאות בגרף, המראים הבנה של החומר ושיקולי המערכת.