Yonathan Bettan

302279138

yonibettan@gmail.com

Alon kwart

201025228

Alon.kwart@gmail.com

המבנה שבחרנו:

```
typedef struct {

    int vertex;
    int cost;
    /* for example {3, 5, 1, NOT_SET, NOT_SET} for citiesNum=5 */
    int shortestPathUntilNow[1];

} State;
```

המבנה בו בחרנו להשתמש עבור ייצוג של "ג'וב" הוא מבנה בשם State אשר מכיל 3 שדות:

- מספר הצומת של העיר בה אנו נמצאים כרגע
- מחיר הצומת בהתחשב בסכום המרחקים של כל הצמתים במסלול עד כה
- מערך של int המכיל מסלול שלם או חלקי של כל הצמתים שהובילו עד הצומת vertex של המבנה.
  - הצומת האחרון במסלול הנ"ל שאינו מכיל את הערך NOT_SET הוא תמיד מספר הצומת (השדה הראשון) וייצרנו לו שדה לצורך נוחות

פונקציות משותפות עיקריות:

- בחלק זה נסביר על מספר פונקציות בעלות תפקיד מרכזי הן בחלק הסטטי והן בחלק הדינאמי
- יתכנו שינויים קלים בין הגרסה הסטאטית לדינאמית אשר יפורטו בחלקים א' ו-ב'

Cpu_main:

זוהי הפונקציה רקורסיבית שכל מעבד מריץ על כל אחת מהצמתים בעץ החיפוש שהם קיבלו.

מדובר בשיטת branch and bound אשר בכל שלב מוסיפה עיר למסלול, בודקת בעזרת יוריסטיקה אם בכלל ניתן להגיע לפתרון טוב יותר מהפתרון הנוכחי ואם כן מפתחת את הצומת אחרת "גוזמת ענף זה מעץ החיפוש".

Heuristic:

היוריסטיקה שבחרנו מחשבת סכום שכל איבר בו מייצג את הקשת המינימלית עבור צומת מסוים **שאינו** נמצא במסלול ומחברת אותו למחיר הצומת אותו היא מחשבת.

כלומר עבור צומת מסוים המכיל מסלול חלקי פונקציה זו נותנת את הפתרון האופטימלי שיתכן (שאולי אף לא קיים) עבור צומת זו מבלי לפתח אותו.

חלק א:

Tsp_main:

הפונקציה הראשית מחולקת למספר חלקים עיקריים:

- **העברת המידע ההתחלתי (1)** – ה- master שולח את הנתונים ההתחלתיים citiesNum, xCoord and yCoord וה- workers מקבלים מידע זה.
- **רגיסטרציה של הטיפוס אותו יצרנו לספריית MPI**
- **ביצוע הלוגיקה המרכזית –** ה- master וה- workers ממשים לוגיקה שונה
- **שחרור הזיכרון**

rootExec:

זוהי הלוגיקה של תהליך ה- master.

בשלב הראשוני פונקציה זו מחשבת את עומק עץ החיפוש אש יניב מספר צמתים שגדול ממספר המעבדים בפעם הראשונה, כלומר היא מחשבת מה אורך ה- prefix של המסלול המינימלי עבורו כבר יש מספיק עבודה לכל המעבדים.

לאחר שבידיה מידע זה היא מייצרת מערך, בגודל מספר העבודות, שכל איבר בו הינו מטיפוס State אותו יצרנו ומכיל בתוכו את כל המידע שמעבד צריך על מנת להריץ את החישובים על כל הפתרונות האפשריים ש-prefix זה מסוגל להניב.

בסעיף זה לא נתנו לנו מגבלה על סיבוכיות הזיכרון ובנוסף זהו הבאפר אשר משמש לתקשורת ולכן אין כאן חריגה ממגבלות התרגיל.

התהליך שולח לכל מעבד **חלק** מסוים ממערך זה **(1)** ע"י שימוש ב- point to point communication.

כאשר כל המעבדים קיבלו את חלקם, ה- master לוקח חלק בחישובים ומחשב את חלקו על מנת להקל על שאר המעבדים, הרי אין שום סיבה שהוא ינוח ויחכה לסיום שאר המעבדים.

לאחר שכל המעבדים סיימו את עבודתם הוא מרכז את התוצאות **(2)** ע"י collective communication, מחשב את התוצאה האופטימלית ומחזיר אותה.

otherExec:

זוהי הלוגיקה המרכזית של כל תהליכי ה- workers.

פונקציה זו מקבלת תחילה את מספר הצמתים אותה היא עומדת לקבל **(3)**.

ברגע זה היא מקצה buffer בגודל מספר הצמתים אותה היא הולכת לקבל כך שכל איבר ב- buffer מייצג צומת בעץ החיפוש עליו יש לבצע חישובים.

כעת כשה- buffer הוקצה, הפונקציה מוכנה לקבל את הנתונים בפועל **(4)**.

כאן החלטנו לשים barrier אשר יוודא שכל התהליכים מתחילים את חישוביהם יחד.

לבסוף כל מעבד מריץ את החישובים שלו , מחשב את הפתרון האופטימלי עבורו ושולח את התוצאה בחזרה ל- master **(5)**.

מבנה התקשורת:

1. בחלק זה ה- master הוא worker בעצמו ולכן רצינו ששליחת הנתונים תתבצע תוך כדי המשך עבודה (בעיקר בהמשך חלוקת המטלות) לכן בחרנו להשתמש ב- MPI_Bsend אשר תאפשר לו להמשיך לעבוד תוך כדי השליחה.
2. החלטנו להשתמש ב- MPI_Gather אשר נחשבת ל- collective communication שכן יש צורך באיסוף מרוכז של כל התוצאות, הרי ל- master אין מה לעשות עם תוצאות של חלק מהתהליכים האחרים בלבד ולכן collective communication כאן הינה פתרון טוב.
3. מכיוון השה- worker אינו יודע מראש מה מספר ה- States אותם הוא עומד לקבל, בחרנו להשתמש ב- dynamic receive ע"י שימוש בקריאה החוסמת MPI_Prob בכדי לקבל את גודל ה- buffer לפני הקבלה עצמה ע"י שימוש ב- MPI_Get_count.
4. בחלק הנ"ל המטרה הייתה, כמו שהבנו אותה, לחלק את כל העבודה בבת אחת ולהתפנות לחישובים. לכן מכיוון ש- MPI בנויה לשליחת מערכים בחרנו להשתמש ב- MPI_Recv עם מספר איברים אשר אותו גילינו בהתאם לסעיף 3 בכדי לעמוד בדרישות התרגיל אשר דרשו שימוש ב- point to point communication.
5. בשלב זה כל התוצאות מוכנות ולכן בחנו להשתמש ב- collective communication לאיסוף התוצאה ע"י MPI_Gather.

חלק ב:

Tsp_main:

הפונקציה הראשית מחולקת למספר חלקים עיקריים:

יצירת מערך גלובלי המכיל עבור כל צומת את הקשת המינימלית שלו על מנת לייעל ביצועים עבור ה- workers

- **רגיסטרציה של הטיפוס אותו יצרנו לספריית MPI**
- **ביצוע הלוגיקה המרכזית –** ה- master וה- workers ממשים לוגיקה שונה
- **שחרור הזיכרון**

rootExec:

זוהי הלוגיקה של תהליך ה- master.

בחלק הדינמי החלטנו לעבוד עם prefixes כמו בחלק הסטטי, עבור מצב נתון במרחב החיפוש, המיוצגים ע"י מערך של int באורך קבוע prefixLen=4 שנקבע כך ממספר סיבות:

- לפי הנחות התרגיל ב- FAQ ניתן להניח כי לא תתבצענה בדיקות עבור מספר ערים הקטן מ-5 מה שמבטיח לנו שה- prefix שלנו תמיד יהיה חוקי
- לפי הנחה נוספת ב- FAQ מספר המעבדים לא יהיה גדול ממספר ה- prefixes הקיימים באורך 4 עבור מספר ערים נתון מה שמבטיח לנו שכל מעבד יקבל לפחות עבודה אחת ולכן, לא נגיע למצב שבו מעבד מגיע ל-deadlock עקב בקשת prefix שלא סופקה.

Can I make any assumptions about the number of cities and processors?
If it makes data partitioning easier for you, you may assume there are at least 5 cities (like in our implementation). You may also assume that there won't be more processors than the number of 4-city long sub-routes (so you can limit the partitioning granularity on the static part). Note, however, that an ideal solution would not have to make these assumptions. Therefore, if you do make them, make sure you check the input at the beginning of the run, and document your limitations.

בשלב הראשוני פונקציה זו מחשבת את ה- prefix הראשון $[0,1,\ldots prefixLen-1]$ ולאחר מכן פותחת 2 קווי תקשורת אסינכרוניים שמטרתם לקבל בקשות מה- workers.

קו תקשורת ראשון עבור בקשת עבודה חדשה **(1)** וקו תקשורת שני עבור bound חדש שהתגלה **(2)**.

מרגע זה מתחילה שגרה קבועה של טיפול בבקשות חוזרות ונשנות של ה- workers, שתפסק ברגע שתשלח הודעת סיום לכל אחד מה- workers **(3)**.

בשגרה זו ה- master מחשב את ה- State הבא לשליחה וממתין לבקשות. עבור בקשה עבודה חדשה ה- State ישלח **(4)** ועבור עדכון bound **(5.1)** ה- master ישמור אצלו את התוצאה הטובה ביותר עד כה לאחר שיבדוק כי היא באמת קטנה יותר (למקרה בו נשלחו עדכונים מכמה workers במקביל) וחוזר חלילה.

הלולאה מתבצעת בצורת לולאה שרצה באופן תמידי כך שהעדיפות הינה לספק בקשות כמה שיותר מהר ללא סדר מסוים בניהן.

בסיום שגרת הטיפולים הוכנס barrier בכדי לוודא שכל התהליכים מסיימים יחד כנדרש בתרגיל ובנוסף על מנת להבטיח שברגע עיבוד התוצאות כל ה- workers שלחו את התוצאות שלהם ל- master.

ה- master אוסף את התוצאות של כל התהליכים **(6)**, בוחר פתרון של תהליך כלשהו (ב- otherExec נבין איך זה מתבצע), מוודא שאכן הפתרון שהתקבל הוא אכן באורך הפתרון הטוב ביותר שה- master מכיר, משחרר את הזיכרון שהוקצה ומחזיר את התוצאה.

:otherExec

זוהי הלוגיקה המרכזית של כל תהליכי ה- workers.

הפעולה הראשונה שפונקציה זו עושה הינה פתיחת האזנה אסינכרונית לעדכוני bound **(1)** ולאחר מכן מבקשת וממתינה לעבודה ראשונה **(7.1)**.

לאחר פעולות אתחול אלו הפונקציה נכנסת לשגרה המכילה מספר שלבים אשר יתבצעו עד אשר תגיע הודעת סיום **(3)**:

- בקשת עבודה נוספת **(7.2)**
- הרצת הפונקציה cpu_main על ה- State הנוכחי וקבלת תוצאה **(8)**.
- בדיקה עם התקבלו עדכוני bound **(5.2)** בכדי להתעדכן במה שקרה בזמן שהיינו עסוקים בחישוב ארוך.
- השוואת התוצאה שקיבלו אל מול התוצאה הטובה ביותר עד כה ובמידה ומצאנו bound חדש:
  - עדכון התוצאה המקומית שלנו (אורך ומסלול)
  - שליחת עדכון לשאר התהליכים על מציאת bound חדש **(9)**
- בשלב זה כבר אין לנו מה לעשות ולכן נמתין שהבקשה שבצענו בתחילת השגרה תגיע ונקבל State חדש **(10)**

כאשר ה- worker סיים את עבודתו, שולחים את התוצאה ל- master **(6)** , ממתינים על barrier, משחררים זיכרון ומסיימים

מבנה התקשורת:

1. **ההאזנה מתבצעת בצורה אסינכרונית ע"י הפונקציה MPI_Irecv עם שימוש ב- JOB_REQUEST_TAG.**
2. **ההאזנה מתבצעת בצורה אסינכרונית ע"י הפונקציה MPI_Irecv עם שימוש ב- BOUND_TAG.**
3. **הודעת סיום מיוצגת ע"י**

$$state = \{.vertex = NOT\_SET,$$
$$.\cos t = 0,$$
$$.shortestPathUntilNow = \{NOT\_SET, ...NOT\_SET\}\};$$

4. **מתבצעת בדיקה עם הגיעה job request חדש ע"י שימוש ב- MPI_Test, במידה וכן נשלח State ל- worker המבקש ע"י MPI_Ssend שכן ה- master פחות עסוק מה-workers ולכן אנחנו מעדיפים פונקציה בטוחה יותר גם אם היא קצת יותר איטית. לבסוף מחדשים את ההאזנה כמפורט בסעיף 1.**
5.
   5.1. **עבור ה- master הטיפול בבקשה זו הוא יחסית פשוט, מתבצעת בדיקה אם הגיע bound request ע"י MPI_Test ובמידה וכן מעדכנים את הערך המינימלי שה- master מכיר ולבסוף מחדשים את התקשורת כמפורט בסעיף 2.**
   5.2. **ה- worker מבצע בדיוק את מה שמבצע ה- master ב- 5.2 בתוספת זה שהוא פוסל את המסלול שהוא מכיר- המסלול הטוב ביותר ש- worker מכיר הוא מסלול אשר אורכו תואם את האורך המינימלי שהוא מכיר, לכן אם ערך זה משתנה אין יותר משמעות למסלול ונאפשר אותו ל- $[NOT\_SET, ...NOT\_SET]$ שכן ה-master מסתמך על פסילות אלו באיסוף התוצאה.**
6. **בשלב זה כל התוצאות מוכנות ולכן בחרנו להשתמש collective communication ע"י MPI_Gather לצורך איסוף התוצאות מכל ה- workers.**
7.
   7.1. **מכיוון שמדובר בהודעה ראשונה ואין לנו מה לעשות בלעדיה החלטנו להשתמש בתקשורת חוסמת ע"י MPI_Ssend עם JOB_REQUEST_TAG לצורך שליחת בקשת העבודה ו- MPI_Recv עם JOB_TAG לצורך קבלת ה- State עצמו.**

7.2. כאשר בקשה זו מתבצעת יש לנו כבר State בקנה עליו אנחנו הולכים לעבוד ולכן אנחנו רוצים לבצע את התקשורת במקביל לחישובים. עקב כך בחרנו להשתמש בפונקציות MPI_Issend עם JOB_REQUEST_TAG ו- MPI_Irecv עם JOB_TAG.

8. ההבדל היחיד עם ה- cpu_main של החלק הסטטי הוא שכן הערך ההתחלתי שה- bound מקבל הוא הערך המינימלי שה- worker מכיר לעומת החלק הסטטי שמגדיר פרמטר זה כ- INF.

9. התכנון המקורי היה לעדכן את ה- master ושהנ"ל יעדכן את כל שאר התהליכים ע"י MPI_Bcast אך לאחר התייעצות עם מתרגל בחרנו את השליחה הבאה:

המתנה תחילה ששליחת העדכון האחרון תסתיים ע"י MPI_Wait (במידה ומדובר בשליחה ראשונה אין המתנה) ולבסוף שליחה ע"י MPI_Issend עם BOUND_TAG בלולאה לכל שאר התהליכים.

שלום,

לא, המטרה היא שכל עובד יעדכן בעצמו את שאר העובדים ולא דרך rank 0.

בהצלחה,
איתי וענת

בתאריך יום ג', 9 בינו' 2018 ב-12:59 מאת גארי :) <harrymbox@gmail.com>:

10. שימוש ב- MPI_Wait פשוט עבור סיום שליחת ה- job request ו- MPI_Wait נוסף עבור סיום קבלת ה- job.

נק' עדינות:

בחלק הסטטי אנו אנו משתמשים במערך של States לצורך חלוקת המטלות בצורה מסודרת בין התהליכים השונים.

מכיוון שספריית MPI אינה תומכת ברגיסטרציה של טיפוסים מורכבים (המכילים מצביעים) אז החלטנו שהשדה במבנה המכיל את המערך יכיל מערך סטטי בגודל 1 וכאשר אנו מקצים זיכרון בצורה דינאמית אנחנו נקצה עבור עוד $citiesNum-1$ integers.

הבעייתיות המתעוררת מכך היא שכבר לא ניתן לגשת למערך זה (מטיפוס State*) ע"י אריטמטיקת פויינטרים פשוטה אחרת האיברים שלנו במערך ידרסו זה את זה.

לכן החלטנו לספק wrappers לקריאה וכתיבה למערך ע"י שימוש בהמרה ואריטמטיקת מצביעים קצת יותר מורכבת אשר עושה את העבודה המלוכלכת "מאחורי הקלעים".

ניתן למצוא פונקציות אלו תחת הלשונית array of states בחלק הסטטי.

בברכת בדיקה עדינה.