תכנות מקבילי ומבוזר

סיכום החומר בקורס "תכנות מקבילי ומבוזר" בטכניון

סיכם: דוד ארינזון

הבהרה: מסמך זה מסתמך במידה רבה על הקורס "תכנות מקבילי ומבוזר" בטכניון, אך אינו חומר רשמי של הקורס, אלא סיכום אישי בלבד. המקורות לכתיבת המסמך הם ההרצאות והתרגולים, והזכויות שמורות לפקולטה למדעי המחשב בטכניון ולמוריה. בראש כל פרק מופיע קישור למקור לכתיבת הפרק.

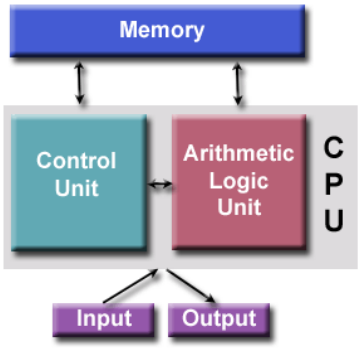
המסמך הנ"ל מתבסס כולו על הקורס תכנות מקבילי ומבוזר (236370) ועל החומרים (שקופיות וקישורים) אשר מפורסמים באתר. תחת כל נושא מופיעים הקישורים הרלוונטיים אשר מהם נלקח החומר. המסמך רחוק מלהיות שלם (וכנראה בחלקו גם לא נכון), ולא מומלץ להשתמש בו לבדו על מנת להבין את כל הבלאגן הזה. אני לא ממליץ לזנוח את השקופיות, שכן הרבה מאוד חומר מהן (בפרט דוגמאות הקוד, והדוגמאות המופשטות) לא הוכנס לכאן. יאללה, שיהיה בהצלחה!
דוד אריינזון.

תוכן עניינים

נושא	עמוד
מבוא	3
זמן וירטואלי (Virtual time)	15
גילוי תנאי תחרות (Data race decection)	17
OpenMP	21
TBB	24
Shared Memory	26
BSP	27
Transactional Memory	29
Map-Reduce	33
Condor	36
CUDA	40
תכנות מקבילי ב-Java	42
Optimistic Design	44
MPI	46
DAGman	51
שיטות סנכרון בעבודה עם רשימות מקושרות	52
השוואה בין MPI ל-OpenMP	55

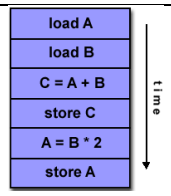
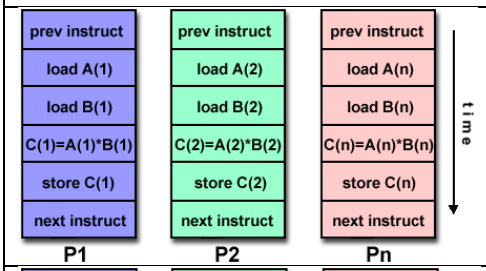
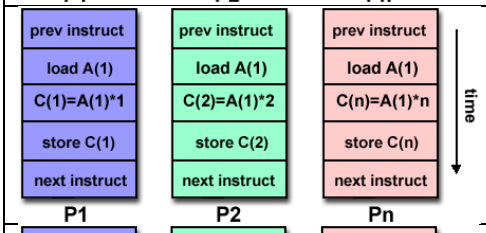
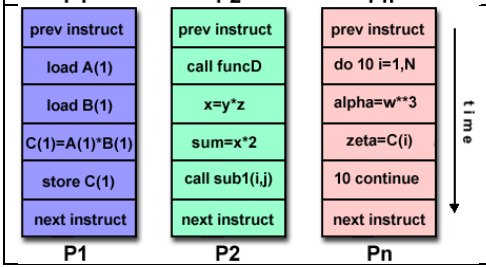
מבוא

/http://www.llnl.gov/computing/tutorials/parallel_comp



ארכיטקטורת פון-ניומן (von Neumann) - ארכיטקטורה בסיסית אשר פותחה בשנת 1945, הכוללת את ארבעת הרכיבים המרכזיים אשר מתוארים בתרשים. כאשר ה-Control Unit מוציא את הפקודות ודואג לביצוע שלהן בצורה סיריאלית.

הסיווג הקלאסי ע"פ פלין (Flynn's Classical Taxonomy) - סיווג כללי של סוגי המחשבים המקביליים, אשר הוצג בשנת 1966. מסווג את סוגי המחשבים לפי אופן ההתמודדות וההתייחסות למידע ולפקודות.

SISD – Single Instruction Single Data	שיטת העבודה הסיריאלית והבסיסית ביותר. כאשר בכל מחזור שעון, מופעלת (/מועברת למעבד) רק פקודה אחת, ומעובד פריט מידע יחיד.	
SIMD – Single Instruction Multiple Data	שיטת עבודה מקבילית. כאשר כל המעבדים מבצעים את אותן הפקודות בכל מחזור שעון. אולם, כל מעבד יכול לעבוד על חלק שונה של המידע (באיור מוצגת עבודה על וקטור). שיטת ההרצה הינה דטרמיניסטית ומסונכרנת (GPU's מבוססים על השיטה הנ"ל)	
MISD – Multiple Instruction Single Data	שיטת עבודה מקבילית. בשיטה הנ"ל, פריט מידע יחיד מועבר לכל המעבדים, וכל מעבד מבצע עליו סוג שונה של פעולות אשר ב"ת במעבדים האחרים	
MIMD – Multiple instruction Multiple Data	שיטת העבודה המקבילית המקובלת היום, רוב המחשבים המודרניים נכללים בקטגוריה הנ"ל. כל מעבד מכיל סט פקודות משל עצמו, ויכול לפעול על מידע שונה. ההרצה יכולה להיות סינכרונית/אסינכרונית, דטרמיניסטית/אי-דטרמיניסטית	

מטלה (Task) - משימה כחלק מעבודה חישובית כוללת. בדר"כ מאופיינת ע"י תוכנית, או סט של פקודות המבוצעות ע"י מעבד.

מטלה מקבילית (Parallel Task) - מטלה שיכולה להתבצע על מספר מעבדים במקביל.

סנכרון (Synchronization) - תיאום בין מטלות מקביליות. קביעת נקודות זמן, אשר כל המטלות/מעבדים צריכים להגיע אליה, על מנת להתקדם הלאה.

גרעיניות (Granularity) - היחס בין כמות החישובים לכמות התקשורת בתוכנית מקבילית.
ישנן שתי קטגוריות כלליות:

1) גרעיניות גסה (Coarse) – כמות גדולה (יחסית) של חישוב מבוצעת בין נקודות תקשורת שונות.
2) גרעיניות עדינה (Fine) – כמות קטנה (יחסית) של חישוב מבוצעת בין נקודות תקשורת שונות.

האצה מקבילית (Observed Speedup) - הערכה גסה להאצה של תוכנית מקבילית אל מול אותה תוכנית בהרצה סיריאלית (זמן ריצה סיריאלי / זמן ריצה מקבילי)

תקורת מקביליות (Parallel Overhead) - המחיר שיש לשלם (בזמן ריצה) על אופן חישוב/עבודה מקבילי. מאפיינים אשר משפיעים על התקורה הנ"ל הינם לדוגמא:

1) זמני התחלה וסיום של מטלות
2) סנכרונים
3) תקשורת מידע בין מטלות/מעבדים
4) תקורת תוכנה כתוצאה מעיבוד מקבילי

מקביליות מאסיבית (Massively Parallel) - שימוש במספר גדול מאוד של מעבדים/מחשבים לעבודה מקבילית.

מקביליות כמעט מוחלטת (Embarrassingly Parallel) - פתרון של מספר עצום של מטלות דומות מאוד, תו"כ שימוש בתיאום מינימלי ביניהן.

יכולת התאמה (Scalability) - היכולת של מערכת מקבילית להעלות את ההאצה (Speedup) כתוצאה של תוספת מעבדים. דוגמאות לגורמים אשר משפיעים על היכולת הזאת:

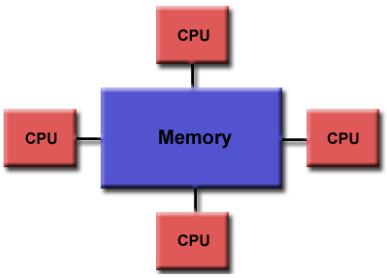
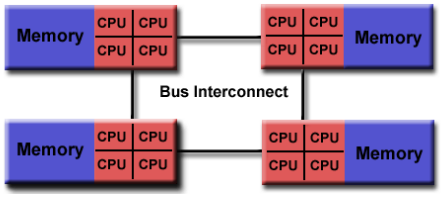
1) חומרה, בפרט תקשורת ורוחב הפס בין המעבד לזיכרון
2) האלגוריתם אשר האפליקציה מריצה
3) תקורת המקביליות
4) מאפיינים הקשורים לאופן כתיבת האפליקציה (ברמת הקוד, לדוגמא)

אשכול מחשבים (Cluster Computing) - שימוש במספר מחשבים, ויחידות מסוגים שונים לצורך הרכבת מערכת מקבילית.

מודלי זיכרון

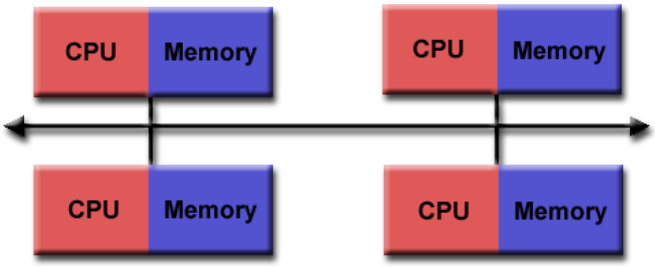
זיכרון משותף (Shared Memory)

המטרה היא לאפשר למספר המעבדים השונים לגשת לאותו זיכרון ולפעול עליו. בצורה זו, מעבדים יכולים לשתף ביניהם מידע רלוונטי, ושינויים של מעבד כלשהו זמינים למעבדים האחרים.

UMA – Uniform Memory Access	במודל הזיכרון הנ"ל, המעבדים זהים, וכל אחד מהם יכול לגשת לזיכרון המשותף. כלל המעבדים נמצאים באותו מרחק מהזיכרון(*עקרון SMP – Symmetric Multiprocessor*), לכן "מהירות הגישה" שלהם זהה. מודל מורכב יותר משתמש בזיכרון מטמון (Cache), ודואג לכך שכאשר מעבד אחד מעדכן מידע כלשהו, כלל המעבדים מודעים לכך (דבר זה מבוצע ברמת החומרה) (*תאימות זיכרון המטמון –Cache Coherency*) שיטה זו דורשת סנכרון של הגישותלזיכרון, ובנוסף, קיימת כאן מגבלת יכולת התאמה (Scalability), אשר מייצרת מגבלה פיזית, ולא בהכרח ניתן להרחיב את המודל שיתאים למספר גדול יותר של מעבדים, שיהיו במרחק זהה מהזיכרון.	
NUMA – Non-Uniform Memory Access	במודל הזיכרון הנ"ל, זמני הגישה לזיכרון בין המעבדים השונים אינם זהים. הגישה לזיכרון דרך הקישור של ה-Bus היא איטית יותר. המודל הזה נמצא בשימוש בדר"כ כאשר רוצים לחבר מספר SMP's יחד לצורך שיתוף מידע.	

יתרונות	חסרונות
אזור כתובות גלובלי מאפשר תכנון "קל" בכל הקשרי הזיכרון והגישה אליו	ישנה פגיעה בסקלביליות של המודל, הוספת מעבדים תיצור בעיה גאומטרית של מיקום, ומרחק מהזיכרון, ותדרוש התאמות Cache רבות על מנת לשמור על המאפיינים של גישה (כמעט) זהה לזיכרון
שיתוף המידע בין המטלות (המעבדים) השונות הינו מהיר (יחסית) ושווה (Uniform), עקב הקרבה של המעבדים לזיכרון	האחריות על גישה מסונכרנת לזיכרון הינה על המתכנת בעת כתיבת האפליקציה (אין מנגנון תומך בחומרה)
	עלות יצירת מכונות בעלות זיכרון משותף מתייקר בהתאם לעליית מספר המעבדים

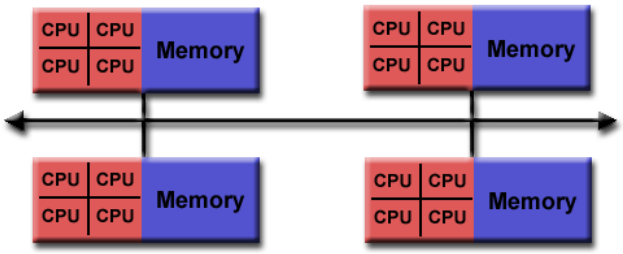
זיכרון מבוזר (Distributed Memory)



כל מערכות הזיכרון המבוזר השונות דורשות רשת תקשורת על מנת ליצור קשר בין הזכרונות המבוזרים במערכת. בגישה הנ"ל, לכל מעבד יש זיכרון משלו, ואין התאמה או מיפוי כלשהו בין זכרונות של מעבדים שונים (כלומר, אין כתובות זיכרון גלובליות, אלא רק ברמת המעבד). מסיבה זו, כל מעבד פועל על הזיכרון שלו באופן ב"ת. ולכן אין כאן צורך בתאימות זיכרון מטמון (כמו במודל המשותף), ואין השפעה/השלכה של פעולות אלו על זיכרון של מעבד אחר. כאשר מעבד/מטלה כלשהי זקוקה למידע ממטלה אחרת, אחריותו של המתכנת להגדיר את אופן העברת המידע, ולהפעיל את אמצעי הסנכרון המתאימים. אופן העברת המידע, וה"רשת" עליו מועבר המידע, אינם מקובעים לסוג מסוים, ויכולים להשתנות ממודל למודל, בהתאם לצרכים.

יתרונות	חסרונות
המודל מאפשר סקלביליות, כאשר עם כל תוספת מעבד, מתווסף לו הזיכרון המתאים	כל הקשור לתקשורת בין המעבדים "נופל" על המתכנת
כל מעבד יכול לגשת בצורה מהירה לזיכרון שלו, מבלי התקורה הנוצרת בצורך לשמור על תאימות זיכרון מטמון	קושי בעבודה/מיפוי של מבני נתונים, עקב אי-קיום זיכרון גלובלי
ניתן לשלב מוצרים שונים במודל הנ"ל, אשר יתקשרו אחד עם השני	הצורך של מעבד אחד לגשת לזיכרון של מעבד אחר

זיכרון משותף-מבוזר (Hybrid Distributed-Shared Memory)



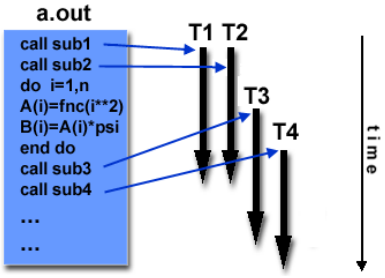
המחשבים המהירים והגדולים ביותר כיום דוגלים בשיטה היברידית ביחס לשני המודלים הקודמים. כל יחידה הינה SMP בפני עצמה, ויכולה להתייחס לזיכרון ש"מחובר" אליה כזיכרון בעל כתובות גלובליות. על מנת ש-SMP מסוים ייגש לזיכרון של SMP אחר, הוא עובר דרך התקשורת אשר מחברת את כולם. למרות שמטרת המודל הנ"ל היא לקחת את הטוב מבין שני המודלים, כמובן שהוא אינו מושלם, וסובל מבעיות דומות לאלו של שני המודלים הקודמים.

מודלי תכנות מבוזר

מודל זיכרון משותף (Shared Memory)

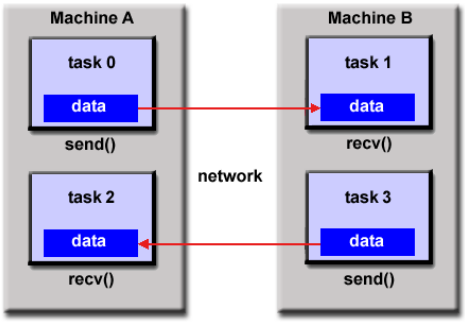
במודל הנ"ל, מטלות חולקות את אותו אזור הזיכרון, וקוראות/כותבות בצורה סנכרונית. מבוצע שימוש במנגנוני סנכרון כגון מנעולים/סמפורים על מנת לנהל את הגישות. היתרון בגישה הזאת הוא הסרת הצורך בתקשורת בין מטלות, מכיוון שהמידע משותף, ולכ"א ישנה גישה אליו. מצד שני, ישנו קושי לשמור על לוקליות של זיכרון ברמת המעבד. כל מעבד עובד עם זיכרון מטמון, וכאשר מעבדים שונים עובדים על אותו הזיכרון, ישנו קושי לתחזק את המטמון הנ"ל, ותיווצר תקורה של שימוש בפס (Bus) המחבר בין הזיכרון למעבדים.

מודל החוטים (Threads)



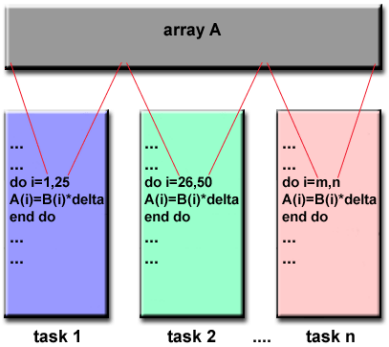
במודל הנ"ל, לתהליך מסוים יכולים להיות מספר מסלולי חישוב בו-זמנית. כאשר תהליך מתחיל לרוץ, הוא יכול ליצור חוטים, אשר משתפים ביניהם זיכרון, ובצורה זו לפצל את העבודה שלו, למספר תתי-עבודות, אשר בדר"כ יכולות לרוץ במקביל (אלא אם ישנן תלויות מידע). לכל חוטיהיה המידע הלוקלי שלו (מחסנית, לדוגמא), אולם ישתף זיכרון עם התהליך (אשר יוגדר להיות החוט הראשי), ועם החוטים האחרים. בצורה הזו, לכל חוט ישנה "ראיה גלובלית" של הזיכרון. החוטים מתקשרים בעזרת הזיכרון המשותף, וכאן נכנס הצורך בגישה מסונכרנת, על מנת למנוע התגשויות/מרוצי מידע. דוגמאות למימושים של המודל הנ"ל הינם *POSIX Threads* ו-*OpenMP*.

מודל העברת הודעות (MPI)



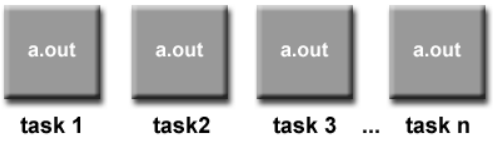
סט של מטלות, אשר לכל אחת זיכרון לוקאלי משלה. מטלות אינן מוגבלות למחשב יחיד, ויכולות להתפרס על מספר כלשהו של מחשבים/אשכולות. המטלות מעבירות מידע בעזרת תקשורת המבוטאת בשליחת וקבלת הודעות. בדר"כ, העברת מידע הינה פעולה מתואמת, לדוגמא, לכל שליחה מצד אחד, תהיה קבלה בצד השני. דוגמא למודל הנ"ל, הינו מנגנון ה*MPI*. (שנלמד בתרגולים)

מודל המידע המקבילי (מבוזר) (Distributed)



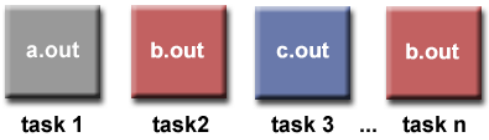
במודל הנ"ל מבוצעת עבודה על אוסף של מידע, אשר מאורגן במבנה נתונים ידוע מראש (אשר מאפשר גישה מקבילית/מסודרת). מצד שני, קבוצה של מטלות פועלות על אותו מבנה הנתונים, אך כל מטלה פועלת (/אחראית) על חלק אחר מהמבנה. ובנוסף, כל המטלות מבצעות את אותה הפעולה. במודלים של זיכרון משותף, כל המטלות ניגשות לאותו הזיכרון, בעוד שבמודלים של זיכרון מבוזר, מבנה הנתונים מחולק לחתיכות (Chunks), וכל מטלה מקבלת אחריות על כמות מסוימת של חתיכות.

מודל SPMD



מספר מעבדים יבצעו את אותה התוכנית בו-זמנית, כאשר בכל רגע נתון, כ"א מהמעבדים יכול להיות בחלק אחר של הקוד של התכונית. במודל הנ"ל, ישנו בדרך כלל שימוש במודלים לוגיים (לדוגמה, תנאים) על מנת להבטיח כי כל מעבד יבצע את החלק שהוא צריך, ולא יפעיל פשוט את כלל התוכנית. בנוסף, ישנה אפשרות כי כל מעבד ישתמש במידע שונה.

מודל MPMD



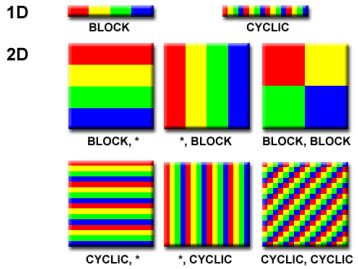
כהמשך למודל הקודם, כאן, כל מעבד יכול לבצע תוכנית שונה, או לבצע את אותה התוכנית אשר מעבד אחר מבצע. ובנוסף, ישנה כאן את אותה האפשרות שבה כל מעבד פועל על מידע שונה.

שני המודלים הנ"ל הינם דוגמא למודלי תכנות ב"רמה גבוהה" (High level programming), ומתבססים על שילוב של חלק מהמודלים השונים אשר תוארו לעיל.

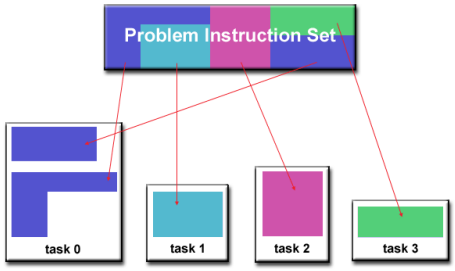
מאפיינים בבניית תוכניות מקביליות

חלוקה (Partitioning)

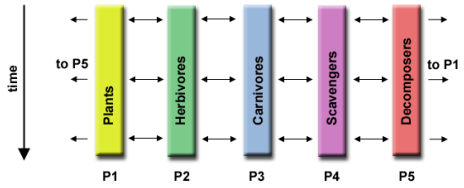
על מנת ליצור תוכנית מקבילית, יש צורך לחלק את המרחב לחתיכות, על מנת שניתן יהיה לעבד אותן בצורה מקבילית. ישנם שני סוגי חלוקה:



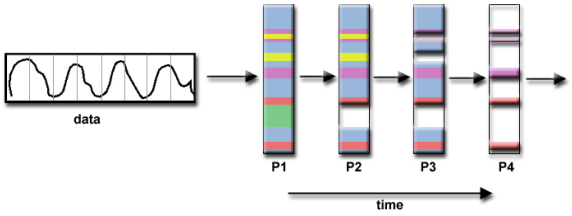
1) *חלוקת מרחב העבודה (Domain decomposition)[נקרא גם Data-parallel]* – המידע אשר מעובד ע"י התוכנית מחולק בין המטלות השונות. ישנן גישות שונות של חלוקה של מידע, בהתאם למבנה ולנגישות של המידע.



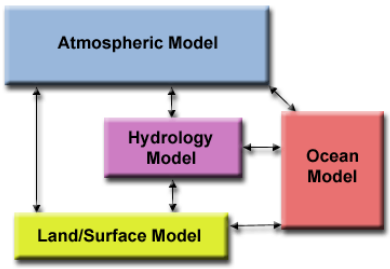
2) *חלוקה פונקציונלית (Functional decomposition)[נקרא גם Task parallel]* – החלוקה הנ"ל מתמקדת בחישוב אשר מתבצע, ולא במידע אשר עליו מתבצע החישוב. החישוב מתחלק בין המטלות השונות, בהתאם לכמות העבודה. ישנן גישות שונות חלוקה פונקציונלית:



- עבודה במודל "העברת מידע" בסמיכות. במודל הנ"ל, כל מטלה מתקשרת (מקבלת ומעבירה מידע) בעזרת קלט/פלט עם המטלות הסמוכות אליה.



- עבודה בשיטת קו-ייצור. כאשר כל מטלה מסיימת את החלק החישובי שלה, ומעבירה את הפלטים שלה כקלט למטלה הבאה.



- עבודה במודל "העברת מידע" כללית. במודל הנ"ל, בשונה מהמודל הראשון, ישנה תקשורת קלט/פלט בין כלל המטלות במערכת, וכל מטלה יכולה לתקשר עם מטלה אחרת.

תקשורת (Communication)

ישנם מספר דברים, אשר משפיעים על הצורך ועל כמות התקשורת:

מחיר התקשורת	• תקשורת פנימית במסגרת מטלה כלשהי בדר"כ מייצרת תקורה • מחזורי שעון משמשים לשליחת/קבלת מידע מאשר ביצוע חישובים • תקשורת בדר"כ דורשת סנכרון (Barriers) בין מטלות, ויכול להיווצר מצב שבו מטלות מחכות במקום לעבוד • עומס בתקשורת, ותחרות על משאבים, ישפיעו לרעה על הביצועים
רוחב פס מול השהייה	*רוחב פס (Bandwidth)* – כמות המידע שניתנת לשליחה/קבלה ביחידת זמן *השהייה (Latency)*– הזמן שלוקח לשלוח יחידת מידע מינימלית מ-A ל-B שליחת מספר גדול של הודעות "קטנות" יגרום להשהייה מרובה, וליצור תקורת תקשורת גבוהה מאוד. לעיתים, עדיף לקבץ את ההודעות להודעה גדולה יותר ומרוכזת, ובכך להעלות את רוחב הפס בין המטלות
נראות של תקשורת	• במודל העברת הודעות, השליחה והקבלה הינה תחת שליטת המתכנת ומבוצעת בצורה ישירה. • במודל המידע המבוזר, התקשורת מבוצעת באופן שקוף לעבודתו של המתכנת. התוכנית, בדר"כ, אינה יודעת שמבוצעת תקשורת פנימית בין תתי-המטלות שלה.
סנכרוניות מול א-סינכרוניות	• תקשורת סנכרונית דורשת "הסכמה" כלשהי בין המטלות (למשל, לכל שליחה יש קבלה מתאימה). הדבר יכול להיות ברמת המתכנת, או ברמה יותר נמוכה. • בדר"כ, תקשורת סנכרונית מוגדרת להיות תקשורת *חוסמת*, אשר מכניסה את החישובים להמתנה עד שהתקשורת תסתיים. • תקשורת א-סינכרונית מאפשרת שליחת הודעה והמשך עבודה ישר לאחר מכן (ללא "דאגה" לגבי קבלת ההודעה בצד השני). לכן, בדר"כ, תקשורת א-סינכרונית מוגדרת להיות תקשורת *לא חוסמת* • שילוב של חישוב ותקשורת הוא היתרון הגדול (והיחידי) בשימוש בתקשורת א-סינכרונית
טווח התקשורת	הידיעה אילו מטלות צריכות לתקשר אחת בין השנייה משפיעה על אופן התכנון של הקוד המקבילי. *Point-to-Point*– סוג תקשורת אשר לכל צד ידוע מי השולח ומי המקבל *Collective*– סוג תקשורת אשר בדר"כ מאפיין שליחה ליותר מגורם אחד, או שליחה קולקטיבית לכולם
יעילות התקשורת	• המתכנת צריך לקבל את ההחלטה הנכונה לגבי השימוש במודל המתאים. שכן, בהתאם למטלה, אולי יש עדיפות למודל הנתמך בתוכנה, מאשר אפשרויות מקביליות אשר נתמכות ע"י החומרה. • במקרים מסוימים, הבחירה הנכונה בין תקשורת סנכרונית או א-סינכרונית, יכולה להשפיע על היעילות של העברת ההודעות
ויש עוד... ועוד... ועוד...	

סנכרון (Synchronization)

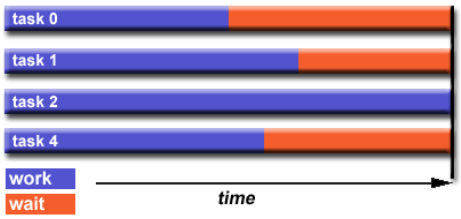
ישנם שלושה סוגי סנכרון מרכזיים:

1) *מחסום (Barrier)*– נקודה בעת הביצוע, אשר אליה צריכות להגיע כל המטלות המעורבות בפעולה (בדר"כ). כל מטלה מבצעת את קטע הקוד שלה, עד שהיא מגיעה למחסום, ואז נאלצת לחכות עד שכל המטלות האחרות יגיעו למחסום (נחסמת/נעצרת). כאשר המטלה האחרונה מגיעה למחסום, כל המטלות הינן מסונכרנות.
2) *מנעול/סמפור (Lock/Semaphore)*– מנגנון אשר ניתן להפעיל על מספר כלשהו של מטלות (לא בהכרח על כולן). המנגנון משמש בדר"כ על מנת להגן על גישה לזיכרון משותף, ומוגדרים קטעים קריטיים, אשר "מוגנים" בעזרת המנגנון הנ"ל. כאשר מטלה מסוימת נכנסת לביצוע, ולשימוש בזיכרון המשותף, עד שהיא לא סיימה לטפל בו ושחררה את המנגנון, אף מטלה אחרת לא יכולה לגשת ולפעול על הזיכרון הנ"ל. המנגנון הנ"ל יכול להיות חוסם (כל מטלה הרוצה לגשת, נכנסת להמתנה), ויכול להיות לא חוסם (המטלה פשוט ממשיכה, מבלי לגשת למידע המשותף).
3) *תקשורת מסונכרנת (Synchronous communication operations)*– מנגנון זה הוא בין מטלות אשר רוצות לתקשר אחת עם השנייה. כאשר מטלה מסוימת רוצה לתקשר עם מטלה אחרת, מבוצע תיאום בין שתיהן (למשל, המקבל "מודיע" שהוא מוכן לקבל הודעות).

תלויות מידע (Data Dependencies)

קיימת *תלות* בין פקודות בתוכנית, כאשר סדר הביצוע של הפקודות משפיע על תוצאת הפעלת התוכנית.
קיימת *תלות מידע* כאשר מטלות שונות מבצעות גישה/עדכון של אותו פריט מידע בזיכרון.
על מנת להתמודד עם תלויות מידע, בארכיטקטורות הפועלות בזיכרון מבוזר, יש להעביר את המידע הנדרש בנקודות הסנכרון. בארכיטקטורות הפועלות בזיכרון משותף, יש לסנכרן קריאות/כתיבות בין מטלות.

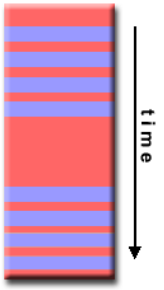
חלוקת עומס (Load Balancing)



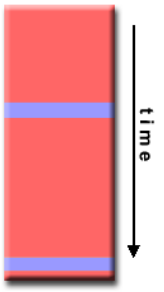
חלוקת העומס מתייחסת לחלוקת העבודה בין המטלות השונות, על מנת שכל המטלות יהיו "עסוקות" כל הזמן. זאת, במטרה להקטין את הזמן שבו הן בהמתנה. דוגמא לכך הינה השימוש במחסום. בהינתן מחסום, המטלה האיטית ביותר מגדירה את ביצועי המערכת.
על מנת להתמודד עם חלוקת עומס, ולייצר חלוקת עומס כמה שיותר אופטימלית, ניתן להשתמש בחלוקת עבודה דינאמית (בהתאם לזמן הפעולה של כל מטלה או כמות המידע), או חלוקה כמה שיותר זהה לחלקים (Chunks) עבור כ"א מהמטלות (לדוגמא, איטרציות על מבנה נתונים).

גרעיניות (Granularity)

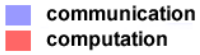
היחס בין כמות החישוב לכמות התקשורת בתוכנית מקבילית. בדר"כ, שלבי חישוב מופרדים משלבי התקשורת בעזרת פעולות/ארועי סנכרון.



- *מקבול ברמת גרעיניות עדינה(Fine-grain)*– כמות קטנה יחסית של חישובים מבוצעת בין פעולות תקשורת, הדבר גורר יחס נמוך של גרעיניות. כמו כן, מתאפשרת חלוקת עומס, עקב הכמות הנמוכה של החישובים בכל שלב. בתוכנית מסוג זה, ישנה תקורה גבוהה מאוד של תקשורת, ואפשרויות מוגבלות של אופטימיזציה. במקרים קיצוניים, בהם הגרעיניות עדינה עד מאוד, ייתכן כי תקורת הסנכרון והתקשורת תעלה על הזמן הדרוש לחישוב.



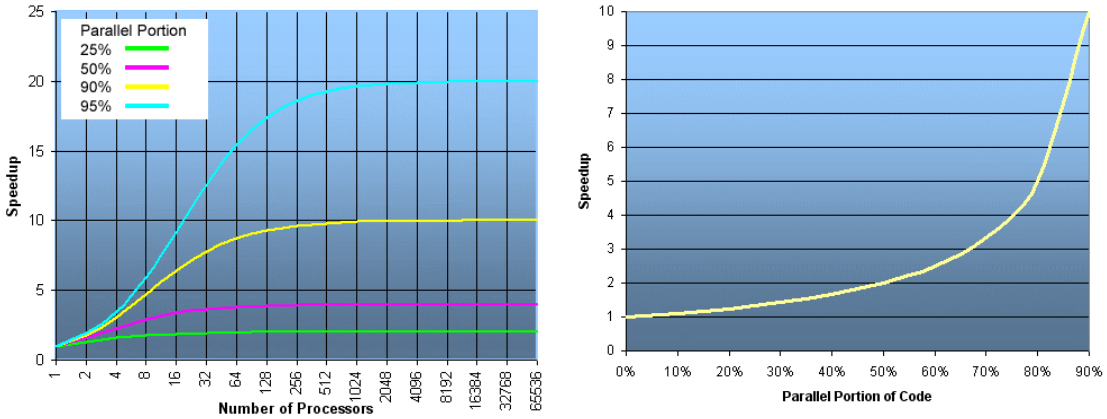
- *מקבול ברמה גרעינית גסה (Coarse-grain)*– כמות גדולה יחסית של חישובים מבוצעת בין פעולות תקשורת, הדר גורר יחס גבוה של גרעיניות. עקב הספק גדול של חישובים, ניתן לבצע אופטימיזציות על החישוב ולייעל את זמן הריצה של התוכנית. מצד שני, קשה לייצר חלוקת עומס מתאימה.



מגבלות התכנות המקבילי

חוק אמדל (Amdahl's Law)

חסם עליון על הפוטנציאל להאצה מאופיין ע"י גודל קטע הקוד אשר ניתן למקבול P, הינו $\frac{1}{1-P}$ ($P=0$ - הקוד לא ניתן למקבול. $P=1$ - כלל הקוד ניתן למקבול). כאשר נתון מספר המעבדים אשר מבצעים את קטע הקוד שניתן להאצה P (נסמן אותם ב-N), נקבל כי הפוטנציאל להתצאה הינו $\frac{1}{\frac{P}{N}+S}$, כאשר S הינו גודל קטע הקוד אשר נשאר סריאלי.



סיבוכיות (Complexity)

בדר"כ, תוכניות מקביליות מסובכות יותר מתוכניות סיריאליות בכמה סדרי גודל. מכיוון שלא רק שפעולות מבוצעות במספר ערוצים, ישנו גם מידע אשר מועבר ביניהן.

ניידות (Portability)

למרות שקיימים APIs כלליים לעבודה בתכנות מקבילי, לא כל שפות התכנות והגרסאות השונות שלהן תומכות ב-APIs הללו, ולכן אם נכתוב משהו ב-API אחד, ונצטרך להעביר אותו שפה, לא בהכרח נצליח לעשות זאת. בנוסף, לפעמים, השינויים אשר מוכנסים במימוש של ה-APIs הללו במקומות שונים, מקשים על הניידות, ומצריכים שינויים והתאמות. בנוסף, ארכיטקטורת החומרה והתמיכה במקביליות ברמה החומרה עלולה להשפיע על הניידות. ואסור לשכוח את העובדה כי מערכות ההפעלה הן אלו שמחליטות על מדיניות הזימון ואופן ההפעלה המקבילי, ולכן, לא ניתן להניח דבר, ולהסתמך על מאורעות/תוצאות במערכת הפעלה אחת, כאשר עובדים עם מערכת הפעלה אחרת.

דרישות משאבים (Resource Requirements)

המטרה של תכנות מקבילי היא להאיץ את זמן הריצה של תוכנית כלשהי, הדבר גורר שימוש מוגבר ב-CPU, ויכול להשפיע לרעה על צריכת המשאבים. בנוסף, בדר"כ, דרישות הזיכרון של תכנות מקבילי הינן גדולות יותר מאשר תכנות סיריאלי, ושכפולי המידע הנדרשים יכולים להשפיע על זמן הריצה. למעשה, יכול להיות כי עבור תוכניות מקביליות אשר פועלות זמן קצר, ריצה סיריאלית על אותה התוכנית היה מהירה יותר, עקב התקורה ביצירת סביבת ריצה מקבילית, אתחולים מתאימים והתקשורת הנדרשת.

יכולת התאמה (Scalability)

לפעמים, הוספה של מכונות/מעבדים/משאבי חישוב אינה מקצרת את זמן החישוב, אלא רק מגדילה אותו. זאת, כתוצאה מהתקורה הקיימת בתקשורת/גישה למידע וכו'. בנוסף, נכנסות כאן מגבלות חומרה, כדוגמת גודל רוחב הפס בין הזיכרון למעבדים, רוחב הפס של התקשורת בין המטלות, גודל הזיכרון אשר ניתן לכל מכונה, זמן המחזור וכו'. נוסף לאלו, ספריות אשר תומכות מתכנות מקבילי (לדוגמא *POSIX, OpenMP, MPI* ועוד), יכולות בעצמן ליצור מגבלות ללא קשר לאפליקציה.

זמן וירטואלי (Virtual time)

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/L2%20VirtualTime.ppt

הבעיה המוצגת כאן הינה בהתמודדות במודל מערכת אסינכרונית מבוזרת, אשר לא קיים בה זיכרון משותף, והתקשורת היא בעזרת שליחת הודעות בלבד. העיכוב בשליחת הודעה אינו ידוע, אולם אינו אפסי.
נבדיל בין שני סוגים של צופים:

- *צופה גלובלי*– רואה את מצב כל המערכת בנקודות כלשהן בזמן, ומסוגל "לצלם" את המצב הגלובלי של המערכת בכל זמן שהוא.
- *צופה לוקלי*– למעשה, אחד התהליכים במערכת. הוא רואה את המצב הלוקלי שלו, ויכול**להעריך** את המצב הגלובלי בעזרת אסיפת מצבים לוקליים של תהליכים אחרים.

▪ עקב אי היכולת של כל תהליך לדעת בודאות מה מצב המערכת, נוצרות בעיות סנכרון רבות, אשר דורשות התייחסות מיוחדת. דרך שגויה, למשל, הינה לשלוח הודעות על כל פעולה, אך פתרון זה פוגע בצורה קשה בגרעיניות, והופך אותה לעדינה עד מאוד.

נציע שלוש דרכי פתרון:

1. ביצוע סימולציה של מערכת סנכרונית. לדוגמא, כל אחד מהתהליכים יבצע פעולה, ויעצור. כל התהליכים יעצרו, ואז יתבצע צעד של סנכרון, וכן הלאה. תיווצר כאן תקורה גבוהה של סנכרון גלובלי לאחר כל פעולה.
2. סימולציה של המצב הגלובלי. צילום snapshots של המערכת בצורה א-סינכרונית, אשר לא יהיו בהכרח נכונים לאחר זמן מסוים, אולם עקביים עם המצב הלוקלי של כ"א מהתהליכים. הבנייה הזאת תתאפשר ע"י איסוף ההודעות שנשלחות בין כל התהליכים, ובניית תמונה גדולה יותר אשר מגדירה מצב בנקודה מסוימת. אולם, גם כאן, ישנה תקורה של אסיפת ההודעות, ובניית התמונה בעזרתן.
3. שימוש בשעון לוגי לוקלי, אשר ניתן להסיק ממנו מידע כלשהו לגבי המצב הגלובלי. המערכת תעבוד בצורה אסינכרונית, אולם אחריותו של כל תהליך לתחזק את השעון הלוגי שלו (ע"י קבלת ערכי שעון מתהליכים אחרים, ולהתעדכן על מצב כלשהו של המערכת בעזרתם, שיטה זו מכונה *Piggybacking*).

מאורע (event)– שינוי במצבו של תהליך.

נגדיר שלושה סוגים של מאורעות:

- *מאורע שליחה (send event)*– גורם לשליחה של הודעה
- *מאורע קבלה (receive event)* – גורם לקבלה של הודעה
- *מאורע פנימי (local event)*– גורם לשינוי המצב הפנימי

▪ במסגרת תהליך, כל ההודעות מתרחשות בזמן סדרתי, ולכל שליחה קיימת קבלה מתאימה.

יחס קרה-לפני Happened Before (Lamport)

נאמר שמאורע A ארע לפני מאורע B אם אחד משלושת הדברים הבאים מתקיים:

- A התרחש לפני B במסגרת אותו תהליך (סדר ביצוע פקודות בתהליך)
- A הוא שליחה ו-B הוא הקבלה המתאימה לה (שליחה-קבלה)
- קיים C, כך ש-A קרה לפני C, ו-C קרה לפני B (טרנזיטיביות)

- נאמר ששני מאורעות, A ו-B, הם *בלתי תלויים / מתרחשים בו-זמנית* אם A לא קרה לפני B וגם B לא קרה לפני A.
- נאמר ששתי דיאגרמות (אשר מייצגות סדר הפעלה בתוכנית) הינן *שקולות* אם *היחס קרה-לפני* בהם הוא זהה.
- בהינתן גרף של מאורעות, *מתך*C יוגדר להיות כחלוקה של הגרף לשני חלקים וקביעת נקודה בזמן, כאשר כל מה ששמאל לחתך מוגדר להיות "עבר" ומה שמימין לחתך מוגדר להיות "עתיד". חתך יהיה *עקבי (consistent)* אם לכל $a \in C$ (כלומר, כל מה ששמאל ל-C), וגם b קרה לפני a, אזי בהכרח $b \in C$. מכאן נסיק כי בהינתן חתך עקבי, כל מאורע של קבלה שנמצא משמאל לחתך, מאורע השליחה המתאים לו נמצא גם הוא משמאל לחתך.
- תוך שימוש במאורעות ובשעון הלוגי, כל התהליכים יתחילו מאופסים/מסונכרנים מבחינת השעון. על מנת לשמור על עקביות, כל תהליך יבצע את הפעולות הבאות:
  1) לפני שתהליך מבצע *מאורע פנימי* השעון של התהליך יקודם ב-1
  2) בכל עת שתהליך שולח הודעה, הוא מצמיד לה את השעון שלו (חותמת זמן)*[לפני השליחה עצמה, התהליך השולח מקדם את המונה שלו ב-1, כאילו זאת הייתה פעולה פנימית]*
  3) לפני שתהליך מקבל הודעה שנשלחה בזמן כלשהו, הוא משווה אותה לשעון שלו, ובוחר ביניהם את המקסימום, ובנוסף מקדם את השעון שלו ב-1. [הדבר מבוצע על מנת ליישר קו ביחס לשאר התהליכים, תוך התייחסות לתהליך שרץ הכי הרבה זמן]
- למרות השימוש בשעון הלוגי, בהינתן שני מאורעות אשר מתרחשים בו-זמנית, אין לנו אינפורמציה מספקת בהקשר לקיום יחס ה"קרה לפני", ובנוסף, לא ידוע האם שני המאורעות בלתי-תלויים. (האינפורמציה היחידה שכן ידועה לנו היא שאם מתקיים C(e) < C(e'), אזי בהכרח <u>לא</u> מתקיים e' < e)
- עקב הבעיה הזאת, ניגש לפתרון בשיטת *וקטור שעון (Vector Clock)* במקום *שעון לוגי*. לכל תהליך יהיה וקטור באורך מספר התהליכים במערכת C[], כאשר התא C[i] מתייחס לחותמת-הזמן של התהליך ה-i. כעת, כל תהליך צריך לבצע את הפעולות הבאות:
  1) לפני שתהליך i מבצע *מאורע פנימי* הוא מקדם ב-1 את וקטור השעון בתא ה-C[i]
  2) בכל עת שתהליך שולח הודעה, הוא מצמיד לה את וקטור השעון שלו (חותמת זמן)
  3) בעת קבלת הודעה, התהליך מעדכן את כלל הוקטור שלו, לכל תא j ערך התא מתעדכן להיות המקסימום בין ערכו העכשוי, לבין חותמת הזמן במקום ה-j בהודעה שנשלחה.*[בצורה זו, תהליך אשר מקבל הודעה מתהליך אחר, יכול לדעת מה קורה בשאר המערכת, ולקבל הערכה לגבי הזמנים של שאר התהליכים, ובכך ניתן לבדוק אי-תלות– לתהליך הנ"ל קוראים piggybacking].*
- Ci[i] >= Cj[i] – זאת מכיוון שרק התהליך ה-i יכול לעדכן את השעון שלו, וכל שאר התהליכים יכולים רק לקבל אינפורמציה לגביו.
- עתה ניתן להשוות בין וקטורי שעון, ולבדוק תלות:
  - u לא קרה אחרי v - $u \le v \Leftrightarrow \forall i, u[i] \le v[i]$
  - u קרה לפני v - $u < v \Leftrightarrow u \le v \wedge u \ne v$
  - u ו-v מתרחשים בו-זמנית - $u \| v \Leftrightarrow \neg (u < v) \wedge \neg (v < u)$

גילוי תנאי תחרות (Data race detection)

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/New%20L5-6%20DataRace.ppt

תחרות על מידע (Data race)– מצב שבו מתקיימות שתיים (או יותר) גישות בו-זמניות למקום משותף, כאשר לפחות אחת מהם היא לצורכי כתיבה.
ישנה בעיה בהתמודדות ומניעה תחרויות מסוג זה, מכיוון שהאחריות הינה על המתכנת. למרות שיש מנגנוני סנכרון והגנה כגון מנעולים, קטעים קריטיים וכו', המתכנת יכול לשכוח להפעיל סנכרון או לא להבין מהו המיקום הנכון להפעיל סנכרון, וכתוצאה מזה ישנו עודף של פעולות סנכרון שמעיק על פעולת המערכת.

תחרות על מידע גלויה לעין (Apparent Data Race)– מצב שבו מתקיימת תחרות על משאב אשר אין עליו סנכרון (מתקבל מידע לא מעודכן)

תחרות על מידע אפשרית (Feasible Data Race)– מצב שבו מתקיימת תחרות על משאב, כך שעקב זאת, המידע במשאב לא יהיה ידוע / לא מחייב.

Dijt+

- כלי, אשר "מתיישב" על התוכנה, ומטרתו למצוא Apparent Data Races. הכלי, בודק אם קיימות תלויות אמיתיות עבור הרצה מסוימת (לכן, על מנת לבדוק אם קיימות תלויות בתוכנית יש להריץ מספר אקספוננציאלי של פעמים). מבוצע שימוש ב*חס קרה-לפני של למפורט*, על מנת להגדיר מתי קיים Apparent Data Race בין A ל-B. כלומר, קיים ADR בין A ל-B אם הם מתרחשים בו-זמנית (כלומר, אם A לא קרה לפני B וגם B לא קרה לפני A). אחרת, הכלי יגדיר אותם כ*מסונכרנים*.
- מגדירים עבור כל חוט מסגרת-זמן *(LTF – Local Time-Frame)*, מסגרת-זמן חדשה (כלומר, קידום הערך הקודם) תהיה לאחר שחרור מנעול שנתפס. בהינתן שתי גישות לזיכרון משותף, גישהa מחוט A וגישה b מחוט B. אזי, אם a התקיימה בזמן Ta, <u>וקיים שחרור מנעול בחוט A אשר מתאים לתפיסת המנעול האחרונה שבאה לפני גישה b</u>, בזמן Tsync, נקבל כי *a קרה-לפני b* אמ"מ Ta < Tsync.
- לכל חוט יהיה וקטור LTF אשר מייצג את כלל ה-LTFs של החוטים במערכת. חוט מעדכן את הוקטור שלו בצורה של piggybacking, כלומר, כאשר יש שחרור של מנעול מחוט A ותפיסה של המנעול הנ"ל מחוט B, אזי מתבצע עדכון של הוקטור וחוט B מקבל את האינפורמציה מחוט A.
- בדיקת מרוץ בין a ל-b מתבצעת ע"י הנוסחא
$(a.type = write \vee b.type = write) \wedge (a.ltf \geq b.timestamp[a.thread\_id])$. כאשר, אם הנוסחא מחזירה TRUE, קיים מרוץ בין a ל-b.(מתוך הנחה כי קיים log כללי של המערכת, ובו a נכתב לפני b, וכן כי ה-log שומר על sequential consistency).
- מכיוון שמספיק לתעד את הקריאה והכתיבה הראשונות למשתנה בכל LTF, ומספיק לבדוק את הגישה הנוכחית אל מול "הגישות האחרונות" של שאר החוטים למשתנה, עבור כל משתנה נשמר מערך של היסטוריית גישות (יהיה תא קריאה ותא כתיבה עבור כל חוט במערכת). כאשר חוט ניגש בפעם הראשונה לקרוא/לכתוב למשתנה, הוא מעדכן את השדה המתאים אליו (לכל חוט יש תא תואם). בעת קריאה, החוט בודק את כל הכתיבות האחרונות של החוטים האחרים למשתנה. בעת כתיבה, החוט בדוק את כל הכתיבות והקריאות האחרונות של החוטים האחרים למשתנה.

יתרונות	חסרונות
אין אזעקות שווא בנוגע למרוצים פיקטיביים	רגישות יתר לשינויים במדיניות הזימון
אין החמצה של מרוצי מידע בהרצות מסוימות (תלוי במדיניות הזימון –scheduling)	דורש מספר רב (אקספוננציאלי) של ריצות, ואפילו במקרה הזה, לא יכול להבטיח שהתוכנית חסימה ממרוצי מידע

Lockset

Lockset עובד לפי גישת נעילה (Locking discipline) אשר דורשת כי כל משתנה משותף יהיה מוגן ע"י מנעול היוצר מניעה הדדית. אלגוריתם Lockset מאתר הפרות של הגישה הנ"ל. הבעיה הראשית שלו הינה מספר רב של אזעקות שווא.

האלגוריתם הבסיסי:

- לכל משתנה v, יהיה סט C(v) אשר מגדיר את מספר המנעולים אשר מגנים על v מתחילת החישוב עד לנקודה ספציפית.
- Locks_held(t) בנקודה, יוגדר להיות סט המנעולים שחוט t מחזיק כעת. בעת אתחול, C(v) יוגדר להיות כל המנעולים הקיימים במערכת.
- בכל גישה של חוט t למשתנה v, נעדכן את C(v) בצורה הבאה :
  - $C(v) = C(v) \cap locks\_held(t)$.
  - אם נקבל כי $C(v) = \phi$, האלגוריתם יוציא אזהרה.

- ניתן לבצע שיפור במסגרת אותו החוט, ולבדוק האם יש גישות a ו-b באותו החוט (a לפני b) ושתי הגישות באותה מסגרת זמן (LTF), אז במקום החיתוך נבדוק אם מתקיים $locks_a(v) \subseteq locks_b(v)$.
ובנוסף, בכל LTF תיבדק רק הגישה הראשונה, מכיוון שכלל הגישות באותו LTF הינן במסגרת אותה נעילה (במקרה הנ"ל, Lockset יכול להיעזר באותו מנגנון Logging שנפגשנו בו ב-Dijt+).

יתרונות	חסרונות
פחות רגיש למדיניות תזמון (Scheduling)	מספר עצום של אזעקות שווא
מוצא את קבוצת כל מרוצי-המידע האפשריים במערכת – אף מרוץ מידע לא מפוספס	עדין תלוי במדיניות הזימון
קיים כלי מעשי (Eraser) אשר פועל לפי העקרון הנ"ל	בכל זאת, לא ניתן להוכיח שהתוכנית עצמה חסרת מרוצי-מידע (בכל מקרה, דורש מספר רב של הרצות בדומה ל-Dijt+)

שיפור אלגוריתם Lockset

אלגוריתם Lockset, יחד עם גישת הנעילה, הינם מחמירים מדי. קיימים שלושה שימושים נפוצים בתכנות אשר מפרים את השיטה, אולם אין בהם כלל מרוצי-מידע:

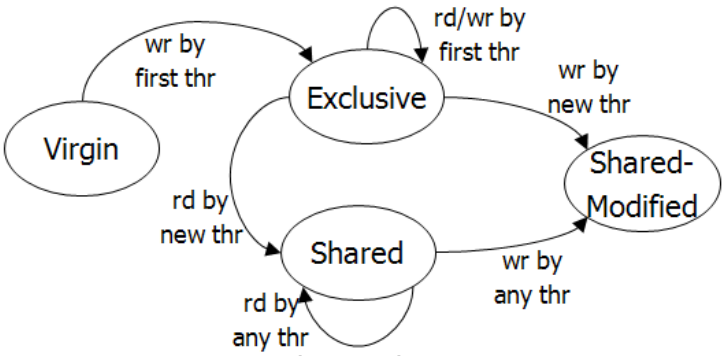
1. *אתחול(Initialization)* – משתנים משותפים בדר"כ מאותחלים ללא תפיסת מנעול
2. *מידע משותף לקריאה בלבד (Read-Shared Data)*– משתנים מסוימים מקבלים ערך רק באתחול, ומוגדרים להיות קריאה-בלבד לאחר מכן
3. *קוראים-כותבים (Read-Write locks)*– מנעולי קוראים-כותבים מאפשרים למספר קוראים לגשת למשתנה משותף בו-זמנית, אך רק לכותב אחד.

בהמשך יוצגו פתרונות ושינויים על מנת להתמודד עם שלושת אלו.

אתחול– כאשר מידע חדש מאותחל, אין צורך לנעול אותו, כיוון שחוטים אחרים לא מחזיקים אליו רפרנס עדיין. אולם, לא תמיד ניתן לדעת מתי האתחול הסתיים. לכן, נגדיר כי משתנה משותף הוא מאותחל כאשר יש אליו *גישה ראשונה מחוט אחר* (שהוא לא החוט המקורי שבו הוא אותחל). כל עוד הגישה למשתנה היא רק מהחוט אשר אתחל אותו, הקריאות והכתיבות לא יעדכנו את C(v).

מידע משותף לקריאה בלבד– אין צורך להגן על משתנה אם הוא הוגדר לקריאה בלבד. לכן, מרוצי מידע ידווחו רק כאשר משתנה שאותחל הופך למשותף ופתוח לכתיבה ע"י יותר מחוט אחד.

נגדיר עבור כל משתנה ארבעה מצבים:



1. **Virgin**–משתנים מתחילים מהמצב הנ"ל, הוא מגדיר כי אין עדיין רפרנס מאף חוט למשתנה הנ"ל.
2. **Exclusive**– נכנסים אליו כאשר בוצעה גישה למשתנה בפעם הראשונה (ע"י חוט יחיד). כל הגישות הבאות ע"י אותו החוט (בתנאי שזה רק החוט הנ"ל) לא מעדכנות את C(v)– מצב זה מטפל באתחול
3. **Shared**– נכנסים אליו בגישה ראשונה *לצורך קריאה* מצד חוט אחר. ישנו עדכון של C(v), אולם במצב זה לא ידווחו מרוצי-מידע. בצורה כזו, מספר חוטים יכולים לגשת למשתנה לצורכי קריאה – מטפל במידע משותף לקריאה בלבד
4. **Shared-Modified**– נכנסים אליו כאשר יותר מחוט אחד ניגש למשתנה *לצורך כתיבה*. במצב הנ"ל C(v) מעודכן, ומדווחים מרוצי-מידע כמוגדר באלגוריתם.

קוראים-כותבים– האלגוריתם הבסיסי לא תומך בצורה נכונה בסנכרון הנדרש. (*הגדרה: עבור משתנה v, מנעול m מגן על v אם m מוגדר במצב כתיבה עבור כל כתיבה ל-v, ומוגדר במצב כלשהו [קריאה או כתיבה] עבור כל קריאה של v*). על מנת להתאים את האלגוריתם, נגדיר שתי קבוצות ונשנה את האלגוריתם, כך שיפעל בצורה הבאה במצב Shared-Modified:

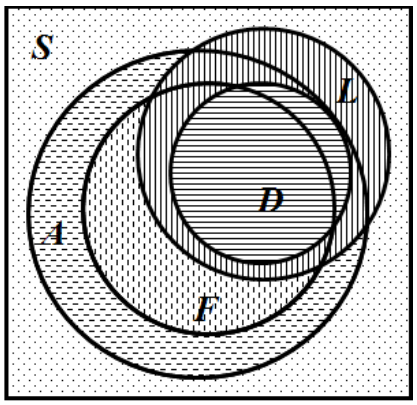
- Locks_held(t)– סט המנעולים המוחזק ע"י החוט t בנקודה מסוימת
- Write_locks_held(t)– סט המנעולים המוחזק ע"י החוט t בנקודה מסוימת *במצב כתיבה*

האלגוריתם המשופר

- לכל משתנה v, נאתחל את C(v) להיות סט כל המנעולים האפשריים
- בכל קריאה ל-v ע"י חוט t נבצע:
  - $C(v) = C(v) \cap locks\_held(t)$
  - אם $C(v) = \phi$, נדווח
- בכל כתיבה ל-v ע"י חוט t נבצע:
  - $C(v) = C(v) \cap write\_locks\_held(t)$
  - אם $C(v) = \phi$, נדווח

שילוב בין Dijt+ ל-Lockset

התרשים מייצג את כלל המשתנים בתוכנית מסוימת, ואת טווחי הזיהוי של שני האלגוריתמים.



- S– כלל המשתנים המשותפים בתוכנית
- A– כל מיקומי ה-Apparent data races
- F– כל מרוצי המידע בתוכנית
- L– כל ההפרות אשר נמצאו ע"י Lockset
- D– כל מרוצי המידע אשר נמצאו ע"י Dijt+

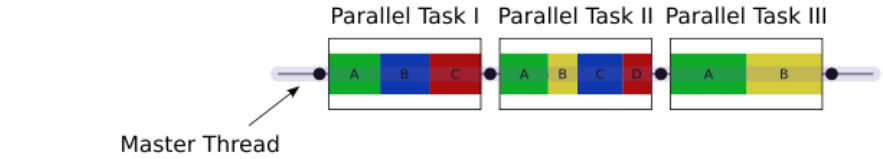
- אלגוריתם Lockset מסוגל לזהות יותר מרוצי-מידע חשודים בהרצות תוכנית שונות, בעוד ש-Dijt+ יכול לסנן את ההתרעות המיותרות של Lockset. מצד שני, Lockset יכול להוריד את מספר הבדיקות אשר נעשות ע"י Dijt+ ע"י כך שעבור כל $C(v) \neq \phi$, Dijt+ לא יבדוק אם קיימים מרוצי מידע על v.

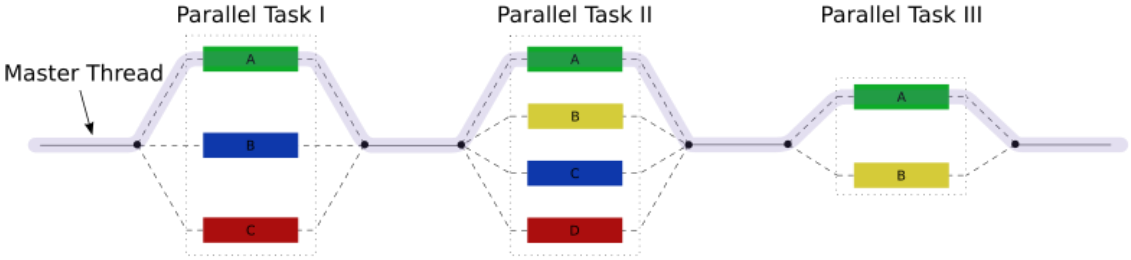
OpenMP

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/OpenMPLecture.ppt

http://en.wikipedia.org/wiki/Openmp

- ספריה, שהיא הרחבה של השפה, ומאפשרת ביצוע חישובים איטרטיביים בצורה מקבילית (לולאות for). פעולות הסנכרון והעברת ההודעות מבוצעות מאחורי הקלעים, כאשר המתכנת יכול לקבל אינפורמציה על כמות המשאבים הזמינה, ולהפעיל את החישוב המקבילי בהתאם, בעזרת הודעות לקומפיילר (pragma). בלב המודל נמצא חוט "מנהל", אשר בהינתן הוראה לתחילת קטע מקבילי, מייצר כמות מוגדרת של חוטים, ומריץ דרכם את קטע החישוב המקבילי.





- החוטים מתקשרים ע"י שימוש במשתנים משותפים. כל משתנה שיש אליו גישה ליותר מחוט אחד, מוגדר להיות משותף, אחר מוגדר להיות פרטי.
- פעולות ופקודות אפשריות ב-OpenMP (אפשרויות שונות שניתן להכניס לאחר pragma omp#):

חלוקת עבודה	
omp for / omp do	חלוקה של איטרציות בין החוטים השונים
sections	הגדרת רצף בלוקים בלתי-תלוי לחוטים שונים
single	הגדרת בלוק אשר יבוצע ע"י חוט יחיד (בסוף הבלוק צריך להיות מחסום)
master	הגדרת בלוק אשר יבוצע רק ע"י החוט הראשי (בסוף הבלוק צריך להיות מחסום)
שיתוף משתנים	
shared(X)	X משותף בין כל החוטים בו-זמנית. כברירת מחדל, כל המשתנים בתוך החלק המקבילי משותפים פרט למונה הלולאה
private(X)	נוצר עותק של X עבור כל אחד מהחוטים. משתנה פרטי לא יאותחל וערכו לא יישמר לאחר היציאה מהקטע המקבילי (כברירת מחדל, מוני לולאות ומשתנים לוקליים אשר נמצאים על המחסנית של קריאות לפונקציות מחלקים מקביליים יוגדרו להיות פרטיים)
firstprivate(X)	זהה ל-private(X), רק שיאותחל לערכו המקורי לפני הכניסה לחלק המקבילי
lastprivate(X)	זהה ל-private(X), רק שערכו יישמר לאחר היציאה מהחלק המקבילי
threadprivate(X)	X יהיה משתנה גלובלי פרטי עבור כל חוט, הוא לא ייווצר מחדש בכל קטע מקבילי, אלא יישאר עבור כל חוט

copyin(X)	ערכו של X מועתק מהערך של החוט הראשי
סנכרון	
critical	קטע הקוד יבוצע כל פעם ע"י חוט אחד, ולא יהיה ביצוע שלו במקביל (שימושי למניעת מרוצי-מידע)
atomic	הפעולה דומה ל-critical section, אך מציעה לקומפיילר להשתמש בתמיכת חומרה (פעולות אטומיות שניתנות ע"י הארכיטקטורה) על מנת לייעל את הביצועים
ordered	הבלוק יבוצע באותו סדר האיטרציות כפי שהיה מצופה בריצה סיריאלית
barrier	כל חוט המגיע למחסום, נכנס להמתנה עד ששאר החוטים יגיעו לאותה הנקודה.
nowait	הפקודה מגדירה כי כל חוט אשר הגיעה לנקודה הנ"ל יכול להמשיך מבלי לחכות לחוטים האחרים. ללא הפקודה הנ"ל, החוטים ייתקלו במחסום אוטומטי בסוף הקטע המקבילי
schedule(type,chunk)/תזמון/מדיניות זימון	
static	בשיטה הנ"ל, האיטרציות יחולקו בין החוטים מראש. החלוקה מתבצעת בצורה שווה, אולם ניתן לשלוט בגודל החתיכה (chunk) של איטרציות עוקבות שייקבל כ"א מהחוטים. (מתאים למספר קטן יחסית של איטרציות)
dynamic	בשיטה הנ"ל, חלק מהאיטרציות מחולקות בין מספר קטן יותר של חוטים. אם חוט מסוים מסיים את האיטרציה שהוקצבה לו, הוא מקבל איטרציה חדשה מאלו שנשארו. (מתאים למספר גדול יחסית של איטרציות)
guided	בשיטה הנ"ל, בדומה לשיטה הדינאמית, חתיכה "גדולה" של איטרציות עוקבות ניתנת לכל חוט, וגודל החתיכות קטן אקספוננציאלית, עד שמגיע לגודל שמוגדר ב-chunk.
שליטה בגרעיניות	
if(exp)	החוטים יבצעו את הפעולה במקביל אם התנאי יתקיים, ניתן בצורה זו לשלוט ולבטל את המקבול כאשר אין בו צורך
num_threads(exp)	מאפשר שליטה במספר החוטים אשר יפעלו
אחרים	
reduction(op|X)	המשתנה X הינו לוקלי אצל כל חוט, אולם בסוף האיטרציה הערכים של כל חוט נסכמים (reduced) למשתנה גלובלי משותף. על מנת למנוע מרוצי מידע, מתבצע עדכון (סכימה) של המשתנה לפי סדר מוגדר מראש.
flush(X)	מחיקת הערך ואתחולו לערכו מחוץ לקטע המקבילי
פקודות מידע(לא במסגרת pragma#) [קיימות בנוסף פעולות המשתמשות במנעולים]	
omp_get_thread_num	מחזיר את המספר הסידורי של החוט
omp_get_num_threads	מחזיר את מספר החוטים בקטע המקבילי

חלוקת עומס וגרעיניות של parallel for#

- אם כל האיטרציות מבוצעות בזמן שווה, השימוש במעבדים הוא אופטימלי. אולם, אם קיימות איטרציות קצרות יותר מהאחרות, חלק מהמעבדים יהיו במצבidle בזמן החלק המקבילי. בנוסף, אנו לא תמיד יודעים מהי חלוקת העבודה, ולפעמים יש צורך בחלוקה דינאמית.
- בתחילת כל קטע מקבילי, יצירת החוטים והסנכרונים ביניהם (בין לבין) צורכת זמן. בנוסף, האפשרות של חלוקת עבודה לחוטים בכל איטרציה יכולה ליצור תקורה גבוהה יותר מזמן החישוב. נוסף לאלו, יש צורך להתגבר על התקורה של שימוש בחוטים ע"י שימוש בגודל חכם של חתיכות.
- קיים ב-OpenMP טרייד-אוף בין שני המושגים הנ"ל, וניתן "לשחק" ולהתאים אותם לפי אפשרויות חלוקת העבודה השונות הקיימות.

יתרונות	חסרונות
פשטות (אין צורך להתמודד עם שליחת הודעות)	פועל ביעילות רק על פלטפורמות מרובות-ליבות עם זיכרון משותף
חלוקת המידע והאחריות היא אוטומטית ומכוונת הוראות	דורש קומפיילר תומך OpenMP
אין צורך לשנות את הקוד בצורה דרמטית	יכולת ההתאמה (סקלביליות) מוגבלת לארכיטקטורת הזיכרון
קוד זהה לאפליקציות מקביליות וסיריאליות. הגדרות OpenMp מוגדרות כהערות בקומפיילרים סיריאליים	אין ניהול שגיאות
הצורך בשינוי מינימלי של הקוד מקטין את כמות הבאגים	חסר מנגנון ניהול מיפוי חוט-מעבד
ניתנת אפשרות למקבול בגרעיניות עדינה ובגרעיניות גסה	לא ניתן לשימוש על GPU

TBB (Thread Building Blocks)

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/Task-based-data-parallel-programming-tbb-1fpp.zip

- ספריית C++ שפותחה ע"י אינטל, על מנת לנצל מעבדים מרובי-ליבות. הספרייה מכילה מבני נתונים ואלגוריתמים גנריים, ומתפקדת גם כספריית זמן-ריצה, ו"מחליטה" על מספר החוטים האופטימלי, גרעיניות המטלות ומדיניות הזימון, תוך שמירה על חלוקת עומס אוטומטית ויעילות בשימוש בזיכרון מטמון.

אלגוריתמים מקביליים גנריים לעיבוד מידע ב-TBB

מקבול לולאות	
parallel_for	הפעלה מקבילית (בחלוקת עומס מתאימה) של מספר סופי של איטרציות ב"ת
parallel_while	
parallel_scan	ביצוע פעולת רקודציה על רישא, כלומר, עבור תהליך i תבוצע רדוקציה של הערכים מ-0 ועד ל-i (כמו scan של MPI)
מקבול רצפים של מידע	
parallel_do	לשימוש על רצף או "ערמה" של עבודה/מידע
pipeline	חלוקה למספר תחנות, כאשר כל תחנה יכולה להיות מקבילית או סיריאלית (משתמש בצורה יעילה בזיכרון המטמון)
מיון מקבילי	
parallel_sort	חלוקה לתת-מטלות אשר ניתנות לביצעו בו-זמנית, בסיבוכיות O(NlogN)

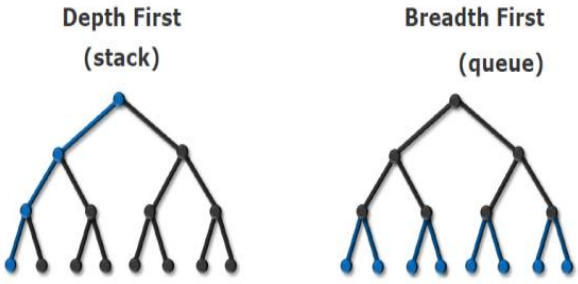
- על מנת להתמודד עם התקורה של שימוש בחוטים, גודל החתיכות (chunks) מנוהל ע"י *partitioner* אשר מוגדר כחלק מה-TBB. ישנם שלושה סוגים:
  - *simple_partitioner*– משתמש בגודל אשר ניתן בצורה ידנית
  - *auto_partitioner*– מתאים את הגודל בצורה היוריסטית
  - *affinity_partitioner*– מנסה לשחזר את הריצה האחרונה של האלגוריתם (במהלך הריצה, נשמר מידע בין האיטרציות)

מטלות ומדיניות זימון

- מתזמן המטלות (Task Scheduler) הוא האחראי לנהל את מאגר החוטים ולמפות את המטלות לחוטים. אלגוריתמים מקביליים מסתמכים על המתזמן הנ"ל, והוא מצד שני, אחראי לפתור בעיות אשר עלולות לעלות בזמן הפעלת האלגוריתם המקבילי.

בעיה	גישה לפתרון
מינוי-יתר (Oversubscirption)	חוט מתזמן יחיד לכל חוט חומרה
זימון הוגן	זימון לא הוגן ללא הפקעה
תקורה גבוהה	המתכנת מייצר מטלות, לא חוטים
חלוקת עומס לא שווה	"גניבת מטלות" מאזנת את העומס

- תלויות בין מטלות מגדירות את סדר הביצוע. בדר"כ, הסדר מגדיר עץ, כאשר השורש שלו מייצג את תחילת החישוב. כחלק ממנגנון הסנכרון, מטלה יכולה להמתין עד שהבנים שלה סיימו (למרות שהחוט המבצע של המטלה לא נחסם. במצב זה, חוט יכול להמשיך "לגנוב מטלות" בזמן שהמטלה הנוכחית בהמתנה), הדבר גורם לכך שהמטלה מעל (ההורה) לא יכולה להמשיך לעבוד וגם היא נחסמת. מצד שני, מטלה יכולה *להצהיר* שהיא ממתינה לסיום כל הבנים שלה, ולאחר שהם סיימו היא תמשיך לבצע את קטע הקוד שהוגדר מראש, ובינתיים, המטלה מעל (ההורה) יכולה להמשיך את העבודה שלה.
- כזכור, על מנת לשמור על חלוקת עומס שווה, מבוצעת "גניבת מטלות". הגניבה מבוצעת ב-BFS (עבודה ב-BFS היא עבודה על מרחב גדול, ולוקחת יותר זמן. כתוצאה מכך יש פגיעה בלוקליות זיכרון המטמון, אולם המקביליות גדולה יותר מכיוון שיש יותר מטלות/פיסות מידע לעבד), בעוד שחלוקת העבודה(מטלות) מבוצעת ב-DFS (עבודה ב-DFS היא עבודה על מרחב קטן, ועקב אופי הריצה ניתן להיעזר בלוקליות של זיכרון המטמון, אולם כמעט ואין אפשרות למקבל).



- בנוסף, TBB מכילה קונטיינרים (כמו STL) על מנת לשמור ולעבד מידע בצורה מקבילית (למשל וקטור, תור, או טבלת-מיפוי). וכן, גם סט של מנעולים, אשר חלקם תלויים מערכת הפעלה, וחלקם של הספרייה.

השוואה בין TBB ל-OpenMP

	TBB	OpenMP
מאגר חוטים	אחד לכל תהליך	נוצר בכל #pragma
שפות תומכות	C++	C, C++, Fortran
תמיכת קומפיילר	ספריית C++	דורש תמיכת קומפיילר
ישימות (applicability)		מתאים בדר"כ לקוד הקשור לחישוב מדעי הכולל לולאה גדולה כאשר ברור מה אפשר למקבל
מקבול מקונן	מובנה	קשה? (אפילו לא קיים עד גרסא 3.0)
איזון חוטים (Thread Balance)	מתבצעת גניבת מטלות. החוט שממנו נגנבת המטלה לא מודע לכך, לכן זה לא משפיע עליו. בנוסף, הגניבה מבוצעת כפעולה אטומית ללא מנעולים	Static, Dynamic, Guided שתי האחרונות משתמשות במנהל עבודה על מנת לקבל מטלות, לכן, במערכות מרובות-מעבדים יש צורך לגשת אליו בצורה סיריאלית

עקביות של משתנים משותפים במסגרת מודל זיכרון משותף (Shared memory – Consistency of shared variables)

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/L4%20SequentialConsistency-new.ppt

מודל עקביות זיכרון (Memory Consistency Model) הוא *הסכם* בין סביבת זמן-הריצה (חומרה, מערכת הפעלה, וכו') לבין התהליכים. סביבת זמן-הריצה מבטיחה לאפליקציה תכונות מסוימות בנוגע לאופן כתיבת הערכים למשתנים משותפים והקריאה שלהם. במסגרת המודל, נקבע מה הם סדרי פעולות חוקיים, ומהם לא.

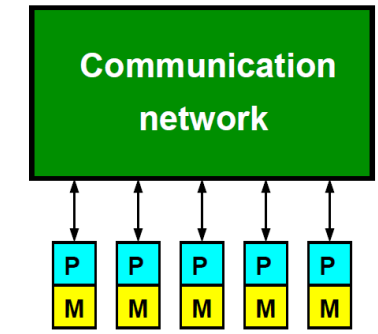
מודל זיכרון עקבי-קוהרנטי (Coherence) הוא מודל זיכרון, שבו סביבת זמן-הריצה מבטיחה שהכתיבות שבוצעו ע"י התהליכים השונים למשתנה מסוים, נראות ברמת התהליכים באותו *סדר מלא*.
סדר התוכנית (Program Order)– הסדר שבו הפקודות מופיעות בכל תהליך. זהו *סדר חלקי* של כל הפקודות של התוכנית.
סיריאליזציה (Serialization) - *יחס סדר מלא* של כל פקודות הקריאה/כתיבה של כל התהליכים, אשר עקבי עם סדר התוכנית.
סיריאליזציה חוקית– סיריאליזציה אשר לכל קריאה של משתנה X מחזירה את הערך שנכתב ע"י פקודת הכתיבה האחרונה ל-X*בסדר המלא*.
בהינתן תוכנית P, ו-Px הוא קטע התוכנית שלה אשר מכיל את כל פקודות הקריאה והכתיבה ל-X (ורק הן), אזי נגדיר:
עקביות-קוהרנטיות (Coherence)–P תהיה קוהרנטית אם לכל משתנה X קיימת סיריאליזציה חוקית.

עקביות סדרתית (Sequential Consistency) לפי למפורט הוא מודל זיכרון אשר בו <u>כל</u> הקריאות/כתיבות ע"י כל התהליכים נראות ע"י התהליכים באותו *סדר מלא. [בשונה מהמודל הקודם, המודל הנ"ל דורש הגבלה גבוהה יותר. במודל הקודם, נדרש למצוא סדר מלא עבור כל משתנה בנפרד. עתה, במודל הנ"ל, צריך למצוא סדר מלא עבור כל המשתנים הקיימים בתוכנית יחדיו]*
עקביות סדרתית–תוכנית P תוגדר להית עקבית סדרתית אם קיימת סיריאליזציה חוקית של כל פקודות הקריאה/כתיבה ב-P.

- ניתן לראות כי תוכנית שהיא עקבית סדרתית, היא גם עקבית-קוהרנטית. בנוסף, הדרישות של העקביות הסדרתית גבוהות יותר, ולכן היא מוגדרת להיות *חזקה* יותר.
- באופן כללי, מודל זיכרון עקבי A יהיה *חזק יותר (Strictly Stronger)* ממודל B, אם כל ההרצות תחת A יהיו חוקיות תחת B.
- חשוב לציין שהשמירה על עקביות היא משימה כבדה, אשר מצריכה מציאת סדר הרצה נכון, וזה יכול לפגוע בביצועים. בנוסף, קיימים מקרים שבהם הקומפיילר לא יכול לבצע אופטימיזציות מסוימות, שכן, אם יבצע את אלו, תיפגע הקוהרנטיות.

BSP – The Bulk Synchronous Parallel Model

http://www.math.uu.nl/people/bisselin/pas1.html

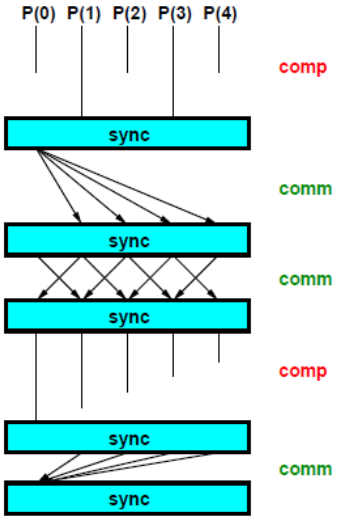


Bulk synchronous parallel (BSP) computer.
Proposed by Leslie Valiant, 1989.

מודל מחשב BSP

המודל מכיל אוסף של מעבדים, כאשר לכ"א יש זיכרון משל עצמו. בנוסף, זהו מחשב אשר עובר בשיטת זיכרון מבוזר. לכל מעבד, הגישה לזיכרון העצמי היא מהירה, בעוד שהגישה לזיכרון מרוחק היא איטית יותר (זמן הגישה לזיכרון מרוחק הוא זהה עבור כל המעבדים). במודל, אין צורך לדאוג להתעסקות עם רשת התקשורת, אלא רק בביצועים הגלובליים. האלגוריתמים שנבנים עבור המודל הינם פורטביליים, וניתנים לריצה בזמן יעיל על מודלי מחשבים מקביליים אחרים.

אלגוריתם BSP



אלגוריתם BSP מחולק לסדרה של צעדי-על, אשר מחולקים לשניים:

1. *צעד חישוב (computation superstep)*– בצעד הנ"ל מבוצעות כל הפעולות החישוביות.
2. *צעד תקשורת/סנכרון (communication superstep)*– בצעד הנ"ל מבוצעת העברת המידע בין המעבדים השונים.

h-relationמוגדרת להיות צעד תקשורת אשר בו כל תהליך שולח/מקבל לכל היותרh מילות מידע.
$h = \max\{h_{send}, h_{receive}\}$. (לצורך חישוב h-relation נקח את המקסימום מבין כל המעבדים המשתתפים)

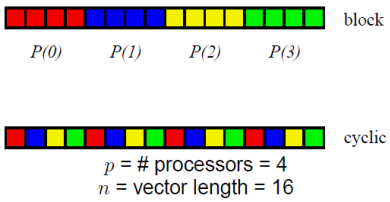
משתנים אשר BSP תלוי בהם

- **p**– מספר המעבדים
- **g**–מחיר השליחה של פיסת מידע (flops)
- **r**–כמות החישוב בהתאם למספר המעבדים (Mflops)
- **l**–עלות הסנכרון הגלובלית (flops) [מתייחס לתקורה של בניית הסביבה לשליחת מידע, וידוא כי המידע הגיע ליעדו, והעלות של מנגנון הסנכרון עצמו].

מחיר של צעד תקשורת מוגדר להיות $T(h) = hg + l$

מחיר של צעד חישוב מוגדר להיות $T = w + l$, כאשר w מוגדר להיות המספר המקסימלי של flops אשר לקח למעבד כלשהו בצעד החישוב (תמיד לוקחים את המקסימום, מכיוון שצעד הסנכרון הוא סוג של מחסום, שכל המעבדים צריכים להגיע אליו. ואם מעבד מגיע אליו, הוא עובר למצבidle, עד שכל שאר המעבדים הגיעו)

- באלגוריתם BSP חשוב להתייחס לשיטת החלוקה, על מנת לאפשר לוקליות של הזיכרון ולהקטין את הצורך בצעדי הסנכרון בין צעדי החישוב.



- מודל BSP מתאים מאוד למחשבי/תוכניות SPMD (Single Program, Multiple Data). הפלט של התוכנית הינו יחיד, אולם כל המעבדים מריצים את אותו הקוד, אך על פיסות מידע שונות. במודל הנ"ל, בדר"כ, נגדיר את המעבד אשר מקבל את התשובות החלקיות ומחזיר תשובה סופית כמעבד 0, וכל שאר המעבדים יקבלו את המידע הרלוונטי, יעבדו אותו בהתאם, וישלחו חזרה למעבד0 (או אחד לשני, באם אלו חישובי ביניים).

מבנה ופקודות בתוכנית BSP

- תוכנית BSP תתחיל תמיד בקטע סדרתי, זאת מכיוון שלפעמים מספר המעבדים תלוי בקלט, וכן, עצם קבלת הקלט עבור החישוב. בנוסף, תוכנית BSP תסתיים תמיד בקטע סדרתי, עקב הוצאת פלט (רק מעבד יחיד יוציא את הפלט). את הקטע הסדרתי, יבצע בפועל רק מעבד0, כאשר כל שאר המעבדים "יקפצו" מעל הקטע הנ"ל, ויתחילו (למעשה, ימתינו למעבד0) בביצוע הקטע המקבילי.
מעבד 0 "יורש" את כל המשתנים והערכים שלהם מהקטע הסדרתי, ויכול להשתמש בהם בקטע המקבילי (ורק הוא!)

bsp_init(smpd,argc,argv)	זאת חייבת להיות הפקודה הראשונה של התוכנית. מטרתה לאתחל את התוכנית במובן BSP. Smpd הוא שם הפונקציה אשר עתידה להיות הפעולה המקבילית של התוכנית. כלומר, קטע הקוד אותו כל אחד מהמעבדים יבצע
bsp_begin(n)	פקודה אשר מתחילה את הקטע המקבילי, בעוד ש-n מגדיר את מספר המעבדים שיריצו את הקטע המקבילי
bsp_end()	פקודה אשר מסיימת את הקטע המקבילי
bsp_nprocs()	מחזירה את מספר המעבדים הזמין המקסימלי. בתוך קטע מקבילי, מחזירה את מספר המעבדים אשר מבצעים את הקטע
bsp_pid()	מחזירה את המספר הסידורי של המעבד (מתחיל מ-0)
bsp_abort(msg)	אם מעבד כלשהו מזהה שגיאה, הוא יכול לגרום לכל המעבדים להפסיק את עבודתם
bsp_put(pid, source, dest, offset, nbytes)	הפקודה מעתיקה nbytes מהמעבד הנוכחי אל מעבד pid. כאשר source מצביע לתחילת קטע המידע שאנו רוצים להעתיק, ו-offset מגדיר את ההיסט מתחילת source. Dest מגדיר את היעד שאליו יועתק המידע. *(פעולת תקשורת חד-צדדית)*

bsp_get(pid, source, offset, dest, nbytes)	הפקודה מעתיקה nbytes ממעבד pid למעבד הנוכחי.
bsp_sync()	זהו למעשה מחסום אשר מפסיק את צעד החישוב הנוכחי. ההפעלה של הפקודה מתחילה את הביצוע של כל השליחות והקבלות שבוצעו במהלך צעד החישוב. לאחר סיום ביצוע הפקודה, ניתן להשתמש בכל המידע שנשלח/התקבל
bsp_push_reg(variable, nbytes)	פקודה אשר מאפשרת מיפוי זיכרון של המשתנה variable עבור כל המעבדים, כאשר nbytes הוא הגודל שלו.
bsp_pop(variable)	פקודה אשר מבטלת את הפקודה הקודמת ומוחקת את המיפוי

- כפי שנאמר לעיל, כל הפקודות אשר נקראות בצעד החישוב, מבוצעות רק בצעד התקשורת שאחריו. מנגנון BSP שומר חוצץ כפול, כך שבכל פעם שמופעלת פקודה שדורשת לכתוב מידע, המידע מועתק לחוצץ השני, ומאפשר שימוש מיידי בחוצץ של המשתמש (זאת, על מנת להבטיח בטיחות). עקב הדחייה של העברות המידע, הערך שמועתק הוא הערך שמופיע ממש לפני צעד התקשורת. בנוסף, BSP מבטיח לבצע את כל פעולות ה-get, ולאחריהן את כל פעולות ה-put, אך אינו מבטיח שהפעולות הנ"ל יתבצעו באותו הסדר בינן לבין עצמן.

Transactional Memory

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/chapter_18.ppt

- מטרתו של TM היא לחסוך מהמתכנת את הדאגה בנוגע לשימוש/אי-שימוש במנעולים. כל מקרי הקצה שיכולים לנבוע (מרוצי מידע, הרעבה, deadlock, livelock, וכו') ייפתרו מאחורי הקלעים. כל פעולה, למעשה, תוגדר להיות קטע קריטי. בצורה זו, TM מאפשר ביצוע עדכונים של מבני נתונים שונים בצורה אטומית.
- TM מגדיר שני מאפיינים בהקשר לטרנזקציות (Transaction)

1. *אטומיות (Atomic)*– ישנן שתי פעולות אפשריות, במסגרת בלוק אשר יוגדר להיות אטומי. הבלוק ייחשב אטומי מבחינת כלל התהליכים
   1.1. *Commit*– ביצוע כלל הפעולות במסגרת הבלוק
   2.1. *Abort*– כל הפעולות לא נחשבות והשפעותיהן מוחזרות (Rollback). בדר"כ, לאחר פעולה זו, מנסים מחדש את הקטע האטומי

   במסגרת קטע קוד אטומי, כלל הפעולות קרו, או אף אחת מהפעולות לא קרתה.
2. *סיריאליזציה (Serialization)*– ניתן למספר את הטרנזקציות, כאילו שהן התבצעו אחת אחרי השנייה בסדר מוגדר מבחינת כל החוטים / התהליכים. (אם לא ניתן לבצע את הסיריאליזציה הזאת, משמע שהקוד לא מומש כהלכה בשיטת TM)
3. פקודות נוספות
   1.3. *Retry*– ניתן לדרוש באמצע קטע קוד המוקף ב-*atomic* לבצע Rollback ולבצע את הקטע שוב, אם תנאי מסוים לא התקיים (לדוגמא)
   2.3. *orElse*– מופיע לאחר קטע קוד אשר מוגדר *atomic* ומגדיר בלוק אטומי בעצמו. אם הבלוק האטומי מעליו לא הצליח, מנסים לבצע את הבלוק הנ"ל. אם הבלוק הנ"ל לא הצליח, מבצעים את הכל מהתחלה.

- לצורך מימוש TM אין צורך בתמיכה של חומרה, אולם לצורכי ביצועים, מומלץ להשתמש ביכולות החומרה. מצד שני, TM דורש תמיכה של תוכנה, מכיוון שלדוגמא, מדיניות משתמש (Policy Issues) כדוגמת הטיפול ב-abort, לא צריכה להיות חלק מהחומרה.

עקביות (Transactional Consistency)

1) *External*– הרצה סדרתית של כלל הקוד צריכה להניב את אותה תוצאה שהתקבלה במציאות בהרצה מקבילית
2) *Internal*– צריכה להיות שמירת עקביות ברמת המשתנה. עצם זה שמשתנים A ו-B מתנגשים, לא צריך להשפיע על חישוב שאר המשתנים במערכת.

Simple Lock-Based STM (Software Transactional Memory)

- כל טרנזקציה שומרת סט של מיקומים וערכים שיש לקרוא (RS), וכן סט של מיקומים וערכים שיש לכתוב (WS). השינויים, וכן בדיקת ההתנגשויות, מתבצעות בשלב ה-commit.
- לכל משתנה, יהיה מנעול ש"שומר עליו" (ייתכן כי למספר משתנים יהיה אותו המנעול). נוסף לכך, תהיה לכל משתנה מס' גרסא, אשר תתעדכן (כלפי מעלה) כאשר נכתב משהו באובייקט הנעול.
- בעת קריאת אובייקט, לוקחים את מס' הגרסא ואת הערך של המשתנים (אלו שלא נעולים כבר) ומעתיקים לסט הקריאות (RS).
- בעת כתיבת אובייקט, לוקחים את מס' הגרסא ויחד עם הערך החדש שיש לכתוב, מעתיקים לסט הכתיבות (WS).
- בתהליך ה-Commit מתבצעות הפעולות הבאות:

1) רכישת המנעולים עבור האובייקטים שרוצים לכתוב אליהם
2) בדיקה שמספרי הגרסאות אשר רשומות ב-RS וב-WS נשארו זהות (אם ישנו שינוי, אז כנראה ישנה טרנזקציה אחרת אשר "מתעסקת" עם אותו המידע)
3) עדכון הערכים החדשים (של הכתיבות)
4) עדכון (העלאה ב-1) של מס' הגרסא (דבר זה מתבצע רק לאחר סיום ה-Commit)
5) שחרור כלל המנעולים

- במהלך בדיקות ה-Commit יכול להיות מצב שבו טרנזקציה מסוימת נכנסת למצב זומבי (Zombie), זאת מכיוון שבמהלך הבדיקות, התברר כי המצב (של אחד המשתנים לפחות) הוא לא-קונסיסטנטי (כלומר, בשלב 2 של ה-Commit, התהליך המסוים רוצה לקרוא ערך, אבל נכנסה לפניו טרנזקציה אחרת, אשר הספיקה לכתוב למשתנה ערך חדש ולעדכן את מס' הגרסא). כאן, יכול לקרות מצב שביצוע ה-abort לא יעזור ונגיע לשגיאות (לדוגמא, חלוקה ב-0). על מנת לפתור את הבעיה הנ"ל, נשתמש בשעון גלובלי משותף בין כלל הטרנזקציות במערכת. נתבונן בשני סוגים של טרנזקציות, ונראה את האלגורתם משתנה בשלב ה-Commit:

- טרנזקציות קריאה-בלבד (Read Only)
  1) העתקת השעון הגלובלי (V) אל שעון פנימי (RV)
  2) קריאת מס' הגרסא של המנעול
  3) קריאת הערך מהזיכרון
  4) בדיקה שהמנעול לא-נעול
  5) בדיקה שמס' הגרסא לא השתנה
  6) בדיקה האם מתקיים מס' גרסא <RV
- טרנזקציות רגילות
  1) העתקת השעון הגלובלי (V) אל השעון הפנימי (RV)
  2) עבור כל קריאה וכתיבה מבצעת בדיקה שהמנעול לא נעול, וכן שמתקיים מס' גרסא <RV, ולבסוף הוספה לסט קריאה/כתיבה (R/W)
  3) רכישת כלל המנעולים
  4) *Fetch&Increment*– פעולה אטומית, אשר מטרתה לקחת את השעון הנוכחי, ולקדם אותו ב-1. זאת, על מנת לשמור על קונסיסטנטיות של המערכת
  5) בדיקה שמתקיים עבור כל מס' גרסא <RV
  6) עדכון מס' גרסא של כל פעולות הכתיבה לערך של RV

בעבודה עם זיכרון מטמון (Cache), TM מבצע ביטול (Invalidation) ובדיקת עקביות (Consistency), ב-TM יש מנגנון של עקביות של זיכרון המטמון (Cache Coherence) אשר דואג לבטל טרנזקציות שמתנגשות מבחינת קריאה-כתיבה, ולסמן את התאים המתאימים ב-D (Dirty) וב-M (Modified).

ברמת הגרעיניות, TM מאפשר גרעיניות ברמת האובייקט בשפות כגון Java ו-C#. דבר זה מקל על האינטרקציות ובדיקות של טרנזקציות. בעוד שבשפות אחרות, כדוגמת C ו-C++, הגרעיניות היא ברמת המילה (Word), ולכן קשה יותר לנהל את הטרנזקציות.

שיטות עדכון אפשריות

1. *דחוי (Deferred)*– העדכונים מתבצעים בעותקים צדדיים, והעדכון האמיתי הוא בזמן ה-Commit. עקב זאת, נדרשות פעולות מיוחדות בזמן ה-Commit. אולם, צורת העדכון הנ"ל מאפשרת שמירה קלה על עקביות.
2. *ישיר/מיידי (Direct)*– העדכונים מתבצעים "על המקום", במקרה של abort מבוצע rollback. עקב זאת ה-Commit קצר יותר, ולכן יעיל יותר. אולם, ישנה פגיעה בשמירת קונסיסטנטיות.

שיטות גילוי קונפליקטים אפשריות

1. *חמדן (Eager/Greedy)*– לגלות את הקונפליקטים לפני שהם מתרחשים. [יכול להתקיים מצב שבו התבטלו טרנזקציות שיכלו להתבצע בפועל]
2. *עצל (Lazy)*– לגלות את הקונפליקטים בזמן Commit/Abort. [לא מתייחס לחישוב]
3. *מעורב (Mixed)*– קונפליקט W/W באופן חמדן, קונפליקט R/W באופן עצל.

טרנזקציות מקוננות (Nested Transactions) ונראות (Visibility)

1. *רמה אחת – שיטוח (Flat nesting)*– אם הבן מבצע Abort, כך גם ההורה שלו
2. *קינון באותה רמה (First-class nesting)*– אם הבן מבצע Abort, מתבצע rollback חלקי של הבן בלבד.

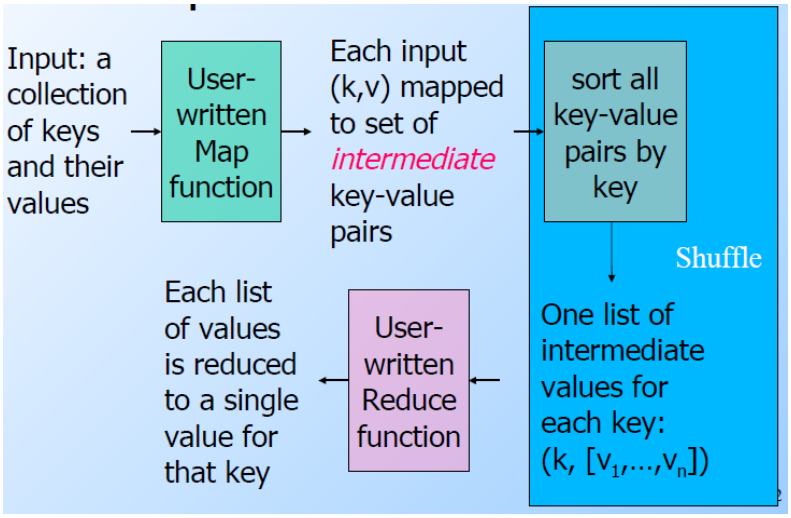
בדר"כ, כאשר הבן מבצע Commit, רק ההורה רואה זאת. אולם, ב-Open Nested, ה-Commit נראה ע"י כולם.

Map-Reduce

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/map-reduce-lecture.pdf

http://labs.google.com/papers/mapreduce-osdi04.pdf

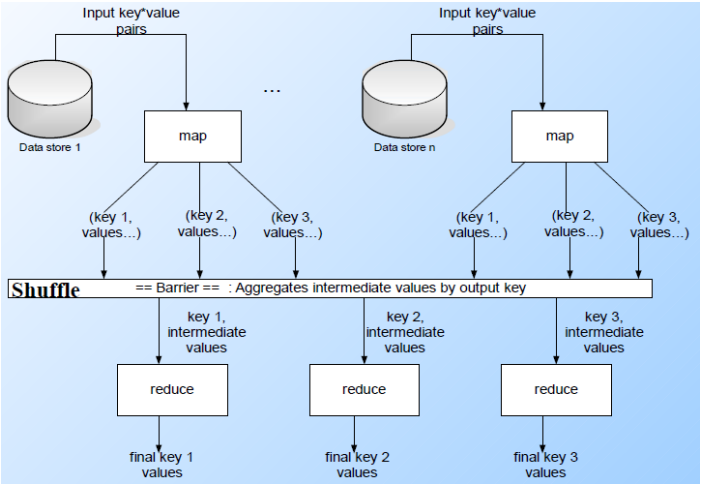
- MR היא מסגרת תוכנה אשר הוצגה ע"י Google במטרה לתמוך בחישוב מבוזר של מידע בסדרי גדול באשכולי מחשבים. המסגרת מספקת את הדברים הבאים:
  - מקבול וחלוקת מידע אוטומטית
  - עמידות בנפילות (היכולת לייצר פלט עקבי, כלומר, אותו הפלט שיתקבל במקרה שלא היו נפילות)– יש שימוש בפעולות אטומיות על מנת להבטיח זאת
  - מדיניות תזמון קלט/פלט
  - ניטור (Monitoring) ועדכונים תו"כ
- חישוב בעזרת MRמתחלק לשלושה שלבים (מתקיימת הנחה כי הפעולות של MR הינן אסוציאטיביות וקומוטטיביות):



1) *Map(key,val)*–פונקציה אשר ממומשת ע"י המשתמש, וצריכה לדעת לקבל אוסף של זוג מפתח וערך, ולייצר סט של מפתחות זמניים וערכים
2) שלב הביניים אוסף את כל הערכים עבור מפתח זמני K כלשהו ושולח את המפתח יחד עם הקבוצה אשר מתאימה לו ל-Reduce. מתבצע כאן שלב איסוף ומיון הערכים, על מנת להקל על פעולת ה-Reduce.
3) *Reduce(key,vals)*–פונקציה אשר ממומשת ע"י המשתמש, וצריכה לדעת לקבל מפתח וסט של ערכים, ולייצר את הפלט המתאים ביחס אליהם.

חשוב לציין כי MR עובד עם מחרוזות בלבד, ואחריותו של המשתמש לקבל מחרוזות כקלט, ולפרש אותם בצורה הנכונה בהקשרים שלו. וכמובן, הפלט צריך להיות גם בצורה של מחרוזות.

מקבול

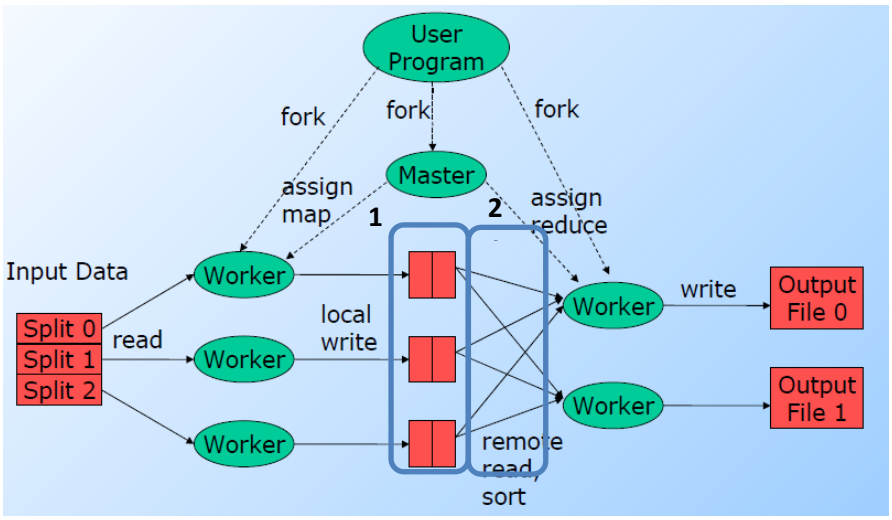


החלוקה מתבצעת בשלבים, ולאחר החלוקה, מתחיל שלב של מקבול (חלוקת משימות בין העובדים). קיימים שלושה שלבים:

1) חלוקת הקלט של זוגות key/value לחתיכות (בדר"כ בגודל 16-64Mb), והרצת מטלות map במקביל
2) ערבוב (Shuffle) של כל הפלטים של מטלות ה-map (כשכולם הסתיימו), מתבצע איחוד של כל הפלטים מה-map עבור כל מפתח ייחודי.
3) חלוקת המפתחות שנפלטו ע"י map, וביצוע מטלות reduce במקביל.

במקרה ו-()map או ()reduce נכשלים, מתבצעת הפעלה מחדש.
החלוקה בשלב הראשון מתייחסת גם ללוקליות, על מנת להאיץ את הביצועים.

מימוש



התוכנית מייצרת תהליך **מנהל** ומספר מרובה של תהליכים **עובדים**. הקלט מחולק למספר **מקטעים (Splits)**, ובהתאם לשלב העיבוד, עובד מקבל משימה של ביצוע Map על מקטע כלשהו, או לבצע Reduce לסט כלשהו של מפתחות.

(1) שלב של יצירת קבצי ביניים
(2) שלב זה נקרא שלב ה-shuffle. והוא צוואר-בקבוק של כל המערכת. הוא צורך תקשורת רבה על מנת לחלק את המפתחות בין העובדים השונים.

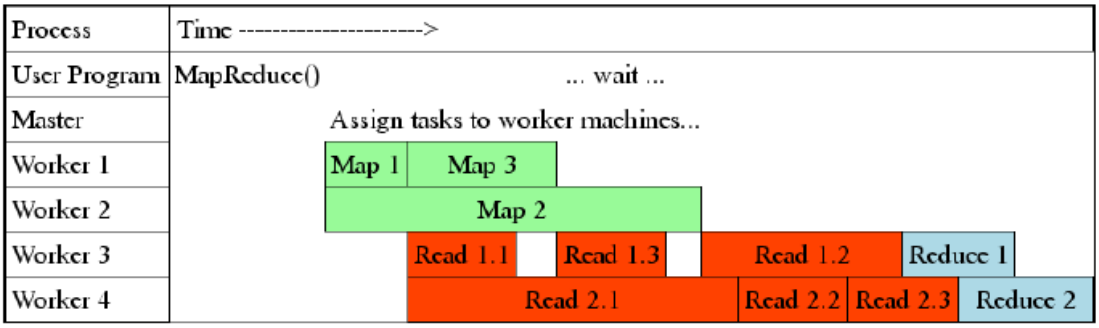
- לכל מטלה יישמר המצב שלה (idle, in progress, completed), ובנוסף זהות העובד אשר אחראי עליה.

תחום האחריות של התהליך המנהל

- הקצאת משימות map/reduce לעובדים
- עמידות לנפילות (התמודדות)
  - התהליך המנהל שולח מדי פעם הודעות (ping) לכל עובד. אם לאחר זמן כלשהו (מוגדר מראש) לא התקבלה תגובה מעובד כלשהו, הוא מסומן כ"נכשל", וכל מטלות ה-map שהעובד הנ"ל סיים, מאותחלות ומסומנות כ"פנויות לעבודה" עבור עובדים אחרים. כמו כן, כל מטלת map או reduce בביצוע ע"י העובד הנ"ל תסבול מגורל זהה. (מתבצעת הפעלה מחדש של כל פעולות ה-map מכיוון שהן נשמרות על המכונה של העובד באופן לוקאלי, ואם הוא נפל, אזי הן לא נגישות. אין צורך לבצע מחדש פעולות reduce מכיוון שהן נשמרות במערכת קבצים גלובלית).
  - התהליך המנהל מזהה מפתחות-ערכים יחודיים אשר גורמים לקריסה של פעולות map, ופוסח על ערכים אלו בזמן ביצוע מחדש.
  - אין גיבוי לנפילה של התהליך המנהל, מכיוון שיש מנהל יחיד, וקיימת הנחה כי כישלון / נפילה הינם לא סבירים (למרות שניתן מדי פעם לבדוק את מצבו של התהליך המנהל, ולדרוש שיבצע גיבוי של כל מה שהוא מכיל, על מנת להתמודד עם נפילה כגון זו)
- העברת התוצאות של map לשלב ה-reduce:
  - בחירת R מטלות reduce
  - חלוקת מפתחות הביניים (משלב ה-map) ל-R קבוצות (לדוגמא, ע"י ערבול)
  - כל מטלת map, מייצרת במעבד שלה R קבצים של זוגות של מפתחות ביניים-ערכים, אחד לכל מטלת reduce
- ביצוע אופטימיזציות – אף reduce לא יכול להתחיל לפני שכל ה-map הסתיימו, במקרה זה, דיסק מרוחק אשר פועל לאט יכול להשפיע עלכל המערכת. כתוצאה מכך, התהליך המנהל מפעיל מספר פעמים "מיותרות" (redundant) את מטלות ה-map שמזוהות כאיטיות, ומשתמש בתוצאה הראשונה שתתקבל.

גרעיניות המטלות ועבודה בקו-ייצור (pipelining)

בדר"כ, מספר מטלות ה-map גדול בכמה סדרי גודל ממספר המכונות. הדבר מאפשר עמידות טובה יותר לנפילות, בנוסף עבודה בצורה של pipeline בין מטלות map לבין shuffle. וכן, ניתן לבצע חלוקת עומס דינאמית בצורה יעילה.
מבחינה מספרית, בדר"כ יש כ-200,000 פעולות map אל מול 5000 פעולות reduce, והריצה היא על 2000 מכונות.



Condor

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/Condor.2006.pdf

אשכול (בריכה) [Cluster]–קבוצה של מחשבים המחוברים ביניהם

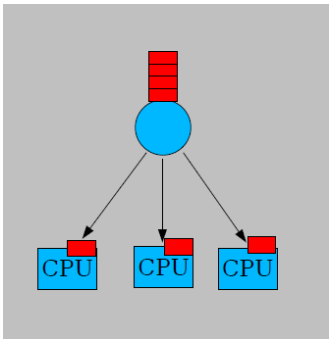
אוסף עבודות ([Job] Batch) –קטע תוכנה סיריאלי אשר יכול להתבצע בעצמו

מערכת לניהול אוסף/ניהול תור (Batch/queuing system) –מערכת לניהול אוטומטי של תזמון והפעלת עבודות, אשר מתחרות על משאבים

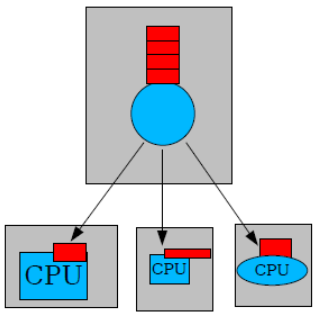
מערכת בעלת ביצועים גבוהים (High Performance System) –מערכת אופטימלית בעלת זמן השהייה נמוך לעבודות (המטרה היא להעביר כמה שיותר מהר כל עבודה)

מערכת בעלת הספק גבוה (High Throughput System)– מערכת אופטימלית להעלאת ניצול המשאבים (המטרה היא שכמה שיותר עבודות יתבצעו, ללא חשיבות של זמן הביצוע של עבודה מסוימת)*[Condor הינה מערכת מבוזרת מהסוג הנ"ל]*

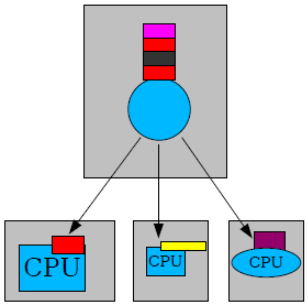
הסוגים השונים של המערכות



1. *משאבים מרובים זהים (Multiple identical resources)*
   - ישנו תור של עבודות
   - המערכת אחראית להפעיל את העבודות ולאסוף את התשובות
   - כל עבודה מופעלת פעם אחת, וקיימים מצבים של נפילות עבודה (וגם יכול לקרות מצב שאם יתקיים ניתוק זמני, עבודה כלשהי תישלח פעמיים)

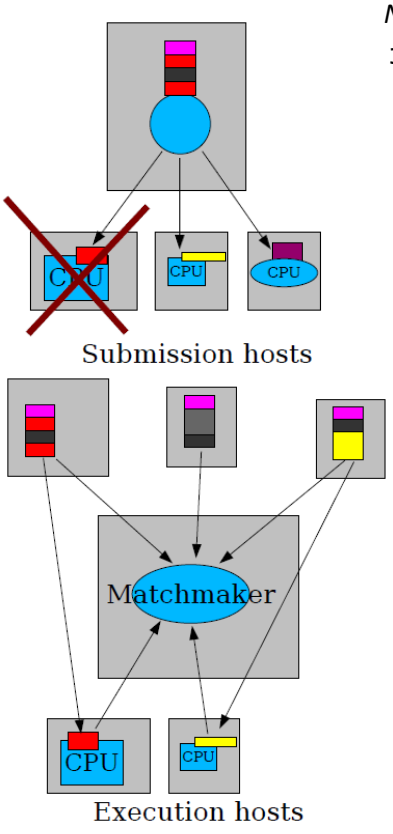


2. *משאבים מרובים שונים (Distributed heterogeneous resources)* [נוסף למאפיינים של המערכת הקודמת]
   - שליטה מרחוק
   - שמירת מידע על המשאבים השונים (על מנת להתאים עבודה למשאב באופן אופטימלי)
   - התייחסות לדרישות המשאבים של העבודה



3. *משתנים שונים (Multiple users)* [נוסף למאפיינים של המערכות הקודמות)
   - בטיחות (משתמשים שונים ניגשים לתור של העבודות, יש צורך למנוע מעקפים וכולי)
   - מדיניות חלוקת/שיתוף משאבים (אותה כמות משאבים לכל תהליך בהתאם לכמות העבודה שלו, זאת על מנת למנוע תפיסת כל המשאבים ע"י עבודות מסוימות)
   - שליטה וניהול הגישה לתור העבודות

4. *משאבים אשר לא מוקדשים לחלוטין לעבודות (גניבת זמן משאב) (Non-dedicated resources – cycle stealing)* [נוסף למאפיינים של המערכות הקודמות]
   - עדכון עיתי של המידע הנשמר על כל משאב
   - פינוי עבודות לפי דרישה
   - עמידות לנפילות (יכולת להתמודד עם מצב של הוספת/הסרת משאבים)
   - התמודדות עם מדיניות הפעולה של המשאבים



ארכיטקטורת Condor

משאבים "מפרסמים" את המפרט (יכולות) שלהם. עבודות "מפרסמות" את הדרישות שלהם, ותפקידו של המשדך (MatchMaker) הוא למצוא את השידוך הטוב ביותר.

הצד המייצר (Submission host)

מכיל שני סוגים של רכיבים:

1. *Schedd* – תור של עבודות. הוא מכיל מבנה נתונים המחזיק את כל העבודות האפשרויות (מתבצעת אליו פנייה לקבלת עבודות והוספת עבודות). הוא דורש משאבים מהמשדך על מנת לשלוח את העבודות המתאימות. עובד על פי לוגיקה של דרישה של ביצוע העבודות בתור, ומוודא שישנה הפעלה של כל עבודה פעם אחת בלבד.
2. *Shadow* – אחד עבור כל עבודה שמורצת. מופעל מרחוק, ו"יושב על העבודה" על מנת לנתר את הביצועים שלה על לצורכי זיהוי הצלחה/נפילות (בכך, מוריד את האחריות מ-Schedd). מדמה פעולות קלט/פלט, ולפעמים גם מתפקד כ-proxy של קלט/פלט.

הצד המבצע (Execution host)

מכיל שני סוגים של רכיבים:

1. *Startd* – מנהל המשאבים. אחראי על הניטור של המשאב (מעקב אחרי השימוש במשאב [לדוגמא, אם עבודה מסוימת מנצלת זיכרון מעבר לרף שהוגדר, יש צורך לבטל אותה], שליחה עיתית של מאפייני המשאב למשדך ואכיפת המדיניות של המשאב במובנים של זמן שימוש וכולי), ומתפקד כשער התקשורת אל מול הרכיבים האחרים (דורש שמירה על בטיחות, ייצור *Starters* ותקשורת עם *Schedd*)
2. *Starter* – מתקשר עם ה-*Shadow* (קלט/פלט), מייצר את סביבת ההפעלה, ומנקה אותה בסיום, ומנהל את הפעלת העבודה.

משדך (Matchmaker)

מכיל שני סוגים של רכיבים:

1. *Collector*– מכיל את כל המידע במבנה הנתונים. *Startd*-ו*Schedd* שולחים לו את האינפורמציה הרלוונטית (מאפייני משאבים / דרישות עבודה). המידע שנשמר ב-*Collector* לא ניתן למחיקה ע"י רכיבים אחרים של המערכת, אלא רק על ידו. דבר זה מאפשר מעקב אחרי עבודות ומשאבים מבלי לגשת אליהם פיזית. (אחריות העדכון היא על *Schedd*-ו*Startd*)
2. *Negotiator*– ה"מוח" שמנהל את Condor. אחראי להוציא מידע מה-*Collector* (למעשה, המידע לא מכיל את העבודות עצמן, אלא את ה-schedd המייצגים ואת מספר העבודות שיש לכ"א), לייצר שידוך בין דרישות משאבים לבין המשאבים המוצעים, ליידע את הזוגות שנמצא ביניהם שידוך, ולשמור על חלוקת משאבים הוגנת בין המשתמשים. מדי פעם (Periodic Cycle) ה-Negotiator מוציא את כל המידע שנמצא ב-Collector על מנת לבצע התאמות, בעזרת אלגוריתם השידוך.

אלגוריתם השידוך

1. הוצא את כל המידע מה-Collector (ClassAds)
2. צור קשר עם כל schedd לפי **העדיפות** ודרוש את ה-ClassAd של העבודה
3. עבור כל עבודה, עבור על כל ה-ClassAds של המשאבים, ונסה להתאים משאב לכל עבודה
4. אם נמצא שידוך
   1.4. בחר את השידוך הטוב ביותר **בהתאם למדיניות גלובלית ולוקלית**
   2.4. יידע את שני הצדדים (schedd ו-startd)
   3.4. הוצא את שניהם מהרשימה
5. אם לא נמצא שידוך, בקש מה-schedd את העבודה הבאה, או גש ל-schedd הבא לפי העדיפות.

המידע ב-Condor מועבר ב"שפה משותפת" הנקראת **ClassAd**. כל עבודה/משאב מוגדרים בעזרת השפה הנ"ל, ומתקיימת הנחה כי ההגדרות ושמות המשתנים הן זהות לחלוטין, אחרת לא ניתן יהיה לבצע שידוך.

מדיניות

- ברמת *Startd*, האחראי על המשאב יכול להגדיר מתי המשאב מוכרז כזמין, מה לעשות כאשר הוא חוזר להשתמש במשאב, או מהי הדרך לפינוי עבודה מהמשאב.
- ברמה הגלובלית, יש צורך שתהיה חלוקת משאבים הוגנת, כך שלא ייקרה מצב שמשתמש כלשהו ינצל את כל המשאבים, ומשתמשים אחרים ייצטרו להמתין לסיומו. הפתרון הוא שמשתמש עם עדיפות גבוהה יותר יכול להפקיע עבודה עם עדיפות נמוכה יותר. במצב זה, העדיפויות משתנות בצורה דינאמית לפי כמות השימוש במשאבים. כלומר, ככל שתהליך ישתמש ביותר משאבים, העדיפות שלו תהיה נמוכה יותר.
  כאשר אצלנו , $prio_{user}(t) = k * prio_{user}(t - dt) + (1-k) * (number\_of\_used\_resources)$
  $k = 0.5^{dt/(priority\_half\_life)}$.
- במצב הרגיל, בתהליך מציאת השידוכים, כאשר מתבצע שידוך בין *Schedd* לבין *Startd*, הקשר ביניהם ממשיך ויוצא מתחום אחריותו של המשדך. מצב זה מאפשר לתהליך מסוים להפעיל את המשאב לזמן ממושך. על מנת למנוע זאת, ה-*Negotiator* שומר את המידע לגבי אלו שיש ביניהם חיבור, ואם הוא מזהה שימוש יתר בהם, הוא מפסיק את החיבור.

אלגוריתם שידוך משופר

1. הוצא את כל ה-ClassAds מה-Collector (סדר בצורה אופטימלית לטובת השידוך)
2. סדר את כל בקשות ה-schedd לפי עדיפויות משתמש (הגבוהה ביותר ראשונה)
3. עבור כל משתמש בצע:
   1.3. כל עוד מכסת המשתמש לא מולאה וגם יש לו עוד בקשות עבודה בצע:
      1.1.3. צור קשר עם schedd לקבלת ה-ClassAd של העבודה הבאה
      2.1.3. עבור על כל ה-ClassAds של המשאבים ונסה למצוא שידוך מתאים
         1.2.1.3. אם לא נמצא שידוך, הודע ל-schedd, וחזור לשורה 3.1
         2.2.1.3. אם נמצא שידוך, בצע AssignWeigths() והוסף את השידוך לרשימת השידוכים הפונטציאליים
      3.1.3. הפעל את ChooseBestMatch() ו-Notify()

AssignWeiths() מגדירה עבור זוג בשידוך הפוטנציאלי את הדברים הבאים:

- עבור המשאב, מגדירה את מצבו מבחינת הפקעה (preemption weight), כאשר:
  - 2 – המשאב זמין (idle)
  - 1 – המשאב עסוק (busy) אך מעדיף לקבל עבודה חדשה על פני העבודה הנוכחית (Resource rank evaluation)
  - 0 – המשאב עסוק, ולמשתמש הנוכחי יש עדיפות גבוהה יותר, והמדיניות הגלובלית מאפשרת הפקעה
- עבור העבודה, מעריכה את מאפייני העבודה (Job rank evalution)

ChooseBestMatch() ממיינת את העבודות (לפי סדר לקסיקוגרפי) ובוחרת את הטובות ביותר

- מבצעת מיון לפי דרוג העבודה (job rank) ובוחרת את הטובה ביותר
- מבין אלה בעלי דירוג טוב ביותר, אולם זהה, מיין ובחר לפי אפשרות משקל ההפקעה (preemption weight) אשר ניתנה ב-AssignWeiths().

CUDA

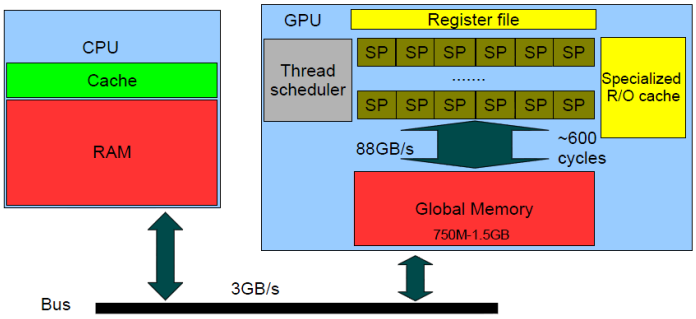
http://sites.google.com/site/silbersteinmark/Home/talks/GPGPU-overview.pdf?attredirects=0

בניגוד ל-CPU, ה-GPU לא מבצע שימוש בזיכרון מטמון מכיוון שבמודל, חישוב של כל נקודה הוא ב"ת, וניתן להשתמש ב-Graphical pipeline ברמת החומרה, ולתמוך בכל זה על הרכיב עצמו. ברמת ה-GPU מבוצע hyper-threading, כלומר, ה-GPU יודע להריץ מספר רב של חוטים, כאשר חוטים אשר מחכים לזיכרון, ייכנסו להמתנה, וחוט אשר דורש חישובים, ייכנס ל-pipeline לחישוב.

אינטנסיביות חישובית (Arithmetic Intensity) היא מספר ה-Flops עבור כל גישה לזיכרון. לדוגמא, אם עבור α פעולות יש β גישות לזיכרון, אזי $AI = \frac{\alpha}{\beta}$).

חישוב הביצועים הוא המינימום בין רוחב פס הזיכרון (כמות המידע שניתנת להעברה) * כמות האינטנסיביות החישובית לבין יכולת החישוב של ה-GPU.

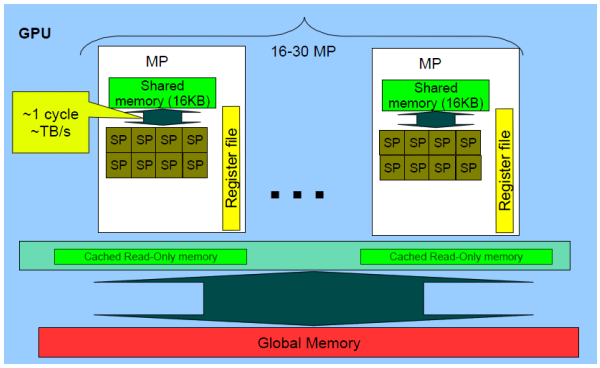
מודל GPU מוכוון הספק (Throughput oriented architecture)



המודל מתבסס על מספר רב של מעבדים אשר עובדים במקביל, כלומר, מספר רב של חוטים אשר פועלים בו-זמנית (ברמת האלפים). החוטים ניגשים לזיכרון הגלובלי של ה-GPU, ואין שימוש בזיכרון מטמון, ולכל חוט יש מספר מוגבל של רגיסטרים לרשותו. בנוסף, המודל הוא SPMD, כלומר, ישנה תוכנית אחת בשם **kernel** אשר מורצת על מידע שונה (כל חוט מריץ את ה-kernel).

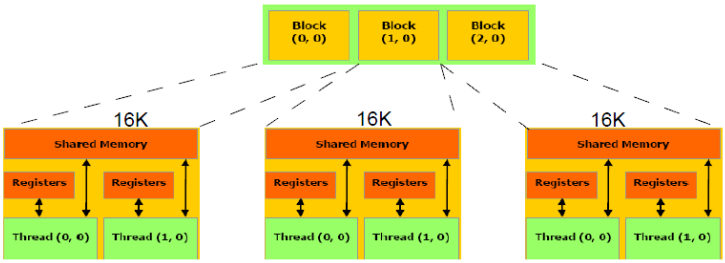
המודל הנ"ל מתאים לחישובים מקביליים בצורה כמעט מוחלטת (Embrassignly parallel), כאשר יש הרצת אלגוריתם כלשהו על מידע רב, ויש שימוש חוזר נמוך יחסית במידע. ובנוסף, כאשר יש מספר רב של חישובים ביחס לכל גישת זיכרון (Arithmetic intensity), על מנת לכפר על ההשהייה של הגישה לזיכרון. במקרים אחרים, ההאצה תהיה נמוכה (ככל שהתלות בזיכרון גבוהה יותר).

מודל NVIDIA (Compute Unified Device Architecture – CUDA)



בניגוד למודל הקודם, ישנו כאן שימוש בזכרון מטמון, שהוא הזיכרון המשותף ברמת ה-MP (Multi processor), כאשר כל ה-SP's (Single processor) אשרבמסגרתו, משתפים את הזיכרון הנ"ל, ולכן יש צורך במנגנון מובנה תוכנה ולא מובנה חומרה על מנת להתמודד עם גישות (מכיוון שכל SP ב"ת באחרים) על מנת למנוע מרוצי מידע. בנוסף, במודל הנ"ל, כל MP יודעלבצע תזמון (Scheduling), להקצות רגיסטרים ולנהל את הזיכרון ברמתו.

חוטים מאוגדים בבלוקים (*Thread block*), כאשר בכל בלוק כזה יכולים להיות עד 512 חוטים. הזיכרון משותף בין החוטים אשר נמצאים באותו הבלוק. הרעיון הוא להתבונן על הבלוקים כעל רצף (*Stream*), והמטרה היא לייצר מבנה חישוב אשר מנצל בצורה אופטימלית את הזיכרון המהיר (זיכרון בעל השהייה נמוכה ורוחב פס גדול). הבלוקים הנ"ל לא מתקשרים אחד עם השני (מבחינת סנכרון), וכל פעולות הסנכרון מתבצעות רק ברמת הבלוק עצמו (תזמון בין החוטים, תקשורת דרך זיכרון משותף וסנכרון).



כפי שנאמר, הבלוקים הנ"ל ב"ת, ויש בעיה בתקשורת ביניהם. שימוש בפעולות אטומיות וסנכרון יכול לגרום ל-Deadlock. (לדוגמא, אם נתבונן במודל מנהל/עובד, העובדים מחכים למנהל, אולם המנהל לא יכול לרוץ מכיוון שכל החומרה תפוסה).

ישנה אפשרות להריץ בלוקים שונים על אותו ה-MP, הדבר יוצר בעיה, מכיוון שהזיכרון המשותף ומספר הרגיסטרים צריכים להיות גדולים מספיק על מנת לאפשר הפקעה. ישנן שתי אפשרויות:

- פחות חוטים בכל בלוק – כך פחות בלוקים יתוזמנו (Scheduled), ונוכל לקבל מצב שבו ההספק יורד.
- יותר חוטים בכל בלוק – יותר בלוקים יתוזמנו, אולם יהיו פחות רגיסטרים וזיכרון משותף עבור כל בלוק.

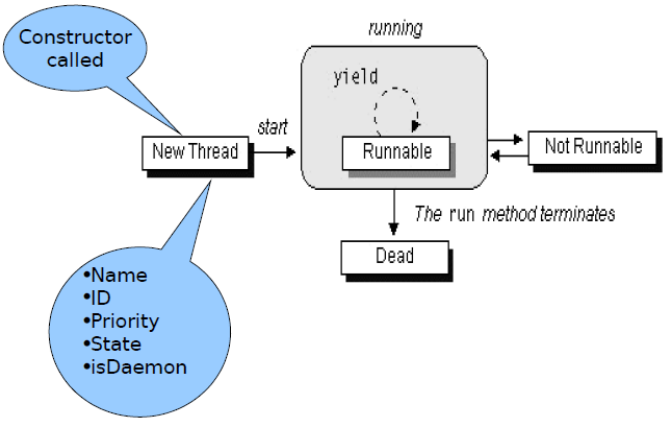
תכנות מקבילי ב-Java

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/threads-in-java.pdf

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/ParallelAlgDesign.pdf

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/AsyncInvocations.pdf

חוטים ב-Java



- ב-Java, חוט הוא אובייקט שניתן להרצה (Runnable object). ה-JVM אחראית על המיפוי בין החוטים הנ"ל לבין החוטים האמיתיים של מערכת ההפעלה (לא ניתן להניח דבר על מדיניות הזימון והחלפת ההקשר), מכאן ניתן להסיק כי כל יצירת חוט הינה פנייה ל-JVM ליצירת חוט אשר יפעל במקביל לחוט הנוכחי. כתיבה/מימוש של חוט מתבצעת ע"י הרחבת המחלקה Thread, או מימוש הממשק Runnable. (במסגרת המימוש הנ"ל, יש לממש את שתי המתודות ()run ו-()halt. הראשונה היא המתודה שתופעל על מנת שהחוט יתחיל לרוץ, והשנייה היא המתודה אשר קובעת את תנאי העצירה. חשוב לציין, כי אין בדר"כ הפעלה ישירה של ()run, אלא הוא מופעל לאחר ()start. אולם, אם מפעילים בצורה ישירה את ()run דרך האובייקט, ה-JVM למעשה לא מייצר חוט נפרד שירוץ במקביל - כמו במקרה של הפעלת ()start, אלא יריץ את הגוף של המתודה ()run במסגרת החוט הנוכחי).
- המנגנון הבסיסי שקיים ב-Java הוא ה-**monitor** (מנעול + משתנה תנאי). בנוסף, לכל אובייקט ב-Java יש מנעול משלו (מוגדר כחלק מהמאפיינים שלו). על המנעול הנ"ל לא מתקיימת הוגנות, ולא בהכרח סדר התפיסה שלו הוא סדר הכניסה והביצוע. המנעולים הם Reentrant, ומונעים מצב של deadlock, בנוסף, ב-Java מתאפשרת תפיסה של אותו המנעול מספר פעמים (ע"י אותו החוט).

גורמים המשפיעים על יעילות המקבול

- חוק אמדאל מציב חסם עליון על ההאצה, לא ניתן לייעל את החלק הסיריאלי ע"י הוספת מעבדים, מכיוון שאלו פעולות שכל המעבדים צריכים "לשלם" (החסם העליון הנ"ל הינו בהנחה כי חלוקת העומס הינה אופטימלית)
- תקורת הסנכרון והאינטרקציה בין המעבדים השונים
- השפעות המעבד, כגון עיכובים כתוצאה מתלויות מידע או חלוקת עומס לא-אופטימלית.

הערכת אלגוריתם מקבילי

- האצה - $Speedup = T_{serial} / T_{parallel}$ (Super linear speedup מתקיים כאשר ההאצה היא ביותר ממספר המעבדים. ניתן להגיע לשיפור כזה בעזרת, לדוגמא, שימוש חכם ב-cache).
- יעילות - $E = S / \#\,processors$
- עלות - $C = \#\,processors * T_{parallel}$
- יכולת התאמה - ההאצה כפונקציה של מספר המעבדים עבור מטלה בגודל קבוע

גרף תלויות (Task Dependency Graph)– בעת חלוקה של אלגוריתם למקבול, בעזרת חלוקה למטלות, ניתן לבנות גרף אשר מייצג את התלויות בין המשימות השונות. בעזרת הגרף הנ"ל ניתן לראות את**רמת המקביליות (Degree of Concurrency)** אשר מייצגת את מספר המטלות שניתן להריץ במקביל. בנוסף, ניתן לראות מהו **המסלול הקריטי (Critical Path)** אשר מייצג את המסלול הארוך ביותר בגרף התלויות, ואורכו נותן חסם תחתון לזמן ריצת התוכנית המקבילית.

הפעלה א-סינכרונית ושימוש במאגר חוטים (Asynchronous invocation & Thread Pool) ב-Java

- הפעלה א-סינכרונית מתבססת על הרעיון שבו הקורא לפעולה לא צריך להמתין עד לסיומה, ויכול לקבל את תשובותיה בזמן כלשהו בעתיד (אם בכלל). השיטה הזו יעילה לעבודה של שרתים (ראה ערך, מערכות הפעלה), ובכללי, להפעלה של מספר כלשהו של פעולות עצמאיות בו-זמנית. ברוב המקרים, אכן נרצה לדעת מהן התוצאות, ונרצה (למשל) "להתעדכן" בנוגע לסיום פעולה/מידע חדש. על מנת לממש פונקציונליות כזאת, ניתן להשתמש ב-*Observer Pattern*. בשיטה הנ"ל, צופה אשר רוצה לקבל אינפורמציה לגבי אירועים שונים, "נרשם" *(Subscribes)* אל מי שהוא רוצה לקבל ממנו אינפורמציה, ובזמן המתאים, הוא מופעל / מוער כאשר המידע מוכן.
- שיטה נפוצה הינה שימוש ב*מאגר חוטים (Thread pool)*. בדר"כ, החוט הראשי אחראי לנהל תור של מטלות, ומייצר מספר חוטי עובדים, אשר מקבלים את המטלות ומריצים אותן בצורה א-סינכרונית. אחריותו של החוט הראשי היא לנהל את המטלות, לחלק אותן לעובדים (למרות שניתן לייצר מבנה נתונים גלובלי של מטלות, וכל חוט ייגש לקחת את המטלה הבאה, או ימתין עד שיהיו מטלות), ולטפל בפלטים הא-סינכרוניים (שוב, גם כאן, ניתן להגדיר כי הפלט, אם יש בו שימוש, יעובד וייפלט ע"י חוטי העובדים עצמם). מספר חוטי העובדים הוא דינאמי, וניתן לייצר/למחוק חוטים לפי דרישה, על מנת לנהל בצורה יעילה יותר את המערכת, הן מבחינת זמן, והן מבחינת משאבים.
- ישנן שלוש שיטות להגדיר מאגר חוטים:

1) תור מטלות לא-מוגבל ומספר מוגבל של חוטים

המודל מתאים אם מספר זמן ההמתנה של כל מטלה הוא סופי ומוגבל, אך קיימת בעיה עם המודל באופן כללי, זאת מכיוון שלא ידוע מתי מטלה כלשהי תתבצע אחרי שהיא נכנסת לתור.

2) תור מטלות מוגבל ומספר לא-מוגבל של חוטים

המודל מתאים אם אין קפיצה (פרץ) של מטלות בבת-אחת, מכיוון שכנראה הדבר ידרוש יצירה של מספר רב מאוד של חוטים, אשר יאיט את המערכת.

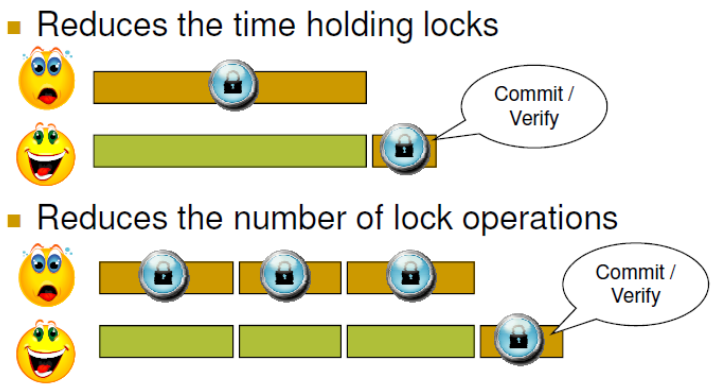
3) תור מטלות מוגבל ומספר מוגבל של חוטים

המודל יכול לגרום לדחיית מטלות, ובנוסף, ייתכן כי המטלה אשר מייצרת חוטים תיחסם ויווצר deadlock. בנוסף, אם חוט מכניס מטלה שתלויה במטלה שלא הוכנסה עדיין לביצוע. או לדוגמא, אין חוטים פנויים ונוצרות תלויות בין החוטים ובין המטלות השונות.

- גודל מאגר חוטים אופייני(/מומלץ) לפי סוג המטלות:
  - מטלה חישובית (תלויית מעבד) – מספר החוטים כמספר המעבדים (אם יהיו יותר חוטים, נשלם על התקורה של החלפות ההקשר)
  - מטלת קלט/פלט (תלויית ק/פ) – $\#CPUs*(1+WaitTime/ServiceTime)$. כאשר, WaitTime הוא זמן ההמתנה הממוצע ללא שימוש במעבד, ו-ServiceTime הוא זמן הממוצע לשימוש במעבד(כולל זמן עבודה + זמן המתנה).
- ב-Java (החל מגרסא 5.0) ישנה תמיכה במאגרי חוטים. ניתן להשתמש ב-*Executors* על מנת לייצר מאגרי חוטים, וכן להשתמש ב-*Future/Callable* על מנת לבדוק ערכי החזרה / שגיאות.

תכנון אופטימיסטי (Optimistic Design)

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/Optimistic%20Design%20new.ppt



המטרה של תכנון אופטימיסטי היא לצמצם את השיטה ה"אגרסיבית"של נעילת כל אובייקט לפני ביצוע השינויים אליו. במקרים מסוימים, הנעילות הללו יכולות להיות מיותרות, ומשפיעות בצורה דרמטית על זמן הריצה. במקום זאת, תכנון אופטימיסטי מנסה לבצע פעולות עם סנכרון חלקי (אם בכלל), ומנצל שחזור של פעולות (Rollback), בעת נפילה/שגיאה.

תכנון אופטימיסטי אינו מתאים לכל בעיה, אולם הוא יכול להיות יעיל במקרים המקיימים את התנאים הבאים:

- כאשר הסיכוי לשגיאות/נפילות/כשלים הוא נמוך יחסית
- כאשר תקורת הסנכרון במקרה הרגיל הינה גבוהה מאוד
- כאשר ניתן לזהות מתי פעולה בוצעה בצורה לא נכונה / לא חוקית
- כאשר ניתן לחזור / לשחזר את המצב לפני השגיאה

נראה שתי שיטות למימוש תכנון אופטימיסטי, ונשווה ביניהם באספקטים השונים:

	"פעולות זמניות" (Provisional Action)	חזרה לאחור / התאוששות (Rollback / Recovery)
מאפיינים כלליים	• הפעולה "כאילו" מבוצעת, והביצוע וההשלכות שלה נדחות עד לנקודה שנפילה/כשלון של הפעולה נפסלת. • בצורה זו קל יותר לנהל מתודות אשר רק מעדכנות משתנים (Instance variables). בשיטה הנ"ל, במקום לעדכן/לשנות את האובייקט, מבוצע עדכון על עותק זמני/צדדי (*Shadow copy*). • מדי פעם, מבוצע Commit לשינויים, והאובייקטים מוחלפים בעותקים הזמניים שעליהם פעלנו. הבעיה כאן היא שרוב הזמן צריך לדאוג שהאובייקטים המקוריים לא השתנו **כלל**. • עקב המגבלה הזאת, ועל מנת להימנע מבאגים, עדיף לעשות את המחלקות בלתי ניתנות לשינוי (*Immutable*).	• במודל הנ"ל, הפעולה אכן מבוצעת במקום. • ניתן לבטל (Undo) את ההשפעות של כל פעולה שנעשתה. על כל פעולה שמופעלת במסגרת של "חשיבה אופטימיסטית" צריכה להיות פעולה הפוכה (inverse action). • על מנת שיהיה אפשר לתחזק את אפשרות הביטול ולהתאושש מנפילות, יש שמירה של לוג (log) של כל הפעולות שבוצעו, על מנת לאפשר חזרה לאחור בסדר הפוך. • מדי פעם מתבצע מעבר על הלוג, על מנת לבדוק שפעולות אכן התבצעו כהלכה, על מנת שניתן יהיה להסיר אותן מהלוג. • המודל מאפשר שינויים בו-זמניים כל עוד ניתן לבדוק אם אלו בוצעו בצורה נכונה.

השוואה	• עבודה על *עותק* קלה יותר למימוש • *Shadowing* לא מאפשר שינויים בו-זמניים (Concurrent) [ניתן לפתור ע"י שיפור גרעיניות המצב, או הוספת פעולת *מיזוג* מצבים]	• אין הבטחה להתקדמות (ללא אמצעי זהירות מתאימים, החוטים יכולים לבצע rollback לנצח, בניגוד ל-Shadowing, אשר שם לפחות חוט אחד יתקדם)
מתודות עזר (התמודדות עם קינון פקודות ורקורסיה)	• מתבצעות פעולות על העותק של המשתנה שנשלח כפרמטר לפונקציה. המתודה הראשית אחראית לבצע Commit לעותק כאשר היא מוכנה	• מתודת עזר אשר מחזירה רשימה של פעולות ביטול (Undo) לשינויים שבוצעו. הפונקציה הקוראת מוסיפה את הרשימה הנ"ל לרשימה שלה, ומוודאת את נכונות מצבי האובייקטים מעת לעת. Rollback מבוצע כאשר מתגלת בעיה
גילוי בעיות	• יש צורך לוודא כי האובייקט המקורי לא השתנה מאז שהתחלנו לבצע את השינויים (שהופעלו עד כה על העותק) – זאת ניתן לעשות בעזרת אופרטור השוואה מתאים, או שימוש בחותמות זמן וסיריאליזציה	• חייב להיות סט של תנאים מוגדרים היטב לגבי מצבו של אובייקט כך שכלל התנאים האלה נכונים אמ"מ כל הפעולות על האובייקט עד עכשיו הופעלו בצורה נכונה ועקבית

Volatile– סימון לקומפיילר כי השדה הנ"ל ניתן לגישה ללא סנכרון. כלומר, מודיע ל-JVM לא להפעיל אופטימיזציות על המשתנה, אלא לכתוב אותו ישר לתוך הזיכרון.

טיפול בשגיאות (Failure handling)

בעת שגיאה, המתכנת יכול להחליט באם להודיע על כשלון או לזרוק חריגה. בנוסף, ניתן לבצע את הפעולה מחדש עד להצלחה (או לזנוח אותה). הגישה של להפעיל את הפקודה מחדש היא לא בהכרח נכונה, מכיוון שנוכל להיכנס למצב של Livelock (כלומר, חוטים מסויימים יכנסו למעגל rollback-retry), לכן, ניתן לוותר על הפעולה לזמן כלשהו (Yield) ולנסות לבצע אותה שוב לאחר מכן, או ניתן להשתמש בדעיכה אקספוננציאלית של ניסיונות הפעלה מחדש, על מנת להתמודד עם פעולות שיוכלו להיכנס ל-Livelock (Exponential Backoff). כלומר, לאחר כל כישלון, התהליך/החוט נכנס להמתנה (sleep) לזמן אקרעי אך חסום אשר עולה אקספוננציאלית.

MPI – Message Passing Interface

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/MPI1.pdf

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/MPI2.pdf

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/MPI3.pdf

דרגה (Rank)– מס' סידורי ייחודי עבור תהליך (מספור מתחיל מ-0)

קבוצה (Group)– אוסף מסודר (ordered) של תהליכים. גם לו יש מס' סידורי ייחודי, אשר ניתן ע"י המערכת ומקושר ל-*Communicator*. באופן התחלתי, כל התהליכים הינם חלק בקבוצה המוגדרת ע"י ה-Communicator**MPI_COMM_WORLD**

Communicator– מגדיר סט של תהליכים (קבוצה) אשר יכולים לתקשר בין אחד לשני.
(**MPI_COMM_WORLD** מכיל כברירת מחדל את כל התהליכים במסגרת ה-MPI)

חוצץ מערכת (System buffer)– אזור זיכרון לשימוש פנימי של מנגנון MPI על מנת לשמור הודעות לצורך שליחה/קבלה. החוצץ מאפשר בצורה הזו תקשורת א-סינכרונית בין התהליכים השונים, והוא נשלט לגמרי ע"י המערכת.

חוצץ אפליקציה (Application Buffer)– אזור זיכרון לשימוש האפליקציה, המשמש לשמירה של המידע לצורכי שליחה/קבלה (הכנה לשליחה למעשה), הוא נשלט לגמרי ע"י האפליקציה.

כל הודעה אשר נשלחת במסגרת MPI מכילה שני חלקים:

1) *Data*– מכילה את אזור התחלת המידע שיש להעתיק/להעביר/לקבל, מספר האלמנטים המשתתפים, וכן סוג המידע.
2) *Envelope*– מכילה את המקור או היעד (בעזרת הדרגה, ניתן להשתמש ב-MPI_ANY_SOURCE כאשר לא ידוע או לא משנה מהו המקור), את התגית (Tag), אשר מוגדרת ע"י המשתמש, על מנת להבדיל בין סוגי ההודעות השונות שיכולות להישלח (ניתן להשתמש ב-MPI_ANY_TAG כאשר לא משנה התגית של ההודעה), וכן ה-Communicator, אשר מגדיר את הקבוצה שבתוכה מתרחשת התקשורת.

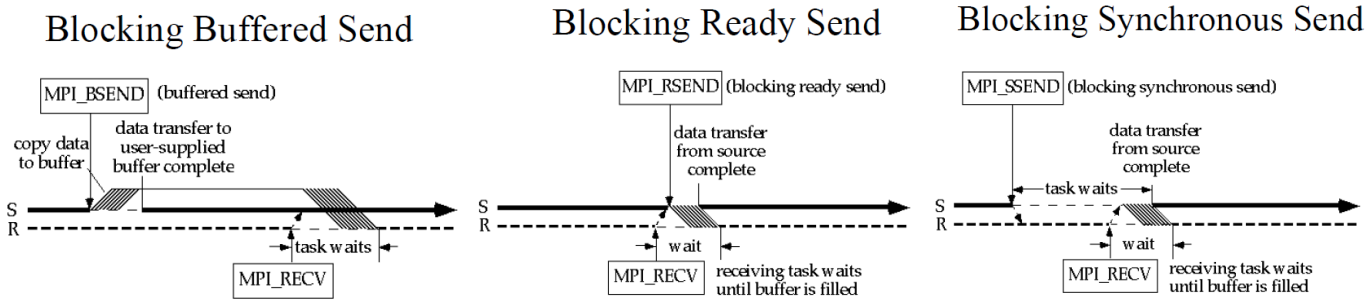
פונקציות כלליות

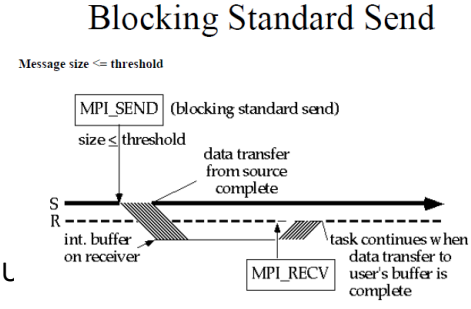
MPI_Init(argc,argv)	מאתחל את סביבת התקשורת של MPI, צריך להיות הקריאה הראשונה בכל תהליך MPI
MPI_Finalize()	סוגר בצורה נאותה את סביבת התקשורת של MPI, צריך להיות הקריאה האחרונה בכל תהליך MPI
MPI_Comm_Rank(comm,&rank)	מחזיר את המס' הסידורי של התהליך הנוכחי בתוך ה-Communicator
MPI_Comm_size(comm,&rank)	מחזיר את מס' התהליכים בתוך ה-Communicator שאליו משתייך התהליך הנוכחי

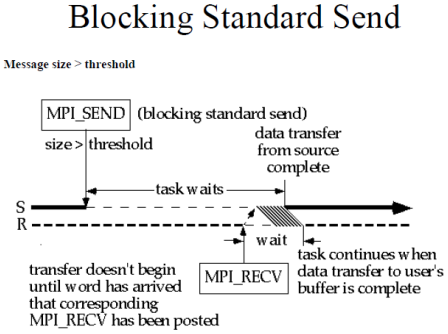
תקשורת P2P– חוסמת מול לא-חוסמת

<u>תקשורת חוסמת</u>

- פרוצדורת התקשורת חוסמת את התהליך עד שפעולות מסוימות התבצעו, ולאחר מכן מאפשרת לו להמשיך. *בשליחה*, רק כאשר כל המידע אשר אמור להישלח הועתק בצורה בטוחה לחוצץ המערכת, ניתן יהיה להשתמש בחוצץ האפליקציה שוב.
- פקודות:
  - *Standard send* ***[MPI_Send]*** – שיטת השליחה הבסיסית ביותר במסגרת MPI לצורך העברת מידע מתהליך אחד לתהליך אחר. ישנו שימוש ב-Threshold (שהוא גודל המקום הפנוי בחוצץ המערכת). אם ההודעה שרוצים לשלוח קטנה מה-Threshold, אזי ההודעה מועתקת לחוצץ המערכת, והתהליך יכול להמשיך לעבוד. אולם, אם ההודעה גדולה מה-Threshold, התהליך נחסם עד אשר יהיה מקום ב-Threshold להודעה, או עד אשר בצד השני יהיה איתות של "קבלה", ורק אז המידע יישלח.*[פשרה תלויית מימוש בגרסת MPI]*
  - *Synchronous send* ***[MPI_Ssend]*** – השליחה הנ"ל חוסמת את התהליך השולח עד שבצד השני הופעל איתות של "קבלה", ורק אז התהליך השולח יכול להמשיך בעבודתו. המטרה של השימוש בשליחה הנ"ל הוא אם רוצים לוודא שאכן ההודעה נשלחה. (אם היה איתות "קבלה" מתאים לפני הקריאה לשליחה הנ"ל, אזי לא תהיה המתנה כלל).*[שליחה בטוחה, ומאפשרת פורטביליות]*
  - *Ready send* ***[MPI_Rsend]*** – השליחה הנ"ל מתקיימת רק כאשר איתות "קבלה" מתאים מהצד השני הופעל כבר. אחרת השליחה תוגדר כשגיאה ולא תתבצע.*[תקורת השימוש הנמוכה ביותר]*
  - *Buffered send* ***[MPI_Bsend]*** – בסוג השליחה הנ"ל, האפליקציה מקציבה חוצץ לשימוש המערכת, אשר אליו מועתק תוכן ההודעה, ונשלח ברגע שיש איתות "קבלה" מתאים. עקב השימוש בחוצץ, התהליך יכול לחזור לעבודתו לאחר ההעתקה.*[מפרידה בין השולח לבין המקבל, ומאפשרת שליטה כלשהי של המתכנת]*
  - *Standard receive* ***[MPI_Recv]*** – שיטת הקבלה הבסיסית (איתות ה"קבלה") במסגרת MPI, התהליך ימשיך להתבצע לאחר שההודעה עצמה תתקבל.*[פשרה תלויית מימוש בגרסת MPI]*
- נשים לב שבשיטה הנ"ל נוכל להגיע למצב של deadlock, אם למשל שני תהליכים רוצים לבצע שליחה אחד לשני, ואין עבור אף אחד מההודעות מספיק מקום בחוצץ המערכת. על מנת להימנע ממצב כזה ניתן לתזמן סדר מתאים בין השליחות, להשתמש בשליחה בעזרת חוצץ מוקצה מראש, או להשתמש בקריאות לא חוסמות.

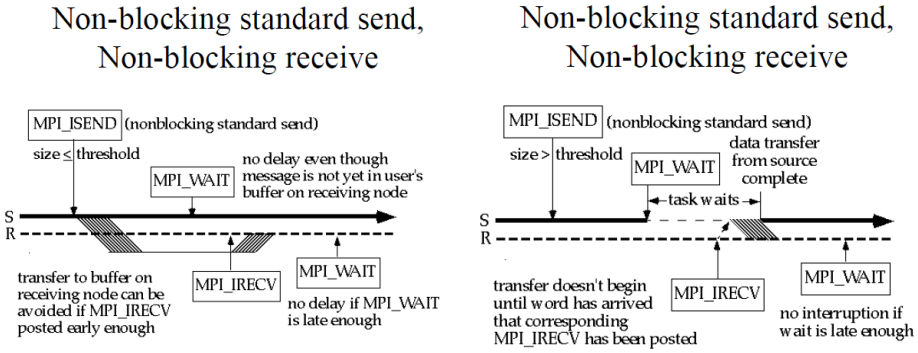






<u>תקשורת לא-חוסמת</u>

- פרוצדורת התקשורת חוזרת מיד מבלי לחסום את התהליך, אולם עקב כך לא בטוח להשתמש בחוצצי האפליקציה לפני סיום השליחה/קבלה (לשם כך ניתן להשתמש בקריאות בדיקה). היתרון באופי התקשורת הנ"ל הוא אפשרות השילוב בין חישוב לתקשורת. בשימוש בתקשורת לא חוסמת, ניתן להקטין את תקורת המערכת, להימנע מחבק ולהקטין את תקורת התקשורת. כאן מתבצע שימוש ב-***MPI_Isend***, ו-***MPI_Irecv*** וישנו שימוש במתודות שונות על מנת לקבל אינפורמציה לגבי השליחות / קבלות בזמן כלשהו.
- פקודות (בדיקה):
  - *Wait**[MPI_Wait]***– בדיקה אשר <u>חוסמת</u> את התהליך עד שהפעולה המתאימה (לפי הפרמטרים) אכן בוצעה. הפקודה הנ"ל שימושית הן עבור שליחות והן עבור קבלות לא-חוסמות.
  - *Test**[MPI_Test]***– בדיקה אשר <u>לא חוסמת</u> את התהליך, אלא רק בודקת אם הפעולה המתאימה (לפי הפרמטרים) אכן בוצעה.הפקודה הנ"ל שימושית הן עבור שליחות והן עבור קבלות לא-חוסמות.
  - *Probe**[MPI_Probe]***–פקודה אשר מתאימה עבור הצד <u>המקבל</u>, הפקודה <u>חוסמת</u> את המקבל עד אשר מגיעה הודעה (מכל אחד!), אשר מוכנה לעיבוד.

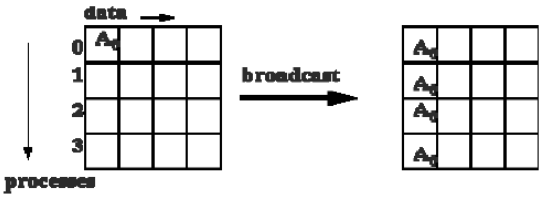


מבני נתונים לשליחות מידע

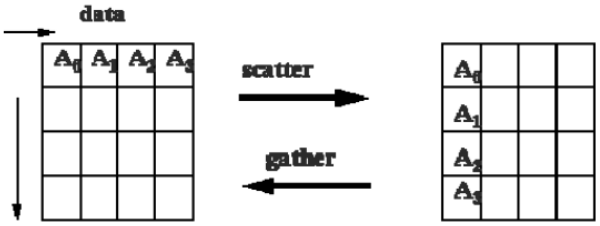
ב-MPI יש אפשרות לשלוח הודעות מורכבות, מעבר להגדרות הבסיסיות שקיימות בשפה (כמוMPI_INT וכולי). ניתן להשתמש בפונקציונליות של MPI על מנת לבנות מבנה נתונים (Derived data type) אשר יישלח, אולם יש להיזהר, לא ניתן להגדיר, לדוגמא,struct פשוט ולהשתמש בו בשליחה, מכיוון שMPI- משמש לא רק לתקשורת בין תהליכים במחשב עצמו, אלא גם לתקשורת בין מחשבים מרוחקים אשר הארכיטקטורה של מבני הנתונים הבסיסיים והפרימיטיביים היא שונה, ובנוסף, לא ידוע לנו מה השינויים/אופטימיזציות שהקומפיילר יחליט לבצע בצד המקבל. בנוסף, ניתן להשתמש ב-MPI_Datatype_contiguous על מנת ליצור רצפים (מערכים) של מבני נתונים בסיסיים או בנויים ידנית (Derived) לצורך שליחות שונות. ע"י הגדרה ושימוש בפונקציות של MPI ניתן להגדיר מבני נתונים יוניפורמיים אשר לא תלויים בתאימות בין הצד השולח לצד המקבל.

תקשורת קולקטיבית

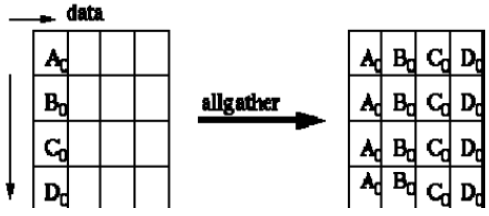
- נוסף לשליחות P2P, לפעמים ישנו צורך בתקשורת אשר תגיע ליותר מיעד אחד, לדוגמא, כאשר רוצים לחשב כמות גדולה של מידע, וכל תהליך מקבל אחריות על חלק כלשהו של מידע, אזי יש צורך לשלוח לכל התהליכים את המידע, על מנת שכ"א יבצע את חלקו (ואולי יש צורך לאחר מכן, להחזיר את המידע הנ"ל). חשוב לציין כי יעילות התקשורת הקולקטיבית תלויית מימוש שלMPI, וכן ישנה תקשורת וסנכרון רבים מאוד בשימוש בסוג התקשורת הנ"ל, ולכן היא אינה בהכרח תמיד הפתרון היעיל ביותר.
- פקודות:
  - *מחסום (Barrier) **[MPI_Barrier]*** – קריאת סנכרון, אשר חוסמת את כל התהליכים הפועלים, עד אשר כולם מגיעים לאותה הנקודה. הפקודה צריכה להיקרא ע"י כל מי שנמצא בקבוצה שעליה תקף המחסום.
  - *שידור (Broadcast) **[MPI_Broadcast]*** – תהליך אחד שולח את המידע לכל התהליכים אשר נמצאים איתו בקבוצה כלשהי. הפקודה צריכה להיקרא ע"י כל מי שנמצא בקבוצה הנ"ל, וכן ע"י השולח (מוגדר להיות root) [למעשה, השולח כאן שולח את המידע גם לעצמו]



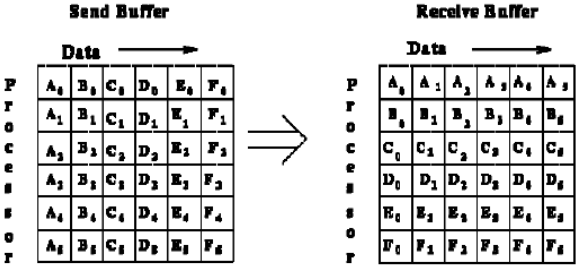
  - *איסוף (Gather) **[MPI_Gather]*** – תהליך אחד (root) אוסף את כל המידע שנשלח מכל שאר התהליכים בקבוצה מוגדרת. לצורך ביצוע הפעולה הזאת, התהליך אשר אוסף את המידע צריך להקצות מערך של תאים, כך שכל תא יוכל להכיל את מבנה הנתונים שמרכיב את ההודעה. ההודעות יתקבלו בתאים לפי המס' הסידורי של התהליכים (rank). כל המשתתפים בשליחה הנ"ל צריכים לקרוא לפקודה הנ"ל.*[קיימת פקודה בשם MPI_GatherV, אשר מאפשרת לכל תהליך לשלוח אל root במיקום לא סימטרי, כלומר, ב-offset כלשהו ביחס למיקום המקורי של המערך של root]*



  - *פיזור (Scatter) **[MPI_Scatter]*** – תהליך אחד (root) שולח את המידע שלו לכל התהליכים מהקבוצה המוגדרת. לצורך ביצוע הפעולה הזאת, התהליך אשר שולח את המידע צריך להקצות מערך של תאים, אשר בכל תא יהיה המידע שצריך להישלח במיפוי של מס' תא אל מס' תהליך (rank).כל המשתתפים בשליחה הנ"ל צריכים לקרוא לפקודה הנ"ל.*[קיימת פקודה בשם MPI_ScatterV, אשר מאפשרת לכל תהליך מקבל לקבל אינפורמציה במקום לא סימטרי ביחס למערך של root, כלומר ב-offset כלשהו ביחס למיקום]*



  - *איסוף כולל (AllGather) **[MPI_Allgather]*** – בדומה לפקודת האיסוף המקורית, אולם כאן אין root, וכולם אוספים את המידע מכולם. *[גם כאן, ישנה אפשרות של שימוש ב-MPI_AllgatherV]*



- *חלוקת מידע קולקטיבית (All to All)* ***[MPI_Alltoall]***– קומבינציה של האיסוף והפיזור. כל התהליכים שולחים את המידע שלהם, ובנוסף כל התהליכים מקבלים אחד מהשני.

- *רדוקציה (Reduction)* ***[MPI_Reduce]***– מתבצע איסוף של כלל הערכים אשר מתקבלים ע"י התהליכים בחוצץ הקלט, לאחר האיסוף מתבצעת רדוקציה לפי הפעולה המוגדרת, והערך נכתב לחוצץ הפלט של התהליך המוגדר כראשי (root). *[ישנו גם שימוש ב-MPI_Allreduce, אשר בו לא רק התהליך root מקבל את תוצאת הרדוקציה, אלא כל התהליכים]*
- *רדוקציית רישא (Prefix reduction)* ***[MPI_Scan]*** - עבור התהליך ה-i, מתבצע איסוף של כלל הערכים בחוצצי הקלט בתהליכים 0,...,i. ערך הרדוקציה על הערכים הללו (לפי הפעולה המוגדרת) נכתב לחוצץ הפלט של התהליך.

CondorDAGman

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/dagman.pdf

- הרצה בסיסית של Condor יכולה לרוץ מקבילית בצורה חלקה אם אין תלויות בין המטלות / השלבים. אולם, כאשר יש תלויות, לא ניתן להריץ זאת בצורה פשוטה. נשתמש בגרפים חסרי-מעגלים (DAG) על מנת לתאר תלויות בין מטלות.
- Condor DAGman הוא למעשה מתזמן עבודות (Personal Job Scheduler). המשתמש מספק את המבנה והעבודות, בתוספת לתלויות (בשפה מוגדרת מראש), וה-DAGman הוא זה שמחליט מהן העבודות שצריכות להישלח ל-Condor בהתאם לתלויות. בנוסף, הוא מנתר את הביצוע של כל אחת מהעבודות, ובעת סיום עבודה מסוימת, הוא מעריך מחדש את התלויות, על מנת לבדוק מהי העבודה הבאה שיש לשלוח.
- כל עבודה שנשלחת ל-Condor, נעטפת בקטע קוד אשר מופעל לפני שהעבודה מתחילה להתבצע (PRE script) וקטע קוד אשר מופעל לאחר שהעבודה מתבצעת (POST script). שני הסקריפטים מופעלים בצורה לוקאלית, וסמנטיקת בדיקת השגיאות היא כדוגמת C (Lazy eval), כלומר PRE && POST. [נניח למשל כי PRE התקיים, והעבודה נכשלה, אולי ב-POST ישנו קטע קוד אשר לא בהכרח קשור לעבודה, אשר אותו נרצה לבצע אפילו אם העבודה נכשלה]. במקרה ובצומת כלשהי התגלה כישלון, אז ננסה לבצע אותה מחדש (כל שאר הצמתים מופעלים כל עוד קיימת תלות), ובמקרה כי השגיאה היא קריטית, הפעולה כולה מופסקת. צריך להבדיל בין שני סוגי נפילות. אם למשל מחשב אשר מבצע מטלה נכבה, זוהי נפילה ש-Condor מחביא, כי אחריותו להפעיל מחדש את העבודה. לכן, DAGman לא יזהה את הנפילה הנ"ל, מכיוון שזאת היא אחריות המערכת. מצד שני, נפילות כגון אי קריאת הקלט, או אי קיום התנאים ב-PRE, הינן נפילות ש-DAGman יכול לזהות, מכיוון שהן בתחום אחריותו של המשתמש.
- במקרה של נפילה, נרצה לתקן את העבודה אשר גרמה לנפילה, ולהמשיך לבצע מאותה הנקודה. בעת אתחול מחשב לדוגמא, schedd לא ייפגע, מכיוון שהוא שומר את המצב הסטאטי שלו על הדיסק. DAGman ייפול יחד עם התור שעליו הוא מאוחסן, ולכן DAGman מבצע לעצמו בדיקה (Checkpoint). לכן, במקרה ועבודה כלשהי נכשלה, יש לייצר DAG חדש, ולסמן את כל העבודות שהספיקו לסיים כ-DONE, וניתן כעת להשתמש ב-DAG הנ"ל על מנת להמשיך לעבוד מהנקודה שהפסקו. כל אלו בתנאי ש-DAGman הספיק לבצע את הדברים הללו לפני האתחול. על מנת לבצע את הבדיקה הזאת, ולהמשיך בעבודה, צריך להכריח את DAGman לבצע בדיקה (Checkpoint), אולם יש לזכור כי DAGman הוא גם עבודה שנשלחת ל-Condor, ולכן מכיוון שמבוצעת בדיקה ל-schedd, ו-DAGman בתור שלו, אזי גם הוא ייבדק.
- ניתן לבצע קינון בין DAGs שונים, על מנת להתמודד עמם בצורה יעילה, יש להשתמש בשזירה (Splicing). בצורה זו נוכל להגדיר sub-DAGs, ו-Condor ישלים את גרף התלויות בשבילנו. כאשר משתמשים בשזירה בין שני DAGs, כל היציאות של ה-DAG אשר מוגדר להיות הורה (PARENT) מחוברות לכניסות של ה-DAG אשר מוגדר להיות הבן (CHILD).
- סינטקס של מפרט של DAG:
  - *JOB X Y* – מגדיר עבודה, אשר השם שלה הוא X, והתיאור שלה הוא בקובץ Y
  - *PARENT X Y Z CHILD A B C* – מגדיר כי X, Y ו-Z הם הורים, ולכל אחד קיימים הבנים A, B ו-C.
  - *SPLICE X Y* – מגדיר שזירה של X, כאשר התיאור שלו הוא בקובץ Y. Condor ישלים בעצמו את ה-subDAG, ומאפשר שימוש ב-X בצורה רגילה בהגדרת התלויות (השימוש ב-PARENT ו-CHILD).

שיטות סנכרון (עבודה על רשימה מקושרת)

http://webcourse.cs.technion.ac.il/236370/Winter2009-2010/ho/WCFiles/08%7Echapter_09%20reduced.ppt

נראה מספר שיטות עבודה עם רשימה מקושרת (זהו מבנה נתונים אשר ניתן לבצע עליו שינויים ושאילתות בו-זמנית).

הרשימה המקושרת תייצג סט (Set). הסט יהיה אוסף כלשהו של איברים (כל איבר יכיל ערך ומפתח), ללא חזרות, ונשתמש בשלושת המתודות add(x), remove(x), contains(x).

נסתמך על אוסף אינווריאנטות (Rep Invariants), אשר נדרוש כי יישמרו לאורך התוכנית, ובפרט עבור כל מתודה (לפני הפעלה ואחרי הפעלה). בדוגמא שלנו, באוסף האינווראינטות נגדיר כי קצה הרשימה (tail) תמיד ישיג מראש הרשימה (head), וכן כי הרשימה ממויינת (מבחינת מפתחות), וכמובן שאין חזרות של מפתחות.

נגדיר את הקבוצה S(head) להיות קבוצת כל ה-x, כך שקיים אינדקס a ישיג מהראש, כך שהערך של a שווה ל-x.

נעילה בגריעיניות גסה (Coarse Grained Locking)

לצורך גישה לרשימה המקושרת עבור כל פעולה שהיא, מתבצעת נעילה של ראש הרשימה. הפתרון הנ"ל הינו הפשוט ביותר. יש כאן שימוש מינימלי במנעולים (רק אחד), והאינווראינטות לעולם לא יפגעו. אולם, עצם העובדה כי קיים מנעול יחיד, אשר אותו כל גישה מחויבת לתפוס, נקבל צוואר-בקבוק רציני אשר ייעכב את כל התהליכים במערכת (לדוגמא, אם תהליך כלשהו רוצה לעשות חיפוש קיום, הוא בכל מקרה יתפוס מנעול, ויתחיל לבצע חיפוש ברשימה. תהליך אחר, אשר גם רוצה לבצע חיפוש, ייאלץ לחכות עד שהתהליך הראשון יסיים את עבודתו).

נעילה בגרעיניות עדינה (Fine Grained Locking)

לכל איבר ברשימה יהיה מנעול, אולם בניגוד לשיטה הקודמת, יש צורך לתפוס רק מנעולים על איברים ספציפיים על מנת לבצע פעולות מסוימות. עקב זאת, ניתן לבצע פעולות במקביל על איברים בלתי-תלויים. אולם, אם לא נשים לב, יכולה להיווצר בעיה במחיקה. לשם כך, באם נרצה למחוק צומת כלשהו, ננעל את הצומת הנ"ל ואת הצומת שלפניו. בצורה זו נוכל למנוע מצב שבו הצומת הבא שאמור לגשר על הפער של המחיקה(כלומר, בהינתן צמתים a-b-c, רוצים למחוק את צומת b ע"י הסרתו וקישור a ל-c) נמחק במקביל ע"י תהליך אחר. בעיה דומה יכולה להיווצר בהוספה. על מנת למנוע את הבעיה, באם נרצה להוסיף צומת כלשהי, ננעל את שני הצמתים הפוטנציאליים אשר אמורים לתפקד כקודם (predecessor) והבא (successor) (כלומר, בהינתן a-c, ונרצה להוסיף את b ביניהם, ננעל תחילה את a ואת c)*[בהוספה, לא מוכרחים לנעות את הבא –successor. מכיוון, שאם חוט אחר ירצה למחוק אותו לדוגמא, הוא ייצטרך לנעול גם את הקודם –predecessor, אשר אותו כבר נעלנו]*. השיטה הנ"ל טובה יותר מהקודמת, מכיוון שחוטים יכולים לנוע על הרשימה במקביל. אולם, יכול להיגרם מצב שבו יש שרשרת ארוכה של תפיסות מנעולים בין החוטים, אשר תיצור המתנה. וכמובן, העובדה כי יש מנעול על כל צומת אינה יעילה (מבחינת צריכת משאבים לדוגמא, למרות שב-Java, לדוגמא, זה מגיע בחינם).

סנכרון אופטימיסטי (Optimistic Synchronization)

העקרון של השיטה הזו הוא לחפש את הצמתים הנדרשים ללא נעילה, ולנעול אותן רק בעת מציאה, ורק אז לבצע את הבדיקות המתאימות. נשים לב, שבמודל הנ"ל מבצעים חיפוש, ורק בעת מציאה מבצעים נעילה. אולם, יכול לקרות מצב שכאשר הגענו ליעד הרצוי, חוט אחר הספיק להגיע ולנעול חלק מהצמתים שאנחנו נזקקים להם, ולבצע שינויים אשר ישפיעו על אופי הפעולה שלנו. על מנת למנוע מצבים כאלו, נבדיל בין השלבים של המציאה והנעילה. לאחר הנעילה נבצע וידוא (Validation) כי המאפיינים אשר היו בעת המציאה של הצמתים הינם גם נכונים כעת. (לדוגמא, אם רוצים להוסיף צומת c בין הצמתים b-d, אזי בעת המציאה ידוע כי b ישיג מהראש, וכן כי b מצביע על d. נבדוק את שני הדברים הללו בעת הנעילה, ואם הם מתקיימים, אזי נמשיך). היתרונות של השיטה הנ"ל היא העובדה שנקודות הבעייתיות (Hot-spots) הינן הצמתים אשר משתתפים בפעולות השונות (מספר מצומצם, בין 2 ל-3 צמתים), ובנוסף, אין תחרות (והמתנה) על המעברים השונים על הרשימה (מכיוון שאין נעילה עד למציאה ממש של הצמתים הנדרשים). מכאן הרי כי ישנו שיפור בביצועים ובמקביליות, ויש מספר נמוך יותר של תפיסות/שחרורי מנעולים. אולם, נשים לב כי באלגוריתם של הפעולות יש צורך לעבור על הרשימה פעמיים, וישנה נעילה של צמתים בהפעלת המתודה contains(), כאשר סטטיסטית היא המתודה אשר נקראת הכי הרבה. לסיכום, שיטה זו תהיה טובה אם המחיר עבור מעבר כפול של הרשימה ללא נעילות הוא נמוך יותר ממעבר יחיד עלה רשימה עם נעילות.

רשימה עצלה (Lazy List)

בשיטה הנ"ל, נשתמש בביט סימון עבור כל צומת, אשר ייצג אם היא מחוקה או לא. מחיקה לא תתבצע ישר (אלא בצורה עצלה). בנוסף, ישנה כאן סריקה יחידה של הרשימה, והמתודה contains() לא נזקקת לנעילה. במחיקה של איבר, נבצע את אותה הסריקה (ותפיסת המנעול של האיבר הנמחק, והאיבר לפניו), אולם לפני מחיקה פיזית, נסמן את האיבר כמחוק, ורק לאחר מכן נבצע מחיקה פיזית שלו ע"י כיוון החץ של הקודם אל הבא ברשימה. (המתודות האחרות פועלות בצורה זהה). כעת, הוידוא יבוצע בצורה מעט שונה בניגוד לשיטה הקודמת, במקום לסרוק מחדש את הרשימה, תתבצע בדיקה האם האיבר הנוכחי (שיש להסיר) לא מסומן, וכן שהאיבר לפניו לא מסומן, ושאכן האיבר הקודם מצביע לאיבר הנוכחי.לצורך זאת, נוסיף להגדרה של S(head) את הדרישה כי a לא יהיה מסומן. כעת, האינווריאנטה שלנו תתעדכן, ונדע כי אם צומת כלשהו לא מסומן, וניתן להגיע אליו מהראש, אזי בהכרח ניתן להגיע אליו גם מהאיבר אשר מוגדר להיות לפניו.אם נשתמש בעובדה כי הרשימה ממוינת (Ordered) וניעזר בביט הנוסף, נוכל לדעת כי אם הצומת לא מסומן, אזי האיבר שבתוכו בסט, ואם הצומת מסומן או לא קיים, אזי האיבר לא בסט– ולכן, אין צורך לבצע נעילות ב-contains().

לסיכום, השיטה הנ"ל שימושית מאוד מכיוון ש-contains() מקיים אחוז גדול מאוד מההפעלות של המתודות, ובנוסף, מתודות אשר לא נזקקות לאותם האיברים לצורך ביצוע הפעולה, לא זקוקות למעבר נוסף על הרשימה. מצד שני, מתודות אשר נזקקות לאיברים חופפים לצורך ביצוע פעולה, יצטרכו לעבור שוב על הרשימה ולבדוק שינויים (על מנת לבצע את הפעולה בהתאם לאילוצים החדשים), ובנוסף, אם חוט מסוים יתעכב, יווצר "פקק" (Traffic Jam, לדוגמא, כאשר חוט מסוים נכנס לקטע הקריטי, ונקלע לכל בעיה אפשרית [Cache miss, page fault, software error וכו'], אזי כל החוטים האחרים יתעכבו בגללו).

רשימות ללא-נעילות (Lock-free Lists)

השיטה נמנעת לחלוטין בשימוש במנעולים, במקום זאת נשתמש בפעולות אטומיות. נשתמש בפעולה compareAndSet()‎, אשר בהינתן צומת שצריך להתווסף / להימחק, תבדוק את הצומת שלפניו, ותעדכן את המצביע שלו בהתאם. אולם, כאן נוכל להגיע למצב שעקב המקביליות, נגיע להצבעה לצמתים שהוסרו (לדוגמא ברשימה a-b-c-d, תתבצע במקביל הסרה של b ושל c. בשלב הראשוני, לאחר CAS נקבל כי a מצביע ל-c ו-b מצביע ל-d. לאחר מכן b יוסר, אולם נגיע לבעיה כאשר נרצה להסיר את c). על מנת להתגבר על הבעיה, נשתמש ב-AtomicMarkableReference. לכל צומת, יהיה דגל אשר מציין האם הוא נידון למחיקה או לא. באלגוריתם המחיקה, נדליק את הדגל של הצומת הנמחק, נבצעCAS על הצומת שלפני, ואם הצומת אינו מסומן בדגל (למקרה שאולי חוט אחר רצה להסיר אותו במקביל), אזי נעדכן את המצביע שלו בהתאם. אחרת (כלומר, הצומת שלפני מסומן למחיקה), הפעולה תיכשל, ולא נעשה דבר. יכול להשתמע כי הפעולה נכשלה, והצומת לא נמחק, אולם צריך לזכור כי הוא עדיין מסומן, ולכן, כאשר חוט אחר יעבור על הרשימה, הוא יוכל להסיר אותו באיזושהי איטרציה אחרת [כאשר נגיע לצומת כזאת, נבצע CAS על הצומת שלפניו, ונמשיך באלגוריתם בהתאם]. (לדוגמא, הרשימה היא a-b-c-d, וחוט אחד רוצה להסיר את b וחוט אחר רוצה להסיר את c. b ו-c מסומנים כמחוקים – בעזרת הדגל, לאחר מכן מתבצע CAS על a ועל b [הקודמים], ו-a אכן מעודכן להצביע על c, ו-b פשוט נמחק. באיטרציה כלשהי, c יימחק מכיוון שהוא מסומן).גם כאן, נוכל להגיע לבעיה, כאשר חוט כלשהו ינסה להוסיף צומת, והצומת הקודם יוגדר להיות צומת שסומן למחיקה (לדוגמא a-b-c-d, כאשר c סומן למחיקה, ועדיין לא בוצע לו CAS על מנת למחוק, וחוט אחר הוסיף פתאום את c’ בין c ל-d), על מנת להתגבר על כך, נגדיר כי גם הדגל וגם המצביע לצומת הבא (next) ייבדקו ב-CAS.

MPI מול OpenMP

http://nf.nci.org.au/training/MPIProg/slides/allslides.html

	MPI	OpenMP
יתרונות	פורטביליות	מקבול אינקרמנטלי – ניתן למקבל קוד סיריאלי קיים בשלבים
	אין צורך בתאימות מצד הקומפיילר	אוסף פשוט של פקודות
	אין דרישות חומרה מיוחדות	המידע משותף, אין צורך להקצות מידע עבור כל אחד בנפרד ולדאוג לעקביות
	גמישות – ניתן להשתמש ב-MPI עבור כמעט כל מודל של מקביליות	פעולות חלוקה (או על חלקים) של מערכים הן פשוטות מאוד
	אין מידע משותף, המתכנת לא צריך לדאוג לגבי גישות של חוטים/תהליכים שונים לאותו מידע	
חסרונות	"הכל או לא כלום", קשה מאוד למקבל קוד סיריאלי בשלבים. יש צורך לבנות את הקוד לפי MPI מהבסיס	יש צורך בקומפיילר תומך
	אין מידע משותף, יש צורך להשתמש במבני נתונים שניתנים לשליחה/קבלה	יש צורך בחומרה מרובת-ליבות אשר תומכת בזיכרון משותף
	תוספת הקוד גדולה יחסית	המידע משותף, יש צורך לדאוג לכך שלא יהיו גישות במקביל לאותו המידע
	פעולות חלוקה על מערכים מחולקים יכולה להיות מסובכת (עבודה עם אינדקסים / חלוקות / תחום אחריות של כל מעבד)	צריך לדאוג/להחליט לגבי חלקי המידע אשר יהיו משותפים, ואלה שיהיו פרטיים
		לא מסוגל להתמודד עם מודלים כגון מנהל/עובד (עדיין?)
		בדר"כ לא סקלבילי [מייצר יותר נקודות סנכרון]
		לא מתאים לעבודה עם מבני נתונים לא טריוויאליים כגון רשימות מקושרות ועצים