# การเพิ่มผลิตภาพของการทดสอบซอฟต์แวร์แบบกระจาย โดยใช้จุดตรวจสอบ

**นายภูริเดช สุดสี**

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์
มหาวิทยาลัยเทคโนโลยีสุรนารี
ปีการศึกษา 2560

# A PRODUCTIVITY IMPROVEMENT OF DISTRIBUTED SOFTWARE TESTING USING CHECKPOINTS

**Bhuridech Sudsee**

**A Thesis Submitted in Partial Fulfillment of the Requirements for the**

**Degree of Master of Engineering in Computer Engineering**

**Suranaree University of Technology**

**Academic Year 2017**

# การเพิ่มผลิตภาพของการทดสอบซอฟต์แวร์แบบกระจายโดยใช้จุดตรวจสอบ

มหาวิทยาลัยเทคโนโลยีสุรนารี อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญามหาบัณฑิต

คณะกรรมการสอบวิทยานิพนธ์

(รศ. ดร.กิตติศักดิ์ เกิดประสพ)
ประธานกรรมการ

(ผศ. ดร.ชาญวิทย์ แก้วกสิ)
กรรมการ (อาจารย์ที่ปรึกษาวิทยานิพนธ์)

(ผศ. ดร.ปรเมศวร์ ห่อแก้ว)
กรรมการ

(ศ. ดร.สันติ แม้นศิริ)
รองอธิการบดีฝ่ายวิชาการและพัฒนาความเป็นสากล

(รศ. ร.อ. ดร.กนต์ธร ชำนิประศาสน์)
คณบดีสำนักวิชาวิศวกรรมศาสตร์

ภูริเดช สุดสี : การเพิ่มผลิตภาพของการทดสอบซอฟต์แวร์แบบกระจายโดยใช้จุดตรวจสอบ (A PRODUCTIVITY IMPROVEMENT OF DISTRIBUTED SOFTWARE TESTING USING CHECKPOINTS) อาจารย์ที่ปรึกษา : ผู้ช่วยศาสตราจารย์ ดร.ชาญวิทย์ แก้วกสิ, 116 หน้า.

จากความก้าวหน้าของเทคโนโลยีการเก็บข้อมูลและอัตราการสร้างข้อมูลที่เกิดขึ้นได้อย่างรวดเร็วในปัจจุบันทำให้เกิดยุคของข้อมูลขนาดใหญ่ แต่เครื่องมือที่ออกแบบมาก่อนหน้านั้นพบว่าไม่เหมาะสำหรับการประมวลผลข้อมูลขนาดใหญ่ที่เก็บอยู่ในปัจจุบัน ดังนั้นเพื่อที่จะประมวลผลข้อมูลเหล่านั้นจึงมีการพัฒนาเครื่องมือที่ใช้ประมวลผลงานในลักษณะงานที่ต้องการประมวลผลข้อมูลที่สามารถขยายได้ เครื่องมือหนึ่งที่เป็นที่นิยมในการใช้งานคือ Apache Spark ซึ่งเน้นการประมวลผลบนหน่วยความจำ ทำให้ทำงานได้อย่างรวดเร็ว แต่อย่างไรก็ตามพบว่ายังไม่มีกลไกในการควบคุมคุณภาพซอฟต์แวร์ที่สามารถทดสอบกับกรณีทดสอบบนข้อมูลจริงที่มีขนาดใหญ่ได้ นักพัฒนาจึงเลือกที่จะใช้ข้อมูลตัวแทนชุดข้อมูลมาเพื่อทดสอบ ทำให้นักพัฒนาบางรายเลือกที่จะไม่ทดสอบซอฟต์แวร์เลยเพื่อลดความยุ่งยากและต้องใช้เวลาทดสอบนาน หรือกระทั่งไม่สามารถทดสอบได้บนเครื่องคอมพิวเตอร์เครื่องเดียว

งานวิจัยในเอกสารนี้จึงได้นำเสนอ Distributed Test Checkpointing (DTC) สำหรับกรอบงาน Apache Spark ซึ่งจะทำให้สามารถทดสอบซอฟต์แวร์เป็นกรณีทดสอบแบบเชิงหน่วย บนข้อมูลขนาดใหญ่ที่ยังคงความถูกต้อง และใช้เวลาทดสอบที่ผู้ใช้สามารถยอมรับได้ จากการทดลองพบว่ากรอบงาน DTC จะใช้เวลาทำงานใกล้เคียงหรือสูงกว่าเมื่อเทียบกับกลไกสร้างจุดตรวจสอบดั้งเดิมของ Spark ในกรณีทดสอบแรก และในกรณีทดสอบถัดมากลับสามารถลดระยะเวลาได้อย่างมาก กรณีการตั้งค่ากรอบงาน DTC ที่ดีที่สุดพบว่าสามารถลดเวลาการประมวลผลได้ถึงร้อยละ 450 ถึงร้อยละ 500 และยังพบอีกว่าสามารถลดการใช้พื้นที่เก็บจุดตรวจสอบได้มากกว่า 19.7 เท่า

สาขาวิชา วิศวกรรมคอมพิวเตอร์ ลายมือชื่อนักศึกษา __________
ปีการศึกษา 2560 ลายมือชื่ออาจารย์ที่ปรึกษา __________

BHURIDECH SUDSEE : A PRODUCTIVITY IMPROVEMENT OF DISTRIBUTED SOFTWARE TESTING USING CHECKPOINTS. THESIS ADVISOR : ASST. PROF. CHANWIT KAEWKASI, Ph.D., 116 PP.

DISTRIBUTED CHECKPOINTING/APACHE SPARK/BIG DATA TESTING/SOFTWARE TESTING;

The advancement of storage technologies and the fast-growing number of generated data have made the world moved into the Big Data era. In this past, we had many data mining tools but they are inadequate to process Data-Intensive Scalable Computing workloads. The Apache Spark framework is a popular tool designed for Big Data processing. It leverages in-memory processing techniques that make Spark up to 100 times faster than Hadoop. Testing this kind of Big Data program is time consuming. Unfortunately, developers lack a proper testing framework, which cloud help assure quality of their data-intensive processing programs, while saving development time.

We propose Distributed Test Checkpointing (DTC) for Apache Spark. DTC applies unit testing to the Big Data software development life cycle and reduce time spent for each testing loop with checkpoint. From the experimental results, we found that in the subsequence rounds of unit testing, DTC dramatically speed the testing time up to 450 – 500% . In case of storage, DTC can cut unnecessary data off and make the storage 19.7 times saver than the original checkpoint of Spark.

School of Computer Engineering		Student's Signature ____________

Academic Year 2017		Advisor's Signature ____________

# กิติกรรมประกาศ

วิทยานิพนธ์เล่มนี้สำเร็จลุล่วงไปด้วยดี เนื่องจากได้รับความช่วยเหลือเป็นอย่างดียิ่ง ทั้งในด้านวิชาการ ด้านการดำเนินการวิจัยและกำลังใจที่ดีจากบุคคลต่าง ๆ ได้แก่ ผู้ช่วยศาสตราจารย์ ดร. ชาญวิทย์ แก้วกสิ อาจารย์ที่ปรึกษาวิทยานิพนธ์ ที่ให้คำปรึกษาในการแก้ไขปัญหา รวมทั้งคอยดูแลอบรมทั้งทางด้านการเรียน กระบวนการทำงานวิจัย และการใช้ชีวิต รองศาสตราจารย์ ดร.กิตติศักดิ์ เกิดประสพ รองศาสตราจารย์ ดร.นิตยา เกิดประสพ ผู้ช่วยศาสตราจารย์ ดร.พิชโยทัย มหัทธนาภิวัฒน์ ผู้ช่วยศาสตราจารย์ ดร.ปรเมศวร์ ห่อแก้ว และ ดร. พิชญา แก้วกสิ ที่กรุณาให้คำแนะนำกระบวนการทำงานวิจัย และการเขียนวิทยานิพนธ์ฉบับนี้ ขอขอบคุณ นายปฐวี เงินไทย และเพื่อน ๆ บัณฑิตศึกษาทุกท่านที่คอยสนับสนุน เป็นกำลังใจ อีกทั้งยังยังใครความช่วยเหลือเป็นอย่างดี ขอขอบคุณ เพื่อน พี่ น้อง ศิษย์เก่าวิศวกรรมคอมพิวเตอร์ มหาวิทยาลัยเทคโนโลยีสุรนารี ที่ช่วยเหลือและคอยเป็นที่ปรึกษา ระหว่างที่ได้ศึกษาอยู่ที่มหาวิทยาลัยเทคโนโลยีสุรนารี

สุดท้ายนี้ผู้วิจัยขอขอบคุณคณาจารย์ทุกท่านที่ได้ประสิทธิ์ประสาทวิชาความรู้ต่าง ๆ ทั้งในอดีตและปัจจุบัน และที่สำคัญอย่างยิ่งคือขอกราบขอบพระคุณบิดา มารดา ที่ให้ความรัก กำลังใจ การสนับสนุน และการอบรมสั่งสอนมาอย่างดีโดยตลอด รวมถึงเป็นต้นแบบการใช้ชีวิตซึ่งเป็นแรงบัลดาลใจอันยิ่งใหญ่ จนทำให้ผู้วิจัยประสบความสำเร็จในชีวิตเรื่อยมา

ภูริเดช สุดสี

# สารบัญ

หน้า

**บทที่**

# สารบัญ (ต่อ)

หน้า

# สารบัญ (ต่อ)

**หน้า**

# สารบัญตาราง

**ตารางที่** **หน้า**

# สารบัญตาราง (ต่อ)

**ตารางที่** **หน้า**

# สารบัญตาราง (ต่อ)

**ตารางที่** **หน้า**

# สารบัญตาราง (ต่อ)

**ตารางที่** **หน้า**

# สารบัญรูป

**รูปที่** **หน้า**

# สารบัญรูป (ต่อ)

**รูปที่** **หน้า**

# สารบัญรูป (ต่อ)

**รูปที่** **หน้า**

# คำอธิบายสัญลักษณ์และคำย่อ

MPI = Message Passing Interface
CPU = Central Processing Unit
GPU = Graphic Processing Unit
HTTP = Hypertext Transfer Protocol
SHA-1 = Secure Hash Algorithm 1
VM = Virtual Machine
JVM = Java Virtual Machine
RDD = Resilient Distributed Dataset
LLVM = Low Level Virtual Machine
AST = Abstract Syntax Tree
API = Application Programing Interface
Spark = Apache Spark
Hadoop = Apache Hadoop
FIFO = First-In, First-Out
D-Stream = Discretized Streams
HDFS = Hadoop Distributed File System
ASC = Automatic Spark Checkpointing
DC = Distributed Collection

# บทที่ 1

# บทนำ

## 1.1 ที่มาและความสำคัญของปัญหาการวิจัย

จากความก้าวหน้าของเทคโนโลยีอินเทอร์เน็ตรวมถึงพื้นที่เข้าถึงที่มีมากขึ้นกว่าในอดีต และเทคโนโลยีการเก็บข้อมูลในปัจจุบันที่สามารถเก็บข้อมูลจำนวนมากไว้ในหน่วยเก็บที่ขนาดเล็กลง อีกทั้งแนวโน้มราคาต่อหน่วยของอุปกรณ์ประเภทเดียวกันยังจะต่ำลงเรื่อย ๆ เมื่อเวลาผ่านไป (Komorowski, 2014) ความก้าวหน้าดังกล่าวทำให้เกิดการสร้างและจัดเก็บข้อมูลที่เกิดขึ้นเหล่านี้ไว้ในระบบเพื่อรอการประมวลผล สังเคราะห์ความรู้ออกจากข้อมูลเพื่อทำให้เกิดความได้เปรียบทางเศรษฐกิจต่อคู่แข่งหรือวิเคราะห์ข้อมูลด้านความมั่นคงของรัฐ เป็นต้น คาดการณ์ว่าในแต่ละวันมีการสร้างข้อมูลใหม่เกิดขึ้นถึง 2.5 เอกซะไบต์ ($2.5x10^{18}$ ไบต์) และข้อมูลร้อยละ 90 เกิดขึ้นภายในระยะเวลาเพียง 2 ปีล่าสุด (IBM, 2013) ซึ่งเป็นปริมาณข้อมูลขนาดมหาศาลที่เกิดขึ้นภายในระยะเวลาอันสั้น ดังนั้นเพื่อประมวลผลสารสนเทศในเวลาที่สามารถยอมรับได้การประมวลผลข้อมูลขนาดใหญ่ (Big Data) เหล่านี้จึงต้องใช้ทรัพยากรคอมพิวเตอร์ในปริมาณที่มากตามไปด้วย

การประมวลผลประสิทธิภาพสูง (High Performance Computing) ถูกพัฒนาขึ้นเพื่อตอบสนองความต้องการทรัพยากรของคอมพิวเตอร์เพื่อใช้ประมวลผลในระดับสูงมาก ระบบประมวลผลประสิทธิภาพสูงประเภทซูเปอร์คอมพิวเตอร์ (Super Computer) จำนวนมากสร้างขึ้นโดยใช้ฮาร์ดแวร์และซอฟต์แวร์ที่ออกแบบมาเฉพาะและเป็นทรัพย์สินเฉพาะของผู้ผลิตแต่ละราย (โทมัส สเตอริ่ง และ กษิดิศ ชาญเชี่ยว, 2012; Anthony, 2012; Sterling, 2002) ถึงแม้จะสามารถพัฒนาฮาร์ดแวร์และซอฟต์แวร์ได้สะดวกเนื่องจากเป็นระบบปิดที่ผู้ผลิตสามารถกำหนดทิศทางของทั้งฮาร์ดแวร์และซอฟต์แวร์ได้เอง จึงสามารถปรับแต่งประสิทธิภาพของระบบคำนวณให้เหมาะกับงานที่จะนำไปใช้ได้ดี แต่ก็ทำให้ไม่เกิดการกระจายความรู้และการพัฒนาต่อยอดเนื่องจากติดปัญหาด้านทรัพย์สินทางปัญญาของบริษัทผู้ผลิตที่เก็บเทคโนโลยีไว้เป็นความลับภายในบริษัท การที่ซูเปอร์คอมพิวเตอร์ถูกออกแบบให้ทำงานเน้นเฉพาะด้านจึงมักจะถูกกำหนดขนาดของระบบ เช่น

จำนวนโหนด (Node) ในระบบถูกกำหนดไว้ในจำนวนที่เหมาะสมกับงานและแต่ละโหนดถูกสร้างขึ้นมาด้วยฮาร์ดแวร์ปรับแต่งเฉพาะ รวมถึงรูปแบบการประมวลผลบางรูปแบบอาจทำให้ไม่สามารถขยายระบบได้สะดวก (โทมัส สเตอริ่ง และ กษิดิศ ชาญเชี่ยว, 2012) ระบบแบบกระจายนั้นนอกจากจะอาศัยฮาร์ดแวร์ที่ประกอบกันเป็นระบบที่สามารถทำงานได้แล้ว ระบบนี้ยังต้องการซอฟต์แวร์เพื่อเป็นตัวประสานงานระหว่างหน่วยภายในระบบอีกด้วย การพัฒนาซอฟต์แวร์ที่ใช้ในการประมวลผลแบบประจายศูนย์ที่ติดต่อกันโดยการส่งผ่านข้อความ (Passing Messages) มีมาตรฐานเพื่อเข้ามาช่วยใช้การส่งผ่าน คือ MPI ซึ่งเป็นคุณลักษณะมาตรฐาน (Standard Specification) เพื่อให้ไลบรารี่ (Library) ที่ปฏิบัติตามเงื่อนไขตามมาตรฐาน MPI สามารถนำไปใช้บนคอมพิวเตอร์โหนดใด ๆ บนคลัสเตอร์ (Cluster) ได้ (Quinn, 2003) และยังสามารถใช้ซ้ำบนคอมพิวเตอร์โหนดใหม่ได้แม้ทรัพยากรบนเครื่องจะแตกต่างกันโดยยังคงความสามารถเข้ากันได้ไว้

ข้อจำกัดหลายด้านของ MPI เช่น ความต้องการในการพัฒนาซอฟต์แวร์ที่ทำงานเฉพาะแต่ละความต้องการประมวลผล ถึงแม้จะมีไลบรารี่ช่วยในการพัฒนาแต่ยังต้องใช้ผู้เชี่ยวชาญในการพัฒนาซอฟต์แวร์ในบางกระบวนการ เช่น ต้องจัดการกับการขยายระบบเอง ต้องจัดการระบบที่คงทนต่อความล้มเหลว (Fault Tolerance) โดยผู้ใช้เอง เป็นต้น เนื่องจากข้อจำกัดดังกล่าวจึงต้องพัฒนาซอฟต์แวร์ขึ้นใหม่สำหรับแต่ละงานที่จะนำไปประมวลผล ด้วยเหตุนี้จึงมีการนำเสนอตัวแบบการโปรแกรม (Programming Model) ซอฟต์แวร์แบบใหม่เรียกว่า MapReduce (Dean and Ghemawat, 2008) ซึ่งพิจารณาซอฟต์แวร์ให้เป็นการทำงาน 2 ขั้น คือ Map และ Reduce การทำงานฟังก์ชัน (Function) Map จะมองข้อมูลเป็นลักษณะ Key และ Value คู่กันเป็นชุด และฟังก์ชัน Reduce ทำหน้าที่รวมรวบ Intermediate Key ที่เหมือนกันเข้ามาเป็นชุด ของ Value ที่มี Key ตรงกัน และมีกลไกในการคงทนต่อความล้มเหลว (Dean and Ghemawat, 2008) ผู้ใช้งานสามารถรับผิดชอบเฉพาะในฟังก์ชัน Map และ Reduce ซึ่งเป็นการทำงานเฉพาะส่วนตรรกะทางธุรกิจ (Business Logic) ที่เป็นส่วนสำคัญของการประมวลผล ความซับซ้อนของการประมวลผลแบบกระจายบางอย่างจะถูกกรอบงาน (Framework) ช่วยจัดการให้

เพื่อให้ซอฟต์แวร์ที่พัฒนามีคุณภาพตามหลักวิศวกรรมซอฟต์แวร์ (Software Engineering) ซอฟต์แวร์นั้น ๆ ต้องได้รับการทดสอบซอฟต์แวร์ (Software Testing) แต่เนื่องจากงานในระบบประมวลผลแบบกระจายนั้นมักจะเป็นงานที่มีภาระงานใหญ่มาก กล่าวคือข้อมูลที่ถูกประมวลผลอาจจะมีขนาดหลายกิกะไบต์ หลายเพตะไบต์ หรือขนาดใหญ่กว่านั้น การประมวลผลดังกล่าวอาจใช้เวลานานระดับหลายชั่วโมงหรือหลายเดือนถึงแม้จะใช้ระบบประมวลผลประสิทธิภาพสูงแล้วก็ตาม ทำให้การควบคุมคุณภาพของซอฟต์แวร์เป็นไปด้วยความยากลำบาก นักพัฒนาบางรายจึงเลือกที่จะทดสอบโดยสุ่มข้อมูลชุดขนาดเล็กขึ้นมาทดสอบ ซึ่งการเลือกทดสอบด้วยวิธีนี้อาจทำให้เกิด

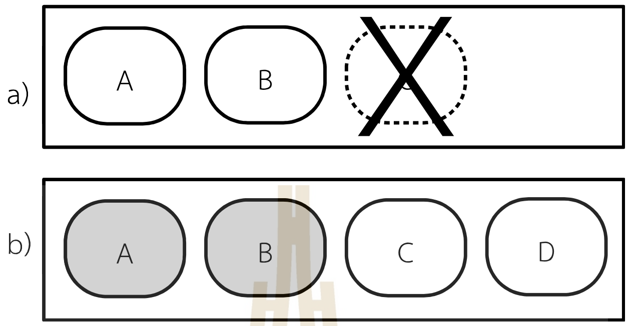
ข้อผิดพลาดเนื่องจากข้อมูลที่ก่อให้เกิดปัญหาบางค่าไม่อยู่ในชุดทดสอบ (Test Set) ทำให้ซอฟต์แวร์แจ้งกรณีทดสอบ (Test Case) ผ่าน ทั้งที่การทดสอบนั้นไม่ครอบคลุมทุกกรณี หรือนักพัฒนาบางรายเลือกใช้การประมวลผลแบบลำดับ (Sequential Processing) เพื่อง่ายต่อการตรวจแก้จุดบกพร่อง (Debug) แต่ก็จะพบว่าระบบทำงานได้ช้า และไม่สะท้อนพฤติกรรมของระบบจริงที่ประมวลผลแบบกระจายซึ่งมีสภาวะการทำงานพร้อม ๆ กัน ในกรณีที่เลวร้ายกว่านั้นคือนักพัฒนาบางรายเลือกที่จะไม่ทดสอบการทำงานของซอฟต์แวร์เลย

จุดตรวจสอบ (Checkpoint) เป็นเทคนิคที่ถูกใช้บันทึกสถานการณ์ทำงานของโปรเซส (Process) เพื่อให้ระบบคงทนต่อความล้มเหลวและสนับสนุนการบวนการย้ายงาน (Job Migration) โดยสามารถกู้คืน (Restore) สถานะได้จากข้อมูลของสถานะที่ถูกบันทึกไว้ในขณะที่ระบบสามารถประมวลผลได้ปกติ (Jangjaimon and Tzeng, 2013; Saridakis, 2003)

ปัจจุบันมีซอฟต์แวร์ที่ใช้ประมวลผลแบบกระจายมีให้เลือกใช้งานทั้งประเภทมีค่าใช้จ่ายประเภทไม่มีค่าใช้จ่ายและประเภทเปิดเผยต้นรหัส (Open Source) เพื่อประหยัดค่าใช้จ่ายและสามารถปรับปรุงการทำงานของซอฟต์แวร์ได้หากเลือกใช้ซอฟต์แวร์ประเภทเปิดเผยต้นรหัส ซอฟต์แวร์ประเภทเปิดเผยต้นรหัสที่เป็นที่นิยมมากตัวหนึ่ง คือ Spark ซึ่งเป็นซอฟต์แวร์ที่นำเอาแนวคิดการโปรแกรมแบบ MapReduce มาใช้ โดยทำการประมวลผลในหน่วยความจำ (In-Memory Computing) จึงทำงานได้อย่างรวดเร็ว (Karau et al., 2015) ซอฟต์แวร์ Spark แม้จะมีการทดสอบตัวซอฟต์แวร์เอง แต่ซอฟต์แวร์ก็ยังไม่มีกลไกรับรองว่าตัวงาน (Task) ที่พัฒนาจากซอฟต์แวร์ตัวนี้จะทำงานได้ถูกต้องตามความต้องการของผู้ใช้

วิทยานิพนธ์นี้เสนอแนวทางการประยุกต์ใช้งานจุดตรวจสอบ เพื่อใช้ควบคุมคุณภาพของงานในระบบประมวลผลแบบกระจายโดยเลือกใช้ Spark เป็นกรอบงาน โดยจะทำการสร้างจุดตรวจสอบเป็นระยะตามกรณีทดสอบ เมื่อระบบทำงานผ่านกรณีทดสอบนั้นไปแล้วจะถือว่าข้อมูลนั้นทำงานได้ถูกต้อง ระบบจะบันทึกสถานะเก็บไว้ในแหล่งเก็บข้อมูลที่น่าเชื่อถือ จากนั้นหากมีการทดสอบรอบใหม่แต่ไม่ได้เปลี่ยนแปลงกรณีทดสอบเดิมจึงไม่จำเป็นต้องเริ่มประมวลผลใหม่ตั้งแต่ต้น ระบบจะกู้คืนตำแหน่งการทำงานจากแหล่งเก็บให้ในกรณีที่กรณีทดสอบใหม่ไม่กระทบกับจุดตรวจสอบนั้น ๆ ทำให้สามารถทดสอบต่อจากกรณีทดสอบเดิมได้ โดยที่ไม่ต้องเริ่มประมวลผลใหม่ตั้งแต่ต้นดังแสดงในรูปที่ 1.1 วิธีนี้จะช่วยลดเวลาที่นักพัฒนาจะใช้พัฒนาซอฟต์แวร์หรืองานในระบบประมวลผลแบบกระจายและยังคงลักษณะเฉพาะ (Characteristic) ที่สะท้อนความเป็นการทำงานของระบบประมวลผลแบบกระจายไว้ได้ รูปที่ 1.1 a) เมื่องานกำลังถูกทำและระบบพบว่าเกิดกรณีผิดพลาดที่จุดตรวจสอบ C กรณีที่ผ่านการทดสอบคือ A และ B, รูปที่ 1.1 b) แสดงภาพการ

ทำงานเมื่อมีการแก้ไขซอฟต์แวร์ที่เป็นตัวงานแล้วสั่งทำงานอีกครั้งระบบเรียกคืนตำแหน่งล่าสุดที่ทำงานได้คือ B แล้วดำเนินการต่อจากเดิมในตำแหน่ง C และ D เพียง 2 จุด



รูปที่ 1.1 แผนภาพแสดงการประยุกต์ใช้ตำแหน่งของจุดตรวจสอบ

## 1.2 วัตถุประสงค์ของการวิจัย

วิทยานิพนธ์นี้เสนองานวิจัยที่มีวัตถุประสงค์เพื่อนำเสนอแนวทางประยุกต์ใช้จุดตรวจสอบที่ปกติใช้ในระบบคงทนต่อความล้มเหลวและการย้ายงาน มาใช้ในกระบวนการทดสอบซอฟต์แวร์ที่ทำงานบนระบบประมวลผลแบบกระจาย โดยวัตถุประสงค์มีดังนี้

1) เพื่อนำระบบจุดตรวจสอบมาประยุกต์ใช้ให้ประหยัดเวลาในการทดสอบซอฟต์แวร์ในระบบประมวลผลแบบกระจาย
2) เพื่อศึกษาผลกระทบของการนำระบบจุดตรวจสอบมาใช้ร่วมกับการทดสอบซอฟต์แวร์ในระบบประมวลผลแบบกระจาย
3) เพื่อศึกษาแนวทางการกำหนดจุดตรวจสอบที่จำเป็นต้องประมวลผลใหม่ เมื่อมีคำสั่งให้ทดสอบซอฟต์แวร์งานใหม่อีกครั้งได้อย่างเหมาะสม

## 1.3 ขอบเขตของงานวิจัย

งานวิจัยที่เสนอในวิทยานิพนธ์นี้จะเป็นการศึกษาข้อมูลและพัฒนาต้นแบบการประยุกต์ใช้จุดตรวจสอบเพื่อบันทึกสถานะและทำการกู้คืนสถานะที่สามารถใช้ซ้ำได้ในการทดสอบซอฟต์แวร์ครั้งใหม่เพื่อประหยัดเวลาในการทดสอบซอฟต์แวร์ ซึ่งเป็นแนวทางในการควบคุมคุณภาพของซอฟต์แวร์ในระบบประมวลผลแบบกระจาย

1) พัฒนาต้นแบบระบบสร้างจุดตรวจสอบเพื่อใช้ในซอฟต์แวร์ Spark รุ่น 1.6 เป็นต้นไปได้
2) สร้างกลไกเพื่อหาจุดตรวจสอบที่เหมาะสมในการกู้คืนการทดสอบโดยไม่ต้องประมวลผลใหม่ตั้งแต่ต้นในซอฟต์แวร์ Spark รุ่น 1.6 เป็นต้นไปได้

## 1.4 ประโยชน์ที่คาดว่าจะได้รับ

วิทยานิพนธ์นี้ได้รับแรงบัลดาลใจมาจากงานทางวิทยาศาสตร์และวิศวกรรมที่จำเป็นต้องใช้ความแม่นยำและเที่ยงตรงในการวัดค่าและการคำนวณ แต่กระบวนการทดสอบซอฟต์แวร์ในระบบประมวลผลแบบกระจายนั้นต้องใช้เวลาสูงมากในการทดสอบแต่ละครั้ง ทำให้มีความจำเป็นที่จะต้องมีแนวทางลดเวลาเพื่อเพิ่มผลิตภาพ (Productivity) ของการพัฒนาซอฟต์แวร์ในระบบประมวลผลแบบกระจายโดยการประยุกต์ใช้จุดตรวจสอบ ซึ่งคาดว่าจะเกิดประโยชน์ดังนี้

1) แสดงให้เห็นว่ากลไกการใช้จุดตรวจสอบจะสามารถลดเวลาที่ใช้ในการทดสอบซอฟต์แวร์งานซึ่งทำงานแบบกระจายได้
2) ได้ผลลัพธ์เปรียบเทียบของการทำ Serialization ของแต่ละกลไกที่นำมาทดสอบเปรียบเทียบ
3) เพื่อทราบผลกระทบต่อระบบเมื่อทำการใช้งานจุดตรวจสอบกับชุดข้อมูลที่มีขนาดต่างกัน เช่น ผลกระทบด้านการใช้พื้นที่เก็บข้อมูล เป็นต้น
4) สร้างระบบต้นแบบของการประยุกต์ใช้จุดตรวจสอบเพื่อเป็นพื้นฐานและใช้อ้างอิงสำหรับระบบอื่นที่จะถูกพัฒนาต่อไปในอนาคต
5) พัฒนาเทคนิคการปรับปรุงเพื่อเพิ่มประสิทธิภาพในการสร้างและทำงานของจุดตรวจสอบ

# บทที่ 2
# ปริทัศน์วรรณกรรมและงานวิจัยที่เกี่ยวข้อง

ในบทนี้จะกล่าวถึงงานวรรณกรรมที่เกี่ยวข้องกับในเรื่องระบบแบบกระจาย ระบบประมวลผลแบบกระจาย การประมวลผลแบบขนาน (Parallel Computing) การทดสอบซอฟต์แวร์ และอธิบายรายละเอียดของซอฟต์แวร์ Spark รวมไปถึงการทำจุดตรวจสอบ แนวคิดการหาค่าแตกต่างของข้อมูลโดยใช้ฟังก์ชันเข้ารหัสทางเดียว (Hash) และงานวิจัยอื่นที่เกี่ยวข้องซึ่งเคยมีการนำเสนอไว้แล้วจะได้รับการวิพากษ์ในแง่มุมต่าง ๆ ในบทนี้

## 2.1 จุดตรวจสอบ

ในการประมวลผลแบบกระจายนั้นงานจะถูกแบ่งเป็นส่วนย่อยแล้วกระจายไปยังโหนดของระบบคลัสเตอร์เพื่อประมวลผล ซึ่งจะพบว่าโอกาสที่ระบบจะล้มเหลวทั้งระบบนั้นถึงมีน้อย แต่หากโหนดใด ๆ ในระบบมีการประมวลผลงานที่ใช้เวลามากผิดปกติหรือไม่สามารถประมวลผลได้ก็จะทำให้งานอื่นที่แม้จะใช้เวลาประมวลผลสั้นแต่ก็เป็นส่วนหนึ่งของซอฟต์แวร์งานทั้งหมดนั้นถือว่าดำเนินการไม่เสร็จไปด้วยและอาจจะต้องเริ่มประมวลผลใหม่ทั้งหมด โดยเฉพาะงานที่ต้องใช้เวลาประมวลผลนาน (Long-Running) อาจจะใช้เวลาเพิ่มอีกมากจนข้อมูลที่ประมวลผลนั้นลดคุณค่าไปแล้ว เช่น งานทางอุตุนิยมวิทยาที่ต้องคำนวณสภาพอากาศหากเกิดข้อผิดพลาดจนต้องคำนวณใหม่ข้อมูลอาจจะลดคุณค่าแล้วเพราะไม่สามารถให้ผลลัพธ์ที่เป็นประโยชน์ภายในระยะเวลาที่จะเกิดประโยชน์สูงสุด หรือกระบวนการย้ายงานจากระบบหนึ่งสู่อีกระบบ เช่น การย้ายงานจากระบบเช่าที่มีราคาสูงกว่าไประบบเช่าที่มีราคาต่ำกว่า ซึ่งต้องการเคลื่อนย้ายโดยไม่รอให้ประมวลผลเสร็จก่อน ดังนั้นจึงจำเป็นต้องมีกลไกเพื่อช่วยลดปัญหาดังกล่าวเพื่อให้สามารถประมวลผลต่อไปโดยไม่ต้องเริ่มประมวลผลใหม่ทุกครั้งเมื่อมีชิ้นส่วนใด ๆ ทำงานผิดพลาด

จุดตรวจสอบถูกนำเสนอขึ้นเพื่อแก้ปัญหาข้างต้นโดยใช้การบันทึกสถานะการทำงานในขณะที่ทำงานได้เป็นปกติเก็บในหน่วยเก็บข้อมูลที่น่าเชื่อถือ เมื่อการทำงานที่ผิดพลาดเกิดขึ้นระบบจะกู้คืนจุดตรวจสอบที่ทำงานได้ปกติจากหน่วยเก็บข้อมูลเพื่อให้ระบบสามารถดำเนินการต่อไปได้อย่างเป็นปกติ (Saridakis, 2003) ประเภทของจุดตรวจสอบสามารถแบ่งกลุ่มตามเทคนิคการแทรกแซงที่ใช้โดยนักพัฒนาได้ดังนี้ (Sudha and Nisha, 2015)

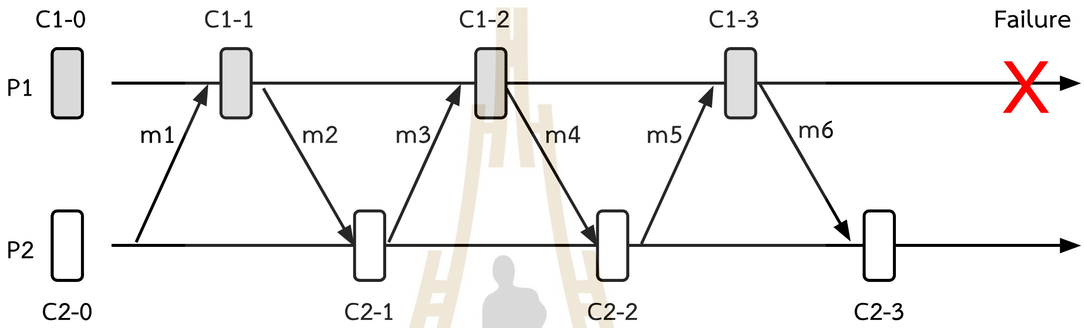
1) User-Triggered Checkpointing
   ระบบนี้นักพัฒนาจำเป็นต้องปฏิสัมพันธ์กับระบบสร้างจุดตรวจสอบด้วยตนเองซึ่งมีข้อดีคือสามารถปรับแต่งได้มากทำให้ขนาดของข้อมูลที่จะถูกเก็บนั้นมีขนาดลดลงและส่งผลให้ประสิทธิภาพดีขึ้น แต่ก็ต้องการนักพัฒนาที่มีความรู้ในการปฏิบัติการเพื่อตรวจวัดข้อมูลที่จะถูกสร้างเป็นจุดตรวจสอบและตำแหน่งของสายการกู้คืน (Recovery Line) ภายในต้นรหัสที่จะถูกส่งไปประมวลผล (Deconinck and Lauwereins, 1997)
2) Transparent Checkpointing
   เป็นเทคนิคการสร้างจุดตรวจสอบที่ไม่ต้องการปฏิสัมพันธ์กับนักพัฒนา ระบบนี้จะแยกออกจากชิ้นส่วนการทำงานของโปรแกรมหลัก ทำให้ระบบไม่ต้องการปรับแต่งไบนารี่หรือระบบปฏิบัติการในการสร้างจุดตรวจสอบสามารถจัดกลุ่มให้อยู่ในลักษณะดังนี้ได้คือ Un-coordinated Checkpointing, Coordinated Checkpointing, Quasi-Synchronous (หรือ Communication Induced Checkpointing) และ Message Logging Base Checkpoint การสร้างจุดตรวจสอบแบบนี้แม้พบว่าสามารถทำได้ง่ายในระดับแกน (Kernel) ของระบบปฏิบัติการแต่ก็พบว่าระดับโปรแกรมประยุกต์ (Application) นั้นทำได้ยาก (Plank et al., 1994)

### 2.1.1 Uncoordinated หรือ Independent Checkpointing

เทคนิคการสร้างจุดตรวจสอบแบบนี้เป็นการสร้างจุดตรวจสอบที่ไม่มีการประสานงานกันในขั้นตอนการสร้างจุดตรวจสอบระหว่างโปรเซสแต่ละตัว ตัวโปรเซสจะสนใจทำบันทึกข้อมูลสถานะของตนเองโดยอิสระซึ่งลักษณะนี้เป็นการให้อำนาจตัวโปรเซสเต็มที่ในการกำหนดว่าเมื่อใดจึงจะทำการสร้างจุดตรวจสอบของตัวมันเอง เช่น โปรเซสจะสร้างจุดตรวจสอบเมื่อสะดวกที่จะสร้างหรือเมื่อจุดนั้นจะก่อให้เกิดประโยชน์สูงสุด เป็นต้น เทคนิคนี้ช่วยขจัดปัญหาโอเวอร์เฮดของการสร้างจุดตรวจสอบแบบมีสถานะโกลบอลที่มั่นคง (Consistent Global State) เพื่อที่จะกู้คืนในขั้นตอนการกู้คืนเมื่อระบบล้มเหลว (Rusu, Grecu and Anghel, 2008)

หากข้อผิดพลาดกับระบบเกิดขึ้นระบบจะเริ่มต้นจากสถานะโกลบอลที่มั่นคงซึ่งจะสามารถติดตามตัวจุดตรวจสอบย่อยในแต่ละโปรเซสที่ขึ้นต่อกัน ขั้นตอนนี้ดำเนินการได้ช้าและยังมีข้อเสียอีก 2 ประการในการสร้างจุดตรวจสอบแบบนี้ คือ (1) ระบบอาจจะเกิดการย้อนกลับจุดตรวจสอบจนถึงตำแหน่งเริ่มต้นเนื่องจาก Domino Effect (2) ระบบต้องการสร้างจุดตรวจสอบหลายครั้งสำหรับแต่ละโปรเซสและต้องการขั้นตอนวิธี Garbage Collection เพื่อเรียกทรัพยากรคืนจากจุดตรวจสอบที่ไม่ถูกใช้เป็นเวลานาน ข้อเสียหลักของเทคนิคนี้คืออาจจะก่อให้เกิด Domino Effect ซึ่ง

แสดงดังรูปที่ 2.1 โปรเซส P1 และ P2 มีอิสระที่จะสร้างจุดตรวจสอบโดยไม่ขึ้นต่อกัน กระบวนการภายในทั้งการส่งข้อความและจุดตรวจสอบของ P1 และ P2 นั้นไม่มีความมั่นคงของจุดตรวจสอบ ยกเว้นตำแหน่ง C1-0 และ C2-0 ซึ่งเป็นจุดเริ่มต้นกระบวนการ ดังนั้นหาก P1 เกิดข้อผิดพลาดขึ้นทั้ง P1 และ P2 จะถูกย้อนกลับไปยังตำแหน่งเริ่มต้น เนื่องจากสถานะ {C1-1,C2-1} ไม่มั่นคงจากขาดข้อความ m2 และ {C1-2,C2-2} ไม่มั่นคงเนื่องจากขาดข้อความ m4 (Sudha and Nisha, 2015)



รูปที่ 2.1 ลักษณะการเกิดขึ้นของ Domino Effect (Sudha and Nisha, 2015)

### 2.1.2 Coordinated หรือ Synchronous Checkpointing

การสร้างจุดตรวจสอบแบบนี้จะได้ผลลัพธ์เป็นสถานะโกลบอลที่มั่นคงซึ่งกระบวนการส่วนใหญ่จะทำตามวิธีแบบ Two-Phase Commit (Sudha and Nisha, 2015) สำหรับในระยะแรกของกระบวนการจะเป็นการสร้างจุดตรวจสอบชั่วคราว จากนั้นในระยะที่สองคือการทำให้จุดตรวจสอบนั้นถาวร ข้อดีคือมีแค่เพียงจุดตรวจสอบถาวรเพียงตัวเดียวและจุดตรวจสอบชั่วคราวอีกจุดหนึ่งเท่านั้นที่จะถูกเก็บ กรณีที่ข้อผิดพลาดเกิดขึ้นจะกู้คืนจุดตรวจสอบถาวรตำแหน่งล่าสุด ลักษณะของการสร้างจุดตรวจสอบแบบโกลบอล (Global) มี 2 ระยะ

ระยะแรก : ตัวประสานงานจะสร้างจุดตรวจสอบและกระจายคำร้องขอสร้างจุดตรวจสอบไปยังโปรเซสแต่ละตัวเพื่อขอสร้างจุดตรวจสอบ เมื่อโปรเซสได้รับแล้วก็จะหยุดการทำงานและระงับการส่งข้อมูลในช่องทางการสื่อสารจากนั้นจะสร้างจุดตรวจสอบชั่วคราว แล้วจึงส่งข้อความตอบกลับไปให้ตัวประสานงาน

ระยะที่สอง : หลังจากที่ตัวประสานงานได้รับข้อความตอบกลับจากทุกโปรเซสก็จะส่งข้อความคอมมิท (Commit) ไปบอกทุกโปรเซสเพื่อให้ดำเนินการตามโพรโทคอล Two-Phase Commit ส่วนโปรเซสที่ได้รับข้อความคอมมิทจะทำการเปลี่ยนจุดตรวจสอบชั่วคราวให้เป็นจุด

ตรวจสอบถาวรและถ้ามีจุดตรวจสอบถาวรอยู่แล้วจุดนั้นจะถูกยกเลิก จากนั้นโปรเซสจะเรียกการประมวลผลเดิมมาทำงานและส่งข้อความเพื่อติดต่อกับโปรเซสอื่น

ลักษณะของ Coordinated Checkpointing มี 2 ลักษณะคือ (Cao and Singhal, 1998; Koo and Toueg, 1987; Kumar and Khunteta, 2010)

1) Blocking : วิธีนี้จะใช้การขัดขวางการสื่อสารของโปรเซสระหว่างที่อยู่ในกระบวนการสร้างจุดตรวจสอบกำลังทำงานอยู่เพื่อป้องกันการรับข้อความซึ่งจะทำให้จุดตรวจสอบไม่มีความมั่นคง
2) Non-Blocking : จะไม่มีการขัดขวางการทำงานของโปรเซสในระหว่างการสร้างจุดตรวจสอบประเภทนี้ ลักษณะการสร้างแบบนี้มีโพรโทคอลคือตัวตั้งต้นจะสร้างจุดตรวจสอบและส่ง Marker ซึ่งเป็นคำร้องขอสร้างจุดตรวจสอบไปยังทุกโปรเซส โปรเซสที่ได้รับแล้วก็จะทำการสร้างจุดตรวจสอบตั้งแต่ครั้งแรกที่ได้รับ Marker แล้วก็จะส่งข้อมูลไปบอกโปรเซสทุกตัวอีกครั้งก่อนที่จะส่งข้อความที่เป็นของโปรแกรมประยุกต์ โพรโทคอลนี้เชื่อว่าผู้ใช้งานกำลังใช้งานช่องทางแบบ FIFO (ข้อมูลที่เข้าก่อนจะออกก่อน)

### 2.1.3 Quasi-Synchronous หรือ Communication Induced Checkpointing

โพรโทคอลนี้ได้นำเสนอจุดตรวจสอบใน 2 รูปแบบคือแบบโลคอล (Local) เป็นจุดตรวจสอบที่สามารถสร้างได้อิสระ และแบบบังคับซึ่งใช้ในการรับรองว่าท้ายที่สุดแล้วกระบวนการในสายการกู้คืนจะทำงานได้อย่างถูกต้องและลดจุดตรวจสอบที่ไม่เกิดประโยชน์ เทคนิคนี้ช่วยหลีกเลี่ยง Domino Effect ในขณะที่โปรเซสบางตัวยังคงสามารถสร้างจุดตรวจสอบได้โดยอิสระและไม่ต้องประสานงานระหว่างโปรเซส เทคนิคนี้จะนำข้อมูลโพรโทคอลที่เกี่ยวข้องอาศัยไปกับข้อความของโปรแกรมประยุกต์ ผู้รับแต่ละข้อความจะใช้ข้อมูลที่มาด้วยนี้สร้างจุดตรวจสอบแบบบังคับไว้ล่วงหน้าสำหรับกระบวนการสายการกู้คืนโกลบอล การสร้างจุดตรวจสอบต้องสร้างก่อนที่โปรแกรมประยุกต์จะประมวลผลเนื้อหาของข้อความ (Alvisi et al.,1999; Baldoni et al., 1997 Saridakis, 2003)

### 2.1.4 Message Logging

โพรโทคอลนี้ต้องการให้แต่ละโปรเซสบันทึกสถานะโลคอล ของตัวเองเป็นรอบ ๆ หลังจากที่สถานะถูกบันทึกในแหล่งเก็บข้อมูลแล้วจะได้ล็อกข้อความขึ้นมา เมื่อโปรเซสดำเนินการล้มเหลวโปรเซสใหม่จะถูกสร้างขึ้นทดแทนและโปรเซสใหม่นี้ก็จะถูกบันทึกสถานะ จากนั้นล็อกข้อความจะถูกอ่านและทำซ้ำการประมวลผลบนโปรเซสใหม่นั้น ทุกโพรโทคอลของ Message Logging ต้องการกระบวนการกู้คืนโปรเซสที่ล้มเหลวอย่างน้อยหนึ่งโปรเซสเพราะความ

มั่นคงของสถานะขึ้นอยู่กับสถานะของโปรเซสอื่น ความสัมพันธ์ที่ก่อให้เกิดความมั่นคงนี้ถูกแสดงในรูปโปรเซสกำพร้า (Orphan Process) ซึ่งเป็นโปรเซสที่ยังทำงานได้อยู่แต่สถานะไม่มั่นคงจึงต้องมีโพรโทคอล Message Logging รับรองว่าเมื่อมีการกู้ข้อมูลจะไม่เกิดโปรเซสกำพร้า สามารถหลีกเลี่ยงโดยการบังคับใช้โพรโทคอล Pessimistic เพื่อหลีกเลี่ยงการสร้างโปรเซสกำพร้าในขณะประมวลผล หรือบังคับใช้โพรโทคอล Optimistic เพื่อกำหนดการดำเนินการอย่างเหมาะสมระหว่างกระบวนการกู้คืนจุดตรวจสอบ (Alvisi and Marzullo, 1995; Deshpande and Kamalapur, 2008; Elnozahy and Zwaenepoel, 1994; Meneses, Mendes and Kalé, 2010)

### 2.1.5 ตัวอย่างขั้นตอนวิธีของกระบวนการสร้างจุดตรวจสอบ

จุดตรวจสอบถูกนำไปประยุกต์ใช้ในหลากหลายระบบซึ่งแต่ละระบบก็จะมีขั้นตอนวิธีที่เหมาะสมกับแต่ละระบบที่นำไปใช้งานต่างกัน ในหัวข้อนี้จะได้อภิปรายขั้นตอนวิธีที่น่าสนใจ

1) Chandy and Lamport (CL)

วัตถุประสงค์ของขั้นตอนวิธีนี้ก็เพื่อสร้าง Snapshot แบบโกลบอลซึ่งก็คือกลุ่มของจุดตรวจสอบระดับโลคอลในระบบประมวลผลแบบกระจาย โดยใช้ขั้นตอนวิธีที่เรียกว่า Snapshot คือการบันทึกสถานะโกลบอลที่มั่นคงของระบบไม่ประสานเวลา (Asynchronous) กลไกการทำงานจะเป็นแบบ Non-Blocking Coordinated Checkpointing ระบบจะใช้การส่งข้อความ Marker ไปกับช่องทางการสื่อสารในระหว่างการสร้างจุดตรวจสอบ ทำให้เกิดความซับซ้อนของการส่งผ่านข้อมูลมีค่าประสิทธิภาพเป็น O($n^2$) การสร้างจุดตรวจสอบและช่องทางการส่งผ่านข้อมูลต้องเป็นแบบ FIFO ระบบส่งผ่านข้อความมักจะใช้ขั้นตอนวิธีนี้เป็นพื้นฐาน (Chandy and Lamport, 1985; Kshemkalyani, Raynal and Singhal, 1995; Sudha and Nisha, 2015) ระบบมีสมมุติฐานว่า

- ระบบมีโปรเซสจำกัดและช่องทางสื่อสารมีจำกัด
- โปรเซสติดต่อกับโปรเซสอื่นโดยการส่งผ่านข้อความผ่านช่องทางการสื่อสาร
- ช่องทางการสื่อสารจะไม่เกิดปัญหาขึ้น
- สถานะโกลบอลของระบบประกอบไปด้วยสถานะโลคอลของตัวโปรเซสและสถานะของช่องทางสื่อการ
- สถานะของช่องทางการสื่อสารอ้างถึงเซ็ตของข้อความที่ส่งผ่านช่องทางการสื่อสาร
- บัฟเฟอร์ (Buffer) ขนาดไม่จำกัด

- การสิ้นสุดกระบวนการจะรับประกันด้วยช่องทางการสื่อสารที่ไม่มีข้อผิดพลาด

ขั้นตอนการสร้างจุดตรวจสอบมีดังนี้

ขั้นตอนแรกตัวตั้งต้นจะบันทึกสถานะของตนเองและส่ง Maker ออกไปทุกช่องทางการสื่อสารที่เป็นช่องทางส่งออก

ขั้นตอนที่สองสำหรับโปรเซสอื่นทุกโปรเซส หากได้รับ Marker มาในครั้งแรกไม่ว่าจะได้รับจากช่องทางการสื่อสารใดที่เป็นช่องทางนำเข้าจะทำการบันทึกสถานะและส่งต่อ Marker ไปยังโปรเซสอื่นตามช่องทางการสื่อสารขาออกแล้วจึงจะทำการประมวลผลงานหลักต่อไป แต่ระหว่างนี้ก็จะบันทึกข้อความที่ถูกส่งเข้ามาผ่านช่องทางนำเข้าจนกระทั่งได้รับ Marker ผ่านทางช่องทางการสื่อสาร

2) Lai-Yang Coloring Scheme

เป็นลักษณะ Snapshot แบบโกลบอลสำหรับระบบที่คิวการทำงานไม่เป็น FIFO ทำให้ระบบของ Lai-Yang เติมเต็มบทบาทของระบบที่ไม่เป็น FIFO โดยใช้แผนผังสีในการประมวลผลข้อความดังนี้ (Jaggi and Singh, 2011; Kshemkalyani, Raynal and Singhal, 1995, Lai and Yang, 1987)

I) กำหนดให้ทุกโปรเซสเป็นสีขาวแล้วเปลี่ยนให้เป็นสีแดงขณะที่กำลังสร้าง Snapshot

II) ทุกข้อความที่ส่งออกจากโปรเซสสีขาวจะกำหนดให้เป็นสีขาวและข้อความนั้นจะถูกส่งก่อนที่โปรเซสผู้ส่งข้อความจะบันทึกสถานะของตนให้เป็น Snapshot แบบโลคอลในกรณีของโปรเซสสีแดงข้อความที่ส่งออกก็จะเป็นสีแดงแต่ข้อความจะส่งหลังมีการบันทึกสถานะลง Snapshot แบบโลคอลแล้ว

III) ทุกโปรเซสสีขาวจะสามารถสร้าง Snapshot ในเวลาใดก็ได้ตามความสะดวกแต่ต้องไม่ช้ากว่าที่จะได้รับข้อความสีแดง

ดังนั้นแล้วหากโปรเซสสีขาวได้รับข้อความสีแดงก่อนที่จะมีการประมวลผลข้อความ โปรเซสจะบันทึกข้อความลงใน Snapshot แบบโลคอลของตัวเองวิธีนี้ช่วยรับรองว่าจะไม่มีข้อความที่ถูกส่งจากโปรเซสหลังจากที่บันทึกลง Snapshot แบบโลคอลแล้ว ซึ่ง Snapshot แบบโลคอลนี้จะถูกประมวลผลที่โปรเซสปลายทางก่อนที่โปรเซสปลายทางจะบันทึก Snapshot แบบโลคอลจึงชัดเจนว่า Marker ไม่จำเป็นต่อวิธีนี้แต่ Marker ใช้สำหรับสร้างแผนผังสี

3) Minimun process Blocking Scheme ของ Koo-Toueg

ขั้นตอนวิธีแบบนี้พยายามใช้การ Blocking เพื่อสร้างจุดตรวจสอบให้น้อยที่สุดตามสมมุติฐานดังนี้

- โปรเซสทั้งหมดติดต่อกันผ่านทางการแลกเปลี่ยนข้อความโดยใช้ช่องทางการสื่อสารที่เป็น FIFO
- ช่องทางการสื่อสารจะไม่เกิดปัญหาขึ้น

ขั้นตอนการสร้างจะมี 2 ระยะ ซึ่งระยะแรกคือตัวตั้งต้นจุดตรวจสอบจะบ่งชี้โปรเซสทั้งหมดที่มีการติดต่อสื่อสารด้วยหลังจากสร้างจุดตรวจสอบล่าสุดจากนั้นจึงส่งคำขอสร้างจุดตรวจสอบไปหาโปรเซสเหล่านั้น เมื่อได้รับคำขอสร้างจุดตรวจสอบแล้วแต่ละโปรเซสจะบ่งชี้ไปถึงโปรเซสที่มันมีการติดต่อสื่อสารด้วยหลังจากสร้างจุดตรวจสอบล่าสุดจากนั้นจึงส่งคำขอสร้างจุดตรวจสอบไปหาโปรเซสเหล่านั้น ทำแบบนี้ไปเรื่อย ๆ จนหมดโปรเซสที่ต้องบ่งชี้ ส่วนในระยะที่สองโปรเซสที่ถูกบ่งชี้ทั้งหมดในระยะแรกจะทำการสร้างจุดตรวจสอบ ผลลัพธ์ที่ได้จากกระบวนการนี้คือจุดตรวจสอบที่มีความมั่นคงและเกี่ยวข้องกับโปรเซสเพียงบางส่วน (Elnozahy et al., 2002; Koo and Toueg, 1987)

4) Zig-Zag Path ของ Xu และ Netzer

เป็นแนวคิดที่ใช้ลักษณะทั่วไปของความสัมพันธ์แบบเกิดขึ้นก่อน (Happened-Before) ของ Lamport ไม่เพียงพอที่จะใช้เพื่ออธิบายจุดตรวจสอบที่สามารถอยู่ใน Consistent Global Snapshot ซึ่งเป็นสถานะที่เซ็ตของจุดตรวจสอบระดับโลคอลซึ่งมีหนึ่งจุดต่อหนึ่ง โปรเซสนั้นไม่มีข้อความที่ถูกบันทึกว่าได้รับแล้วทั้งที่ยังไม่ได้ส่งประกอบอยู่ จึงมีการนำเสนอแนวคิด Zig-Zag Path (Elnozahy et al., 2002; Netzer and Xu, 1995) โดยจุดตรวจสอบที่ประกอบเป็น Consistent Global Snapshot ต้องไม่เป็น Zig-Zag Path กับจุดตรวจสอบอื่นใดที่อยู่ใน Snapshot เดียวกัน Zig-Zag Path จะเกิดขึ้นก็ต่อเมื่อจุดตรวจสอบ $C_{\mathrm{x},i}$ ไปยัง $C_{y,j}$ มีการส่งผ่านข้อความ $M_1, M_2, \ldots, M_n$ $(n \,\square\, 1)$ แล้ว

- $M_1$ ถูกส่งจากโปรเซส x หลังจากมีการสร้างจุดตรวจสอบ $C_{\mathrm{x},i}$ แล้ว
- ถ้า $M_k$ $(1 \,\square\, k < n)$ ได้รับโดยโปรเซส z แล้วโปรเซส z ส่งข้อความ $M_{k+1}$ ออกไปในช่วงที่กำลังสร้างจุดตรวจสอบหรือเมื่อสร้างจุดตรวจสอบแล้ว (แม้ว่า $M_{k+1}$ จะส่งก่อนหรือหลังจากที่โปรเซส z ได้รับ $M_k$)
- $M_n$ ได้รับโดยโปรเซส y ก่อนที่จะทำ $C_{y,j}$

### 2.1.6 ตัวอย่างการประยุกต์ใช้จุดตรวจสอบ

Libhashckpt (Ferreira et al., 2011) ดัดแปลงมาจากไลบรารี่ Libckpt มีลักษณะผสมระหว่างการใช้กลไกการป้องกันเพจ (Page Protection) และกลไกการเข้ารหัสทางเดียวส่งผ่านการประมวลผลโดย MPI การทำงานของไลบรารี่นี้คือระหว่างที่โปรแกรมประยุกต์กำลังทำงานอยู่ระบบจะใช้กลไกป้องกันป้องกันเพจ เพื่อทำเครื่องหมายว่าเพจของหน่วยความจำเสมือนใดบ้างที่อาจจะปนเปื้อน เมื่อมีการร้องขอสร้างจุดตรวจสอบเพจเหล่านี้จะถูกแบ่งเป็นบล็อกย่อยแล้วส่งไปประมวลผลหาค่ารหัสทางเดียวยัง GPU เมื่อเสร็จแล้วจึงนำค่ารหัสทางเดียวที่ได้ของบล็อกย่อยมาเปรียบเทียบกับจุดตรวจสอบก่อนหน้าหากค่ารหัสทางเดียวมีค่าต่างกันจึงจะบันทึกการเปลี่ยนแปลงของบล็อกย่อยนั้น วิธีการนี้ช่วยประหยัดพื้นที่เนื่องจากแต่เดิมต้องใช้ข้อมูลทั้งเพจซึ่งมีขนาดใหญ่กว่าขนาดของบล็อกและแม้เพจนั้นเปลี่ยนแปลงเพียงเล็กน้อยก็ต้องเก็บข้อมูลเอาไว้ทั้งหมดเพจ แต่ไลบรารี่นี้ทำให้สามารถตรวจจับบล็อกข้อมูลขนาดที่เล็กลงได้ แต่ก็ต้องการใช้การประมวลผลซึ่งจะทำให้เกิดภาระงานเพิ่มขึ้น

## 2.2 จุดตรวจสอบโดยใช้กลไกของ Spark

ซอฟต์แวร์ Spark มีความจำเป็นต้องมีจุดตรวจสอบสำหรับงานที่ต้องใช้เวลาประมวลผลเป็นเวลานาน การประมวลผลที่ต้องการความต่อเนื่องกันหลายขั้นตอน หรือกรณีที่ RDD นั้นขึ้นต่อกันกับ RDD อื่นจำนวนมาก การสร้างจุดตรวจสอบจะทำให้สายตระกูล (Lineage) ของ RDD นั้นสลายไป (Laskowski, 2016) กระบวนการแคช (Cache) เป็นการคงข้อมูลไว้ในหน่วยความจำจึงสามารถกระทำได้พร้อมงานหลักในโปรแกรมประยุกต์ การประมวลผลงานหลักของ RDD จึงถูกประมวลผลเพียงครั้งเดียว แต่การสร้างจุดตรวจสอบนั้นจะต้องรอให้การประมวลผลงานหลักเสร็จสิ้นก่อนจึงจะสามารถประมวลผลงานสร้างจุดตรวจสอบแล้วเก็บข้อมูลในแหล่งเก็บที่น่าเชื่อถือ ทำให้เกิดการประมวลผลงานซ้ำอีกครั้ง ดังนั้นกระบวนการแคชจึงถูกใช้ร่วมกับกระบวนการสร้างจุดตรวจสอบเพื่อที่จะไม่ต้องคำนวณซ้ำแต่จะเป็นการนำข้อมูลที่แคชไว้มาใช้สร้างจุดตรวจสอบแทน (Xu, 2015a) จากลักษณะที่เป็นแบบอ่านอย่างเดียวของ RDD ทำให้สามารถสร้างจุดตรวจสอบโดยการเขียนข้อมูลออกจากระบบด้วยโปรเซสเบื้องหลังโดยไม่ต้องหยุดการทำงานของการประมวลผลงานหลัก (Zaharia et al., 2012a) มีนักวิจัยได้นำจุดตรวจสอบของ Spark ดังกล่าวเข้ามาประยุกต์ใช้ดังจะได้นำเสนอหัวข้อที่น่าสนใจ

## 2.3 การทดสอบซอฟต์แวร์

กระบวนการทดสอบซอฟต์แวร์เป็นกระบวนการหนึ่งที่ถูกใช้ควบคุณคุณภาพของซอฟต์แวร์ตามหลักของวิศวกรรมซอฟต์แวร์ ว่าเป็นไปตามความต้องการในการพัฒนาซอฟต์แวร์หรือไม่? ซอฟต์แวร์นั้นทำงานได้อย่างถูกต้องหรือไม่? ซอฟต์แวร์นั้นพร้อมที่จะนำไปใช้งานหรือไม่? ซอฟต์แวร์นั้นมีความปลอดภัยเพียงพอหรือไม่? หรือซอฟต์แวร์นั้นสามารถทำงานในสภาพแวดล้อมของผู้จ้างได้ถูกต้องหรือไม่? คำถามเหล่านี้สามารถตอบได้ด้วยการทดสอบซอฟต์แวร์ซึ่งโดยปกติแล้วจะมีการทำข้อตกลงระหว่างผู้ว่าจ้างกับผู้พัฒนาเกี่ยวกับความต้องการด้านซอฟต์แวร์ เมื่อพัฒนาไปตามวงจรการพัฒนาซอฟต์แวร์ (Software Development Life Cycle) จนถึงเวลาหนึ่งแล้วเพื่อเป็นตัวชี้วัดว่าซอฟต์แวร์ทำงานได้ตามความต้องการของผู้ว่าจ้างและมีคุณภาพเพียงพอ ผู้พัฒนาจะทดสอบการทำงานของซอฟต์แวร์เพื่อให้ผู้ว่าจ้างยอมรับซอฟต์แวร์ที่พัฒนานั้น ซึ่งพบว่าการทดสอบซอฟต์แวร์ใช้เวลาและค่าใช้จ่ายของการพัฒนาซอฟต์แวร์ไปร้อยละ 25 ถึงร้อยละ 50 (Spillner et al., 2014) ประเภทของการทดสอบซอฟต์แวร์นั้นถูกแบ่งออกเป็นหลายประเภทและหลายเทคนิค (Pressman, 2010) แต่จะเลือกแสดงเฉพาะการทดสอบที่เกี่ยวข้องกับวิทยานิพนธ์นี้เท่านั้น

### 2.3.1 การทดสอบแบบกล่องดำ (Black-Box Testing)

การทดสอบด้วยเทคนิคนี้เป็นการทดสอบพฤติกรรมการทำงานของระบบโดยที่ผู้ทดสอบไม่จำเป็นต้องทราบกลไกการทำงานของซอฟต์แวร์ การทดสอบจะเน้นไปที่ความสามารถทำงานของซอฟต์แวร์ว่าทำงานจนได้ผลลัพธ์ถูกต้องตามที่คาดหวัง การต่อประสาน (Interface) ระหว่างฟังก์ชันหลายส่วน การเชื่อมต่อกับระบบอื่น กระบวนการเริ่มทำงานและสิ้นสุดงาน รวมถึงประสิทธิภาพการทำงานของระบบ (Pressman, 2010; Spillner et al., 2014)

### 2.3.2 การทดสอบแบบกล่องขาว (White-Box Testing)

เทคนิคนี้เป็นการทดสอบที่ผู้ทดสอบล่วงรู้กลไกการทำงานของระบบ การทำงานของหน่วยตัดสินใจ การตรวจสอบข้อมูลในตัวแปรหรือขอบเขตเงื่อนไขของการวนซ้ำซึ่งเป็นกระบวนการภายในตัวซอฟต์แวร์ดังนั้นผู้ทดสอบต้องมีความรู้ในการพัฒนาซอฟต์แวร์จึงจะสามารถใช้เทคนิคนี้ และเทคนิคนี้มักจะถูกใช้ในการทดสอบที่ทำงานใกล้ชิดกับต้นรหัส (Pressman, 2010; Spillner et al., 2014)

### 2.3.3 การทดสอบระดับหน่วย (Unit Testing)

การทดสอบระดับหน่วยมักถูกใช้ร่วมกับการพัฒนาซอฟต์แวร์เนื่องจากการทดสอบด้วยวิธีนี้จะเป็นการทดสอบหน่วยย่อยที่สุดของการออกแบบระบบ กรณีทดสอบมักจะถูกเขียนในระดับชั้นเดียวกับต้นรหัสทำให้สามารถทดสอบขั้นตอนวิธี โครงสร้างข้อมูล ขอบเขตการ

ตรวจสอบข้อมูลหรือใช้ทดสอบกรณีคำสั่งงานซึ่งจะต้องมีการปฏิบัติงานอย่างน้อยหนึ่งครั้งเพื่อป้องกันคำสั่งงานซึ่งจะไม่เคยทำงานจริงเลย (Hamill, 2004; Pressman, 2010)

### 2.3.4 การทดสอบระดับบูรณาการ (Integration Testing)

การทดสอบระดับนี้เป็นการทดสองว่าชิ้นส่วนต่าง ๆ ที่ถูกสร้างขึ้นนั้นสามารถประกอบรวมเข้าเป็นระบบเดียวกันโดยที่ไม่เกิดปัญหา ซึ่งปัญหาหลายครั้งของการพัฒนาซอฟต์แวร์จะพบก็ต่อเมื่อมีการบูรณาการเข้าเป็นระบบเดียวกัน เมื่อบูรณาการแล้วการทำงานภายในแต่ละส่วนต้องถูกต้องและการติดต่อระหว่างแต่ละส่วนก็ต้องสามารถทำงานได้ถูกต้องด้วย ข้อมูลไม่หายไประหว่างการทดสอบและเป็นไปตามความต้องการของซอฟต์แวร์ที่ได้ออกแบบไว้ ซึ่งการทดสอบระดับนี้จะสัมพันธ์กับการทดสอบระดับหน่วย กล่าวคือการทดสอบระดับบูรณาการมักจะทำหลังการทดสอบระดับหน่วยเพื่อให้มั่นใจได้ว่าระบบที่หน่วยเล็กกว่านั้นทำงานได้ถูกต้องทั้งหมด การทดสอบกับหน่วยที่ใหญ่กว่าจึงจะมีโอกาสทำงานได้ถูกต้อง (Pressman, 2010; Spillner et al., 2014)

### 2.3.5 การทดสอบแบบถดถอย (Regression Testing)

การทดสอบระดับนี้จะทำหลังการทดสอบแบบบูรณาการ เพื่อยืนยันว่าชิ้นส่วนของซอฟต์แวร์ชิ้นใหม่ที่ถูกเพิ่มเข้าไปในระบบหรือชิ้นส่วนของซอฟต์แวร์ที่มีการแก้ไขนั้นจะไม่ทำให้ระบบที่มีอยู่เดิมและสามารถทำงานได้อยู่แล้วเกิดข้อผิดพลาดขึ้น ซึ่งช่วยในการตรวจสอบการทำงานของระบบและรับรองความถูกต้องของระบบได้ในระดับหนึ่ง (Pressman, 2010; Spillner et al., 2014)

### 2.3.6 การทดสอบการกู้คืน (Recovery Testing)

การทดสอบการกู้คืนเป็นหน่วยย่อยของกระบวนการทดสอบระบบ (System Testing) การทดสอบการกู้คืนเป็นการนำกระบวนการทดสอบไปใช้กับกระบวนการคงทนต่อความล้มเหลว เพื่อทดสอบว่าจุดตรวจสอบนั้นมีการกู้คืนได้ถูกต้องตามที่ควรจะเป็นหรือไม่และเมื่อกู้คืนระบบได้แล้วระบบทำงานตามตรรกะทางธุรกิจต่อได้อย่างถูกต้องตามที่ต้องการ และทันภายในเวลาที่กำหนด (Pressman, 2010)

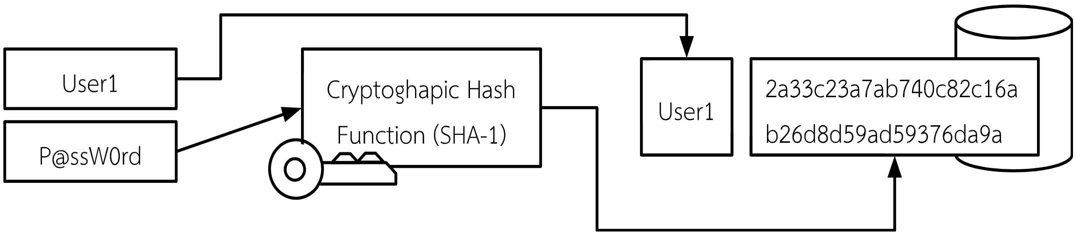
### 2.3.7 เทคนิคการพัฒนาซอฟต์แวร์แบบ Test-Driven Development (TDD)

เทคนิคการพัฒนาซอฟต์แวร์แบบนี้เป็นการนำเสนอรูปแบบที่มีกรณีทดสอบควบคุมการพัฒนาต้นรหัสที่จะถูกใช้งานจริง โดยเมื่อมีความต้องการเพิ่มความสามารถของซอฟต์แวร์จะเพิ่มกรณีทดสอบเข้าไปก่อน จากนั้นเรียกทดสอบกรณีทดสอบที่มีทั้งหมดซึ่งจะพบว่ามีแค่กรณีทดสอบที่เพิ่มเข้าไปใหม่เท่านั้นที่เกิดปัญหาเนื่องจากยังไม่ได้พัฒนาต้นรหัส เสร็จแล้วจึงพัฒนาต้นรหัสแค่ให้กรณีทดสอบที่เพิ่มเข้าไปใหม่นั้นทำงานผ่าน และลองเรียกกรณีทดสอบทั้งหมดใหม่อีกครั้งต้องสามารถดำเนินการผ่านทุกกรณีทดสอบ ทำให้สามารถที่จะแก้ไขต้นรหัส

หรือลบส่วนที่ทำงานซ้ำซ้อนโดยที่หากแก้ไขได้ถูกต้องกรณีทดสอบทั้งหมดก็ยังคงจะทำงานผ่าน เพราะกรณีทดสอบคือส่วนที่ช่วยในการควบคุมคุณภาพของซอฟต์แวร์ในกรณีที่มีการแก้ไขชิ้นส่วนของซอฟต์แวร์เพื่อรับรองว่าทุกชิ้นส่วนจะยังคงทำงานได้ถูกต้อง (Beck, 2003; Hamill, 2004)

## 2.4 การตรวจสอบความคงสภาพของข้อมูล (Data Integrity)

กระบวนการทำงานใดก็ตามบนระบบคอมพิวเตอร์มีความเสี่ยงที่จะเกิดความผิดพลาดในการรับ-ส่งข้อมูลหรือประมวลผลทำให้ข้อมูลบนระบบคอมพิวเตอร์นั้นไม่มีความคงสภาพ ข้อมูลอาจจะเปลี่ยนแปลงไประหว่างการทำงานกลไกของระบบโดยที่ผู้ใช้ระบบไม่ได้ต้องการให้เกิดขึ้น ดังนั้นจึงจำเป็นต้องมีกระบวนการเพื่อทำให้มั่นใจได้ว่าระบบยังคงสภาพของข้อมูลนั้นไว้โดยไม่เปลี่ยนแปลง วิธีการหนึ่งที่ถูกนำเสนอเพื่อแก้ปัญหานี้ก็คือ Checksum ซึ่งเป็นกลไกที่ใช้ตรวจจับการแก้ไขหรือเปลี่ยนแปลงของไฟล์ (Smoak et al., 2012) โดยมีกระบวนการทำงานคือการนำข้อมูลเข้าระบบเพื่อคำนวณ Checksum จากนั้นระบบจะคำนวณตามขั้นตอนวิธี ที่เลือกใช้ เมื่อเสร็จสิ้นการคำนวณผลลัพธ์ที่ได้ออกมาคือ Checksum ซึ่งขั้นตอนวิธีมีหลายรูปแบบตามความเหมาะสมของงาน เช่น การใช้งานเป็นฟังก์ชันทางเดียวสำหรับวิทยาการรหัสลับ (Cryptographic Hash Function) เมื่อนำข้อมูลนำเข้า (Input) ไปแล้วประมวลผลจนกระทั่งได้ Checksum ที่เป็นผลลัพธ์จากกระบวนการแล้ว Checksum ที่ได้จะคงทนต่อวิศวกรรมย้อนกลับทำให้การคำนวณย้อนกลับนั้นแทบจะเป็นไปไม่ได้เลยจึงถูกใช้ในกระบวนการปกปิดรหัสลับ ผู้ให้บริการจะนำข้อมูลของผู้ใช้บริการระบบไปเข้าฟังก์ชันดังกล่าวเพื่อคำนวณหา Checksum จากนั้นเมื่อได้ Checksum แล้วจึงเพิ่มเข้าในฐานข้อมูลแทนที่จะเพิ่มข้อมูลรหัสเข้าไปในระบบโดยตรง ทำให้แม้มีผู้ไม่ประสงค์ดีได้ค่า Checksum ที่ถูกใช้แทนรหัสลับไปก็ไม่สามารถทำกระบวนการวิศวกรรมย้อนกลับเพื่อให้ได้รหัสลับที่แท้จริงได้ เช่น ขั้นตอนวิธีแบบ SHA-1 ดังแสดงดังรูปที่ 2.2



รูปที่ 2.2 แผนภาพแสดงการนำรหัสลับเข้าฟังก์ชัน SHA-จากนั้นจึงนำไปเก็บไว้ฐานข้อมูล

### 2.4.1 Secure Hash Algorithm 1

Secure Hash Algorithm 1 เป็นฟังก์ชันเข้ารหัสทางเดียว ซึ่งสามารถใช้งานเป็น Checksum ได้ ออกแบบโดยหน่วยงาน NSA และนำถูกนำเสนอโดยหน่วยงาน NIST ของสหรัฐอเมริกาในปี ค.ศ. 1995 ฟังก์ชันเข้ารหัสทางเดียวชนิดนี้ให้ค่ารหัสออกมาเป็น 160 บิต (20 ไบต์) หรือ 40 ตัวอักษรในหน่วยเลขฐานสิบหก (Wikipedia, 2016) เปิดเป็นมาตรฐานที่ FIPS PUB 180-1 และใช้บล็อกข้อมูลขนาด 512 บิต (ธนา หงษ์สุวรรณ, n.d., pp. 5-8) ขั้นตอนการสร้างค่ารหัสทำได้ดังนี้

รูปที่ 2.3 แผนภาพแสดงตำแหน่งของ Padding Bits (ธนา หงษ์สุวรรณ, n.d., pp. 5-8)

1) เติม Padding Bits หลังข้อความ เพื่อให้ได้บล็อกที่มีขนาด 448 บิตดังรูปที่ 2.3

2) เติมค่าความยาวของข้อมูลขนาด 64 บิตท้าย Padding Bits ดังรูปที่ 2.3 ทำให้ได้บล็อกขนาดรวม 512 บิต
3) เนื่องจากฟังก์ชันทางเดียวแบบ SHA-1 ใช้บัฟเฟอร์พักข้อมูลที่ส่งออกจากขั้นตอนก่อนหน้าเพื่อป้อนเป็นส่วนหนึ่งของข้อมูลนำเข้าในขั้นตอนถัดไป ซึ่งบัฟเฟอร์สามารถแทนด้วยรีจิสเตอร์ (Register) ขนาด 32 บิตจำนวน 5 ตัว รวม 160 บิต คือ รีจิสเตอร์ A, B, C, D และ E เนื่องจากขั้นตอนแรกเริ่มนั้นยังไม่มีข้อมูลส่งออกในขั้นตอนก่อนหน้า จึงกำหนดให้รีจิสเตอร์ A=67452301, B=EFCDAB89, C=98BADCFE , D=10325476, E= C3D2E1F0
4) ในแต่ละบล็อกข้อมูลขนาด 512 บิตจะถูกแบ่งเป็นบล็อกละ 32 บิต เมื่อได้ 16 บล็อกแล้วจึงขยายเป็น 80 บล็อกตามสมการที่ 2.1 โดยจะประมวลผล 4 รอบ แต่ละรอบจะดำเนินการสมการทางคณิตศาสตร์ดังตารางที่ 2.1 จำนวน 20 ขั้นตอนย่อยรวมแล้วจะเกิดการทำงาน 80 รอบขั้นตอนย่อยแสดงในรูปที่ 2.4

ตารางที่ 2.1 แสดงสมการทางคณิตศาสตร์ที่ใช้ในการประมวลผลแต่ละรอบ

| รอบที่ (r) | ขั้นตอนที่ (t) | สมการทางคณิตศาสตร์ |
|---|---|---|
| 1 | $0 \leq t < 20$ | $(B \wedge C) \vee (\overline{B} \wedge D)$ |
| 2 | $20 \leq t < 40$ | $B \oplus C \oplus D$ |
| 3 | $40 \leq t < 60$ | $(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$ |
| 4 | $60 \leq t < 80$ | $B \oplus C \oplus D$ |

$$W[t] = S[1]\,(W[t-16] \square W[t-14] \square W[t-8] \square W[t-3]) \qquad (2.1)$$

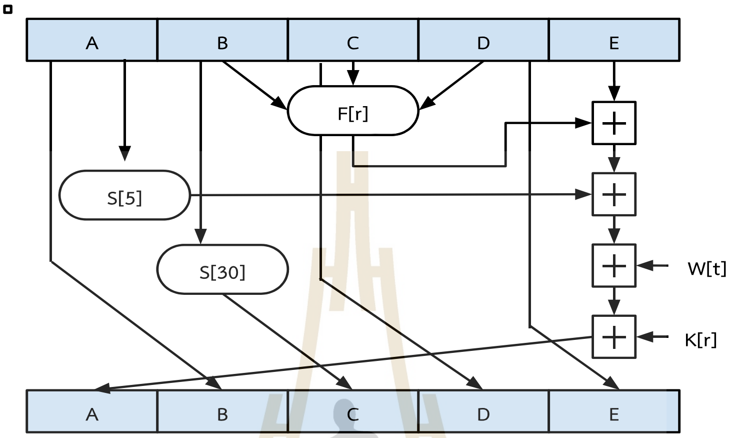
เมื่อ W[t] คือบล็อกข้อมูลที่ตำแหน่ง t

S[n] คือการเลื่อนบิตไปทางซ้ายเป็นวงลม n ครั้ง

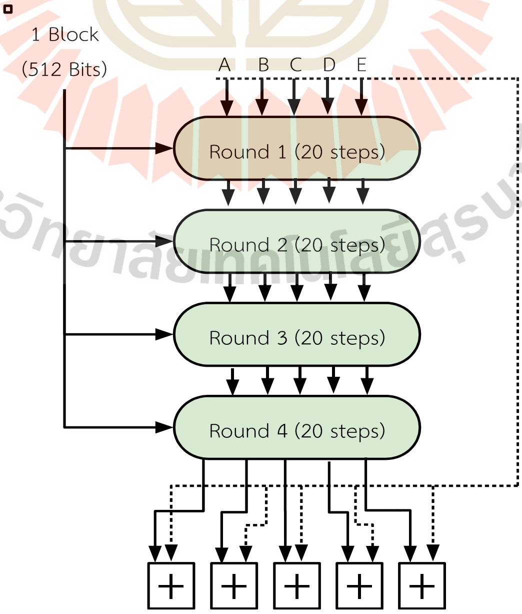
I) รีจิสเตอร์ A, B, C, D และ E จากขั้นตอนก่อนหน้า
II) ค่าคงที่ K ในแต่ละรอบรอบที่ 1=5A827999, 2=6ED9EBA1, 3=8F1BBCDC, 4=CA62C1D6

มีการสลับค่ารีจิสเตอร์ใหม่ตามตำแหน่งหัวลูกศร และเมื่อสิ้นสุดทุกรอบการทำงานและทุกขั้นตอนย่อยแล้วของแต่ละบล็อกข้อมูลแล้วระบบจะบวกค่ารีจิสเตอร์ผลลัพธ์จาก

กระบวนการกับค่ารีจิสเตอร์เดิมเข้าด้วยกันดังรูปที่ 2.5 ในกระบวนการ SHA-1 นี้มีข้อกำหนดว่าตัวแปรทุกตัวจะถูกหุ้ม (Wrap) ด้วย $2^{32}$ ยกเว้นความยาวของข้อมูลที่มีขนาด 64 บิตและค่ารหัสที่มีความยาว 160 บิต



รูปที่ 2.4 แผนภาพกระบวนการย่อยในแต่ละขั้นตอนของการเข้ารหัสทางเดียว SHA-1



รูปที่ 2.5 แผนภาพกระบวนการในแต่ละบล็อกของข้อมูล (ธนา หงษ์สุวรรณ, n.d., pp. 5-8)

## 2.5 ลักษณะของซอฟต์แวร์ประมวลผลแบบกระจาย

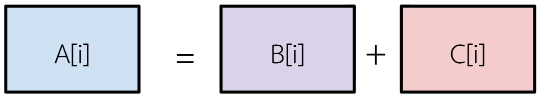
### 2.5.1 หลักการพิจารณาส่วนของระบบที่สามารถทำงานพร้อมกัน

ซอฟต์แวร์ที่ถูกใช้ในระบบกระจายนั้นมีหลายตัวให้เลือกใช้ แต่ละตัวก็จะมีความซับซ้อนและลักษณะเฉพาะตัวแตกต่างกันไป และมีทั้งซอฟต์แวร์ที่มีค่าใช้จ่าย ไม่มีค่าใช้จ่ายและเปิดเผยต้นรหัส การพิจารณางานที่สามารถประมวลผลแบบกระจายหรือแบบขนานได้ (Quinn, 2003) โดยหลักแล้วต้องพิจารณาข้อมูลดังนี้

1) กราฟการขึ้นต่อกันของข้อมูล (Data Dependency Graph) จะเป็นกราฟระบุทิศทางเพื่อพิจารณางานที่ต้องทำให้เสร็จก่อนจึงจะทำงานอื่นต่อไปได้ เพื่อใช้วิเคราะห์ตำแหน่งที่จะสามารถทำงานแบบไม่ขึ้นต่อกันได้ ถ้า *u* มีเส้นเชื่อมและหัวลูกศรชี้เข้าหา *v* แล้ว *v* จะขึ้นต่อ *u* จากรูปที่ 2.6 จะแสดงให้เห็นว่าต้องตัดไฟฟ้าก่อนจึงจะเชื่อมต่อสายโทรศัพท์ได้และต้องตัดไฟฟ้าก่อนจึงจะเปลี่ยนสวิทซ์ไฟฟ้าได้ จะเห็นว่างานทั้งสองสามารถทำไปพร้อมกันได้ แต่ต้องเสร็จก่อนจึงจะเชื่อมต่อไฟฟ้า และจะเห็นว่างานที่อยู่ฝั่งหัวลูกศรชี้ไปจะขึ้นอยู่กับงานที่อยู่ฝั่งหางของลูกศร ต้องทำงานฝั่งหางลูกศรให้เสร็จก่อน

รูปที่ 2.6 แสดงความขึ้นต่อกันของข้อมูล

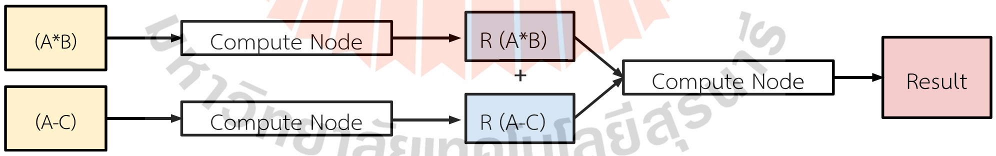
2) การขนานของข้อมูล (Data Parallelism) ลักษณะที่ข้อมูลนั้นมีการปฏิบัติการแบบเดียวกัน แต่ใช้ข้อมูลที่ต่างกัน เช่น กรณีมีการวนซ้ำ N ครั้งบนตัวแปร A, B และ C ซึ่งเป็นข้อมูลนำเข้าที่ไม่ขึ้นต่อกันจะพบว่าสามารถทำพร้อมกันได้ ดังรูปที่ 2.7 (การขนานของข้อมูลที่กล่าวถึงการปฏิบัติงานเดียวกันโดยใช้ข้อมูลที่ต่างกัน ในขณะที่กราฟการขึ้นต่อกันอธิบายความเชื่อมโยงกันของการใช้ข้อมูล)



```
For i from 0 to N
    A[i] = B[i] + C[i]
Endfor
```

รูปที่ 2.7 แสดงข้อมูลนำเข้าที่เป็นอิสระต่อกัน (Quinn, 2003)

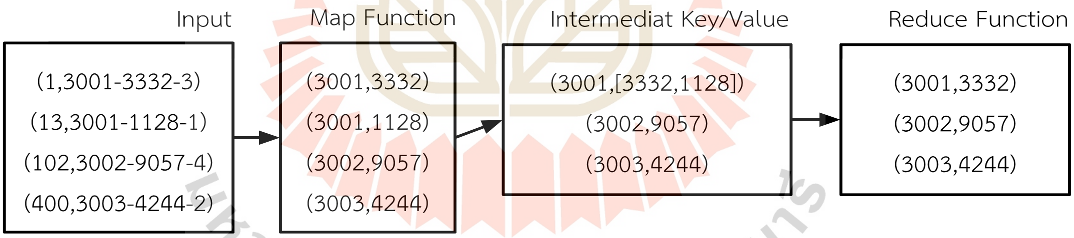
3) การขนานของฟังก์ชัน (Function Parallelism) เป็นการปฏิบัติการประมวลผลที่แยกออกจากกันได้อย่างอิสระแม้ข้อมูลนั้นจะมีความสัมพันธ์กัน และผลลัพธ์ของการแยกส่วนแล้วรวมกลับนั้นมีค่าถูกต้องการหลักการประมวลผล เช่น ในกรณี (A*B)+(A-C) ที่สามารถแยกการประมวลผลระหว่าง A*B กับ A-C ออกไปประมวลผลแล้วนำผลลัพธ์ที่ได้กลับมาบวกกันอีกครั้งแล้วยังได้ค่าตรงตามหลักการคำนวณ จากรูปที่ 2.8 แสดงการกระจายการประมวลผลไปที่โหนดอื่นได้ โดยที่ยังคงความถูกต้องตามหลักการประมวลผลไว้



รูปที่ 2.8 แสดงการทำงานขนานกันของฟังก์ชัน

### 2.5.2 ตัวแบบการโปรแกรมแบบ MapReduce

MapReduce เป็นตัวแบบการโปรแกรมที่เกี่ยวข้องกับการพัฒนาซอฟต์แวร์เพื่อประมวลผลชุดข้อมูลขนาดใหญ่ ผู้ใช้จะกำหนดฟังก์ชัน Map[1] ซึ่งเป็นลักษณะของคู่ Key และ Value ให้กับข้อมูลของตนเองเพื่อที่ระหว่างประมวลผลจะเกิด Intermediate Key/Value ซึ่งเป็นคู่ของ Key และ Value ที่จะถูกฟังก์ชัน Reduce รวบรวมชุดของ Value ที่มี Key เดียวกันเข้าไว้ด้วยกัน ทั้งฟังก์ชัน Map และ Reduce ผู้ใช้จะเป็นคนกำหนดตรรกะเอง (Dean and Ghemawat, 2008) ดังรูปที่ 2.9 แสดงให้เห็นตัวอย่างชุดของข้อมูลนำเข้าที่มีจำนวน 4 Maps โดยตัว Key ในกรณีข้อมูลนำเข้านี้ไม่มีนัยสำคัญ (ในแผนภาพเป็นเลขที่แสดงเลขบรรทัดเท่านั้น) และ Value อยู่ในรูปแบบ *หมายเลขสาขา 4 หลัก-มูลค่าซื้อขาย 4 หลัก-จำนวนการซื้อขาย* จากนั้นจะทำการพัฒนาฟังก์ชัน Map ให้สามารถอ่าน Value แล้วแยกค่าตำแหน่งที่ 1 ถึง 4 (เช่น 3001) มาเป็น Key จากนั้นนำตำแหน่งที่ 6 ถึง 9 มาเป็น Value ในขั้นตอนนี้ เมื่อเสร็จฟังก์ชัน Map ตัวไลบรารี่จะรวมข้อมูลที่ Key ตรงกันเพื่อเอา Value รวมไว้ไว้เป็นชุดเดียวกัน (เช่นกรณี Key=3001 Value=[3332,1128]) จากนั้นจะสั่งให้ทำงานฟังก์ชัน Reduce ซึ่งพัฒนาเพื่อหาค่าที่สูงที่สุดของมูลค่าซื้อขาย ก็จะได้ค่าที่สูงที่สุดในแต่ละสาขา

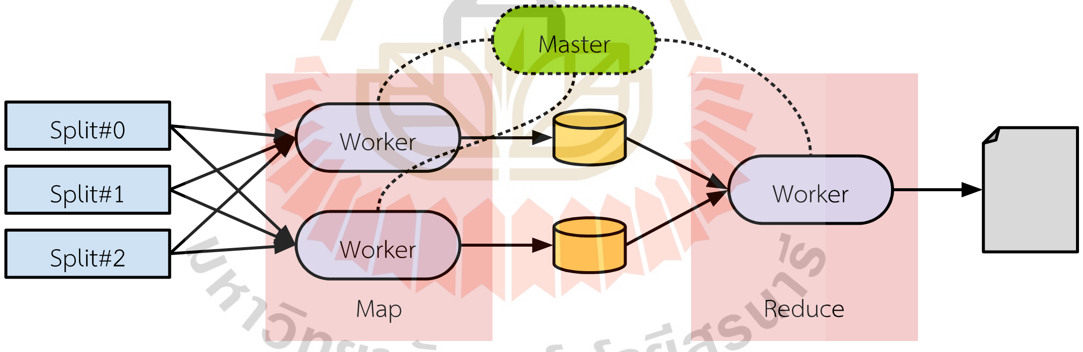


รูปที่ 2.9 ทิศทางการไหลของข้อมูลเมื่อประมวลผลแบบ MapReduce

---

[1] Map เป็นโครงสร้างข้อมูลประเภทหนึ่งมีลักษณะเป็นคู่กัน ซึ่งมีข้อมูลตัวหน้าเรียกว่า Key และข้อมูลตัวหลังเรียกว่า Value เช่น (3,5) 3 คือ Key และ 5 คือ Value และฟังก์ชัน Map เป็นฟังก์ชันการทำงานที่แปลงข้อมูลให้อยู่ในโครงสร้างแบบ Map

รูปแบบการพัฒนาซอฟต์แวร์งานซึ่งอธิบายทุกอย่างให้อยู่ในรูปของ Map ดังกล่าวทำให้การพัฒนาซอฟต์แวร์สะดวกขึ้น ผู้ใช้สามารถมุ่งเน้นเพียงตรรกะทางธุรกิจของตนเอง เนื่องจากความยุ่งยากของการพัฒนาซอฟต์แวร์ในระบบประมวลผลแบบกระจายนั้นจะถูกซ่อนไว้ เช่น ระบบคัดเลือกโหนดทำงาน (Worker Node) ระบบแบ่งงาน ระบบจัดการการล้มเหลว ระบบกระจายงาน เป็นต้น ซึ่งช่วยประหยัดเวลาของผู้ใช้ได้อย่างมาก MapReduce สามารถอธิบายกลไกการทำงานได้ดังในรูปที่ 2.10 คือโปรแกรมแบบ MapReduce ข้อมูลที่รับมาจะถูกแบ่งเป็นส่วนย่อย ๆ จากนั้นโหนดผู้นำ (Master Node) จะสั่งให้โหนดทำงานประมวลผลฟังก์ชัน Map โดยจะสั่งโหนดทำงานโดยเลือกตามความเหมาะสม และเมื่อโหนดทำงานประมวลผลเสร็จเป็นรอบ ๆ แล้วก็จะบันทึกลงหน่วยเก็บข้อมูลภายใน จากนั้นจะแจ้งตำแหน่งเก็บกลับไปที่โหนดผู้นำ จากนั้นโหนดผู้นำก็จะทำหน้าที่ส่งคำสั่งและตำแหน่งเก็บข้อมูลให้โหนดทำงานที่เหมาะสมในการประมวลผลฟังก์ชัน Reduce เมื่อการประมวลผลเสร็จแล้วก็จะบนทึกลงแล้วจึงจะทำงานอื่นต่อไปซึ่งจะได้ขยายความดังนี้



รูปที่ 2.10 สถาปัตยกรรมของ MapReduce (Dean and Ghemawat, 2008)

1) ระบบมีการแบ่งงานออกเป็นส่วนย่อย ๆ เพื่อกระจายงานให้โหนดบนคลัสเตอร์ที่จะใช้ประมวลผลแบบกระจายเป็นการแบ่งภาระงานให้อยู่ในระดับเดียวกัน ไม่ให้เกิดภาระงานล้น (Overload) ที่โหนดใดโหนดหนึ่ง เพื่อป้องกันการรอผลลัพธ์เนื่องจากระบบจะทำงานเสร็จสิ้นก็ต่อเมื่องานย่อยทั้งหมดเสร็จสิ้น

2) โหนดผู้นำจะแจ้งว่าให้โหนดทำงานใดบ้างทำงาน และทำงานกับข้อมูลย่อยส่วนใด ในส่วนนี้ขึ้นอยู่กับกลยุทธ์ของซอฟต์แวร์แต่ละตัวว่าจะกำหนดอย่างไรจึงจะเหมาะสม
3) โหนดทำงานจะทำงานตามตัวแบบการโปรแกรม MapReduce ในขั้นตอนของฟังก์ชัน Map โดยข้อมูลที่ทำเสร็จจะถูกบันทึกลงแหล่งเก็บข้อมูลโลคอลเป็นรอบ ๆ จากนั้นจะส่งตำแหน่งเก็บไฟล์ไปให้โหนดผู้นำ
4) โหนดผู้นำจะสั่งโหนดทำงานให้ทำงานโดยเลือกจากโหนดทำงานที่เหมาะสมและจะส่งตำแหน่งแหล่งเก็บข้อมูลที่ได้จากการทำงานฟังก์ชัน Map ไปให้ด้วย เมื่อโหนดทำงานจะประมวลผลฟังก์ชัน Reduce จะเรียกข้อมูลจากแหล่งเก็บมาสร้าง Intermediate Key/Value ซึ่งก็คือการจัดข้อมูลที่ Key เดียวกันไว้ในกลุ่มเดียวกัน
5) เมื่อได้ Intermediate Key/Value แล้วก็จะส่งต่อให้ฟังก์ชัน Reduce ทำงานตามที่ผู้ใช้กำหนด เสร็จแล้วจึงคืนกลับไปทำงานในส่วนอื่น

## 2.6 ซอฟต์แวร์ Spark

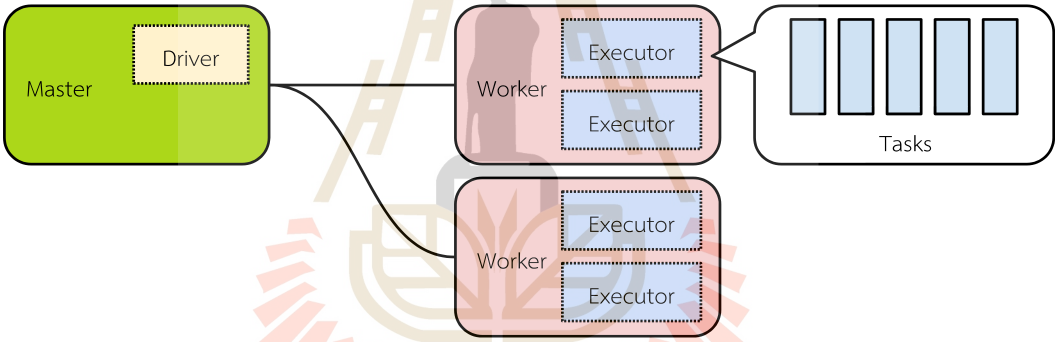
ซอฟต์แวร์ Spark สร้างจากห้องปฏิบัติการ AMPLab[2] ของมหาวิทยาลัย UC Berkeley เมื่อปี ค.ศ. 2009 (Karau et al., 2015) จากนั้นได้เปิดเผยต้นรหัสและอยู่ภายใต้การดูแลของ Apache Software Foundation (ASF) ในเดือนมิถุนายน ปี ค.ศ. 2013 ตัว Spark ถูกพัฒนาเพื่อปรับปรุงจากข้อจำกัดที่มีใน Apache Hadoop หากจะเปรียบเทียบกันโดยการประมวลผลข้อมูลโดยใช้ขั้นตอนวิธี Logistic Regression พบว่า Spark ทำงานได้เร็วกว่า Hadoop MapReduce ถึง 100 เท่า (Shoro and Soomro, 2015) เนื่องจาก Hadoop นั้นทำงานบนแหล่งเก็บข้อมูลลักษณะดิสก์ซึ่งใช้เวลานานในการเข้าถึง ขณะที่ Spark ประมวลผลในหน่วยความจำซึ่งเข้าถึงข้อมูลได้อย่างรวดเร็ว ตัวแบบการโปรแกรมของ Spark เป็นแบบ MapReduce ทำให้พัฒนาซอฟต์แวร์งานได้อย่างรวดเร็ว ตัวกรอบงานถูกสร้างด้วยภาษาโปรแกรม Scala เป็นหลัก และตัวกรอบงานเองก็รองรับมากกว่า 3 ภาษาในการพัฒนาซอฟต์แวร์ที่สามารถใช้งานกรอบงานได้ คือ ภาษาโปรแกรม Python, Java และ Scala

---

[2] ชื่อเดิม RAD Lab

เอง นอกจากนี้จากการตอบแบบสอบถามพบว่ามีมากกว่า 500 บริษัทที่ใช้ Spark หรือมีแผนที่จะใช้งาน Spark บนสภาพแวดล้อมการทำงานจริง (Production Environment) ภายในปี 2015 (Zaharia, 2015) งานที่เหมาะจะประมวลผลในรูปแบบโปรแกรมแบบ MapReduce จะเป็นงานที่มีการประมวลผลข้อมูลที่มีขนาดใหญ่ ไม่ต้องการประมวลผลแบบคำนวณซ้ำชุดข้อมูลเดิม (Kang et al., 2015)

รูปแบบการติดต่อประสานงานระหว่างแต่ละองค์ประกอบดังแสดงในรูปที่ 2.11 ซึ่งตัวโหนดผู้นำจะมีตัวขับ (Driver) เพื่อใช้ควบคุมการทำงานและสั่งงานโหนดทำงาน และโหนดทำงานจะมี ตัวกระทำการ (Executor) เพื่อประมวลผลงานซึ่งแต่ละโหนดทำงานมีตัวกระทำการหลายตัวได้



รูปที่ 2.11 แผนภาพแสดงการประสานงานระหว่างโหนดผู้นำ
และโหนดทำงานของ Spark (Xu, 2015b)

### 2.6.1 สถาปัตยกรรมของ Spark

ระบบของ Spark ประกอบด้วยองค์ประกอบดังนี้

1) ตัวขับเป็นซอฟต์แวร์ที่ผู้ใช้พัฒนาขึ้นมาเพื่อใช้ในการควบคุมการประมวลผลโดยรวมของระบบซึ่งจะต้องมีตัวแปรของ SparkContext เป็นส่วนประกอบสำหรับสั่งงานระบบและติดต่อกับตัวกระทำการเพื่อรับส่งข้อมูลระหว่างกัน การตั้งค่าต่าง ๆ ให้กับระบบนั้นจะทำผ่านตัวแปรของ SparkConf ซึ่งอยู่ในรูป

คู่ Key และ Value ระบบจะอนุญาตให้มีเพียง SparkContext ตัวเดียวในแต่ละ JVM เท่านั้น

2) โหนดผู้นำ เนื่องจากระบบเชื่อมต่อกันอยู่ในลักษณะคลัสเตอร์ของโหนดทำงาน การสั่งงานจะสั่งผ่านโหนดผู้นำซึ่ง Spark ยอมให้นักพัฒนาสามารถเลือกใช้ตัวจัดการคลัสเตอร์ได้หลากหลาย เช่น YARN, Mesos เป็นต้น

3) โหนดทำงาน จะมีตัวกระทำการซึ่งทำงานอยู่ภายในแต่ละโหนด และเป็นที่อยู่ของตัวจัดการบล็อก (Block Manager)

4) ตัวกระทำการ เป็นเอเจนท์ (Agent) ที่กระจายตามโหนดทำงานซึ่งแต่ละโหนดสามารถมีได้หลายตัวกระทำการแบ่งตามโปรแกรมประยุกต์ที่ทำงานอยู่ในโหนดทำงานนั้น ซึ่งส่วนนี้เป็นส่วนที่จะประมวลผลงานหลักในระบบและส่วนนี้จะจัดการแหล่งเก็บข้อมูลทั้งในหน่วยความจำและแหล่งเก็บข้อมูลอื่นให้กับงาน เมื่อตัวกระทำการเริ่มการทำงานตัวมันเองจะลงทะเบียนไปยังตัวขับ เพื่อรับคำสั่งประมวลผลของงานโดยตรงจากตัวขับซึ่งคำสั่งแต่ละส่วนจะถูกแปลงเป็นรหัสไบต์ (Bytecode) และส่งผ่านโดยโปรโตคอล HTTP ไปยังตัวกระทำการ

### 2.6.2 Resilient Distributed Dataset

RDD เป็นลักษณะของตัวเก็บชุดข้อมูลที่ไม่สามารถเปลี่ยนค่าได้ ซึ่งใน Spark จะใช้เป็นมุมมองนามธรรมของหน่วยความจำซึ่งจะถูกใช้กับการประมวลผลในหน่วยความจำ แต่ละ RDD จะถูกแบ่งย่อยเป็นส่วน ๆ เพื่อกระจายการประมวลผลไปยังโหนดในระบบประมวลผลแบบกระจาย ซึ่ง RDD เหล่านี้จะไม่สามารถเปลี่ยนแปลงค่าตัวเองได้ (Zaharia et al., 2012a) ตัว RDD ไม่ได้ใช้แหล่งเก็บข้อมูลทางกายภาพเพื่อเก็บองค์ประกอบ (Element) แต่เก็บข้อมูลที่เพียงพอต่อการคำนวณ RDD จากแหล่งเก็บข้อมูลซึ่งลักษณะนี้จะเรียกว่าสายตระกูลของข้อมูลที่เก็บอยู่ในการประมวลผลเกิดขึ้นจริงเมื่อมีการสั่งคำสั่งที่เป็นลักษณะการกระทำ (Action) ตัว RDD สามารถสร้างขึ้นได้ด้วย 4 วิธี (Zaharia et al., 2010) ดังนี้

1) อ่านข้อมูลจากไฟล์ที่มีอยู่ในแหล่งเก็บข้อมูล

2) สั่ง Parallelize ข้อมูลที่เป็นลักษณะ Collection เช่น Array, List เป็นต้น

3) จากการแปลงของ RDD เดิม เช่น เมื่อมีการสั่ง Filter RDD ตัวเดิมจะถูกแปลงเป็น RDD ตัวใหม่

4) แปลงจาก RDD ที่ถูกเก็บไว้ในแหล่งเก็บชั่วคราวซึ่งเป็นผลจากกระบวนการแคชหรือบันทึกให้กลายเป็น RDD อีกครั้ง

### 2.6.3 การดำเนินการ (Operation)

การปฏิบัติงานใน Spark มีลักษณะเป็น Lazy Evaluation ระบบจะประมวลผลก็ต่อเมื่อมีความจำเป็นที่ต้องใช้งานนั้น ๆ แล้ว การดำเนินการกับ RDD แบ่งเป็น 2 แบบคือ

1) การแปลง (Transformations) การดำเนินการประเภทนี้จะให้ผลลัพธ์เป็น RDD ตัวใหม่กลับออกมา ซึ่ง RDD ตัวใหม่จะมีรูปใหม่ เช่น filter, map, union เป็นต้น ซึ่งยังไม่เกิดการประมวลผลจริงกับชุดข้อมูลในการปฏิบัติงานขั้นตอนนี้
2) การกระทำ การดำเนินการประเภทนี้จะเป็นการสั่งให้ระบบทำงานประมวลผล ดังนั้นการประมวลผลจะเกิดขึ้นหลังจากที่สั่งดำเนินการประเภทนี้ ตัวอย่างคำสั่ง เช่น collect, count เป็นต้น

## 2.7 งานวิจัยที่เกี่ยวข้อง

Moody A. และคณะ (2010) พัฒนาไลบรารี่ Scalable Checkpoint/Restart (SCR) เพื่อใช้สร้างจุดตรวจสอบแบบหลายระดับ (Multi-Level Checkpointing) ซึ่งระบบจะแบ่งการสร้างจุดตรวจสอบเป็น 3 ระดับเรียงลำดับจากระดับต่ำสุดไปสูงสุด คือ LOCAL, PARTNER/XOR และระบบไฟล์แบบขนาน Lustre การกู้คืนแบ่งตามระดับจุดตรวจสอบซึ่งระดับที่สูงจะมีค่าใช้จ่ายสูงความยืดหยุนสูงกว่าและจะสามารถกู้คืนระดับต่ำกว่าได้ และมีการเปรียบเทียบประสิทธิภาพจุดตรวจสอบระดับ LOCAL กับ Lustre พบว่าระดับ LOCAL เร็วกว่าถึง 1,000 เท่า และประสิทธิภาพของระบบเมื่อใช้ SCR สูงถึงร้อยละ 85 และลดความถี่ของการสร้างจุดตรวจสอบบนระบบไฟล์แบบขนานได้ 2-4 เท่า

Jangjaimon I. และ Tzeng N. F. (2013) นำเสนอการใช้ Adaptive Incremental Checkpointing ซึ่งเป็นการประยุกต์ใช้งานจุดตรวจสอบโดยไม่กำหนดรอบเวลาการสร้างจุดตรวจสอบอย่างชัดเจน เนื่องจากพบว่าในบางครั้งจุดตรวจสอบที่เหมาะสมอาจไม่ได้เป็นรอบเวลาเดิม ผู้วิจัยเชื่อถือขั้นตอนวิธี Stepwise Regression และ Gradient Descent Algorithm เพื่อเลือกเวลาสร้างจุดตรวจสอบ และผู้วิจัยได้ใช้กระบวนการบีบอัดข้อมูลแบบ Xdelta3-PA ในการบีบอัดข้อมูลหน่วยความจำเพื่อสร้างจุดตรวจสอบโดยใช้หน่วยประมวลผลที่ว่างจากการประมวลผลงานหลัก ซึ่งพบว่าสามารถลดเวลาที่ใช้จากเริ่มต้นการทำงานของซอฟต์แวร์ไปจนถึงสิ้นสุดกระบวนการทำงานได้ร้อยละ 47 เมื่อเทียบกับการสร้างจุดตรวจสอบแบบกำหนดรอบเวลาอย่างชัดเจน โดยใช้เวลาเพิ่มขึ้นจากกระบวนการปกติร้อยละ 2.6 เมื่อเทียบกับการไม่นำระบบจุดตรวจสอบมาใช้งานเลย

Zaharia M. และคณะ (2012b) พัฒนาสตรีมไม่ต่อเนื่องชื่อ D-Stream ซึ่งลักษณะข้อมูลที่เกิดขึ้นต่อเนื่องกันทำให้ระบบไม่สามารถหาจุดสิ้นสุดเพื่อประมวลผลข้อมูลแบบนี้ได้ จึงมีการ

พัฒนากลไกดังกล่าวเพื่อกำหนดช่วงเวลาเป็นรอบ ๆ เพื่อให้ข้อมูลนำเข้าอยู่ในรูปชุดของ RDD ที่สามารถนำไปประมวลผลได้ และผู้วิจัยได้นำเสนอกระบวนการกู้คืนแบบขนาน (Parallel Recovery) ซึ่งใช้จุดตรวจสอบของ RDD เป็นรอบ ๆ เพื่อป้องกันข้อผิดพลาดจนต้องเริ่มประมวลผลใหม่ทั้งหมด ในงานบางงานสามารถเริ่มทำงานบนหลายโหนดพร้อมกันได้

Ferreira K. B. และคณะ (2014) ลดโอเวอร์เฮดและเพิ่มประสิทธิภาพการสร้างจุดตรวจสอบโดยใช้ไลบรารี่ libhashckpt คล้ายกับในกระบวนการป้องกันเพจแล้วนำเข้ากระบวนการฟังก์ชันเข้ารหัสทางเดียวเพื่อตรวจหาความแตกต่างของข้อมูลหน่วยความจำ ซึ่งใช้บล็อกข้อมูลขนาด 512 ไบต์แทนการใช้ทั้งเพจ คือ 4 กิโลไบต์ในกระบวนการกำหนดการสร้างจุดตรวจสอบเฉพาะส่วนของหน่วยความจำที่มีการเปลี่ยนแปลงเท่านั้นที่จะถูกสร้างจุดตรวจสอบ โดยงานจะถูกส่งไปประมวลผลที่ GPU เนื่องจากพบว่าการเข้ารหัสทางเดียวเหมาะสมกว่าการใช้หน่วยประมวลผลกลาง วิธีนี้สามารถลดขนาดของจุดตรวจสอบให้เหลือประมาณร้อยละ 15 จากวิธีเดิมและขนาดมากกว่าการจัดการด้วยมือร้อยละ 35

Maruyama M., Tsumura T. และ Nakashima H. (2005) นำเสนอการใช้ล็อกการทำงานของไลบรารี่ MPI เพื่อทำ Data-Replay โดยบันทึกข้อมูลเนื้อหาของข้อความที่โปรเซสได้รับมาทำให้สามารถสั่งทำงานหรือหยุดโปรเซสได้อย่างอิสระ กลไกการหยุดที่จุดตรวจสอบใด ๆ และสามารถย้อนกลับหรือไปข้างหน้านี้ทำให้เกิดการตรวจสอบการทำงานของโปรแกรมแบบขนานได้ โอเวอร์เฮดที่เกิดขึ้นจากการเก็บข้อมูลด้วยวิธีนี้เฉลี่ยร้อยละ 24 ขณะที่สามารถเร่งความเร็วของกระบวนการทำซ้ำ (replay) ได้เฉลี่ยร้อยละ 38

Dinh M. N. และคณะ (2011) พิสูจน์แนวคิดการยืนยัน (Assertion) การทำงานของโปรแกรมในระบบประมวลผลแบบขนาน โดยแสดงตัวอย่างการใช้งานโดยทดสอบกับโปรแกรมคำนวณสมการ Shallow Water Equations โดยงานวิจัยได้แสดงตัวอย่างการกระจายข้อมูลเพื่อประมวลผลในโหนดแม่ข่ายตรวจแก้จุดบกพร่องตามสถาปัตยกรรมของซอฟต์แวร์ Guard ซึ่งมีลักษณะคล้ายกับการทำงานของตัวตรวจแก้จุดบกพร่อง GDB ที่สามารถกำหนดจุดที่ต้องการตรวจสอบ เซ็ตของโปรเซสที่จะตรวจสอบและตั้งข้อกำหนดการทดสอบได้ ซึ่งช่วยในการทดสอบการทำงานเพื่อยืนยันการทำงานได้ถูกต้อง

Schwartz-Narbonne D. และคณะ (2011) ได้ดัดแปลง LLVM เพื่อตรวจสอบ AST สำหรับใช้ตรวจสอบความถูกต้องของโปรแกรมแบบขนาน ซึ่งสามารถตรวจสอบได้ทั้งสถานะการทำงานปัจจุบัน และลักษณะการกระทำที่อาจจะไม่ครอบคลุมหากตรวจสอบแบบเดียวกับโปรแกรมแบบลำดับโดยแสดงการแก้ปัญหา Interference, Statement Atomicity, Thread Safety และ ABA problem โดยใช้เครื่องมือที่ปรับแต่งได้

Shetty A. และ Marshall N. (2014) นำเสนอการทดสอบซอฟต์แวร์บนระบบ Spark ทั้งส่วนของ Spark Context และ Spark Streaming โดยกรณีทดสอบระดับหน่วย และการทดสอบความเครียดของคลัสเตอร์ผ่านทางการเรียกใช้ RESTful API และวัดผลผ่านการต่อประสานทาง Command Line และการต่อประสานทางเว็บได้ การทดสอบนี้ทำในโหมดโลคอลบนโหนดผู้นำ

Gulzar M. A.และคณะ (2016) ได้นำเสนอ BigDebug ซึ่งเป็นกลไกที่ออกแบบมาเพื่อช่วยตรวจแก้จุดบกพร่องของการทำงานของโปรแกรมใน Spark สำหรับการประมวลผลกับข้อมูลขนาดใหญ่ แต่พบว่าซอฟต์แวร์ต้องการให้ผู้ใช้มีปฏิสัมพันธ์ด้วนเพื่อให้งานนั้นสำเร็จ เช่น การเขียนไฟล์เมื่อเสร็จสิ้นการทำงานนั้นไม่สามารถทำงานได้จนกว่าจุดหยุดตำลอง (Simulation Breakpoint) จะถูกเลือกคำสั่งการทำงานอย่างน้อยหนึ่งคำสั่ง และสั่งดำเนินการกระบวนการต่อ ซึ่งผู้ใช้จะต้องกำหนดการทำงานของคำสั่งเหล่านี้เอง อีกทั้งยังไม่สนับสนุนการเริ่มต้นการทำงานของ Spark ใหม่ ซึ่งการพัฒนาซอฟต์แวร์โดยทั่วไปจะไม่สามารถค้างการประมวลผลจนกระทั่งเสร็จสิ้นการประมวลผล อีกทั้งโปรแกรมการประมวลผลอาจจะถูกเปลี่ยนใน RDD แต่ละตัว ซึ่งระบบไม่สามารถตรวจสอบได้และหากการเปลี่ยนแปลงโปรแกรมดังกล่าวนั้นกระทบกับทุกเรคอร์ดข้อมูลทำให้เกิดความผิดพลาดของข้อมูลที่ประมวลผลโดยใช้โปรแกรมชุดเดิมและโปรแกรมชุดใหม่ และเนื่องจากผู้ใช้ต้องปรับเปลี่ยนการพัฒนาโปรแกรมให้อยู่ในรูปแบบของ BigDebug และปัจจุบันสนับสนุนเพียง Spark รุ่น 1.2.1 เท่านั้น จึงทำให้นักพัฒนาไม่สะดวกในการใช้เครื่องมือนี้

Sharma P. และคณะ (2016) นำเสนอ Flint ซึ่งใช้กลไกการสร้างจุดตรวจสอบแบบดั้งเดิมของ Spark โดยเน้นการใช้งานจุดตรวจสอบกับเครื่องคอมพิวเตอร์เช่าใช้ชั่วคราว (Transient Instance) ซึ่งเป็นเครื่องคอมพิวเตอร์เสมือน (Virtual Machine) ที่มีราคาถูกแต่อาจจะมีการเรียกคืนการใช้งานเมื่อใดก็ได้ พื้นที่ที่ใช้สร้างจุดตรวจสอบของทุกพาร์ทิชัน (Partition) ของ RDD จุอยู่บนเครื่องคอมพิวเตอร์เช่าตามความต้องการใช้ (On-demand Instance) ซึ่งมีราคาสูงกว่าแต่จะไม่มีการเรียกคืนโดยผู้ให้บริการผ่าน HDFS แต่ก็พบว่ากลไกดังกล่าวไม่สามารถใช้ได้เมื่อมีการเริ่มทำงานของ Spark ใน JVM ใหม่ อีกทั้งกลไกการสร้างจุดตรวจสอบแบบนี้จะถูกเรียกโดยระบบอัตโนมัติ ทำให้หากออกแบบคลัสเตอร์ของ Spark ไม่เหมาะสมเนื่องจากไม่สามารถทราบขนาดของพื้นที่ที่ต้องการของระบบอาจจะส่งผลให้ระบบผิดพลาด

Yan Y. และคณะ (2016) TR-Spark มีลักษณะที่คล้ายกับ Flint แต่จะปรับความละเอียดของกลไกการสร้างจุดตรวจสอบได้มากกว่าเนื่องจากใช้การสร้าจุดตรวจสอบในระดับตัวงาแทนการสร้างในระดับ RDD ของ Flint ทำให้ช่วยลดข้อมูลที่ต้องเก็บ แต่ก็พบว่ากลไกของ TR-Spark ต้องการทราบถึงความน่าจะเป็นที่จะเกิดการเรียกคืน VM ซึ่งต้องเตรียมข้อมูลเพื่อคำนวณความ

น่าจะเป็นที่ VM จำเกิดความผิดพลาดเพื่อให้สามารถใช้ได้อย่างเหมาะสม และยังไม่สามารถใช้งานได้ในกรณีที่มีการแก้ไขตัวโปรแกรม

Zhu W., Chen H. และ Hu F. ได้นำเสนอ ASC ซึ่งออกแบบมาเพื่อหา RDD ที่เหมาะสม โดยวิเคราะห์ความคุ้มค่าของการสร้างจุดตรวจสอบและการกู้คืนจุดตรวจสอบเปรียบเทียบกับกรณีที่ไม่ใช้งานจุดตรวจสอบ โดยตรวจสอบจากสายตระกูลแต่ก็พบว่าไม่มีความคงทนต่อการเริ่มต้นใหม่ และไม่สามารถใช้งาน RDD ซ้ำหลายครั้งซึ่งจะช่วยลดระยะเวลาในการทดสอบซอฟต์แวร์ได้ เนื่องจากไม่มีกลไกที่จะทราบว่าเป็น RDD ตัวเดิมหรือคล้ายเดิมที่เคยคำนวณไว้แล้วแต่อย่างใด

สำนักข่าว Bloomberg (2017) เปิดให้เข้าถึง Spark-flow เครื่องมือที่ออกแบบให้สามารถกู้คืนจุดตรวจสอบใน Spark ได้แม้ JVM จะถูกปิดไปแล้ว โดยใช้ Distributed Collection (DC) ซึ่งคล้ายกับ Dataset API โดยอ่านค่าจากรหัสไบต์ของ RDD และของฟังก์ชันไม่ระบุตัวตน (Anonymous Function) ผ่านเครื่องมือ ASM (Kuleshov, 2017) แล้วจึงวิเคราะห์ตำแหน่งที่ RDD จะถูกสร้างจุดตรวจสอบโดยใช้ฟังก์ชันเข้ารหัสทางเดียวแบบ MD5 แต่ก็พบว่าการที่ใช้ DC เมื่อมีการเรียกใช้ DC.checkpoint() แล้วหลังจากจุดตรวจสอบสร้างเสร็จระบบยังต้องอ่านกลับขึ้นมาในคราวเดียวกันอีก และเมื่อมีการกู้คืนจุดตรวจสอบ เครื่องมือนี้จะบังคับให้เกิดการประมวลผล DC.count() ขึ้นอีก ซึ่งจะทำให้การประมวลผลจริงนั้นเกิดขึ้นซ้ำ ๆ และยังไม่มีกลไกเพื่อกระจายการประมวลผลไปยังคลัสเตอร์

ตารางที่ 2.2 ตารางสรุปเปรียบเทียบงานวิจัยที่เกี่ยวข้องกับการเพิ่มผลิตภาพของการทดสอบซอฟต์แวร์แบบกระจายโดยใช้จุดตรวจสอบ บทความที่เกี่ยวข้องประกอบไปด้วย “A” แทนงานวิจัยของ Moody A. และคณะ (2010) “B” แทนงานวิจัยของ Jangjaimon I. และ Tzeng N. F. (2013) “C” แทนงานวิจัยของ Zaharia M. และคณะ (2012b) “D” แทนงานวิจัยของ Ferreira K. B. และคณะ (2014) “E” แทนงานวิจัยของ Maruyama M. และคณะ (2005) “F” แทนงานวิจัยของ Dinh M. N. และคณะ (2011) “G” แทนงานวิจัยของ Schwartz-Narbonne D. และคณะ (2011) “H” แทนงานวิจัยของ Shetty A. และ Marshall N. (2014) “I” แทนวานวิจัยของ Gulzar M. A.และคณะ (2016) “J” แทนงานวิจัยของ Sharma P. และคณะ (2016) “K” แทนงานวิจัยของ Yan Y. และคณะ (2016) “L” แทนงานวิจัยของ Zhu W., Chen H. และ Hu F. “M” แทนงานวิจัยของ Bloomberg (2017) และ “N” แทนงานวิจัยเรื่อง การเพิ่มผลิตภาพของการทดสอบซอฟต์แวร์แบบกระจายโดยใช้จุดตรวจสอบ (งานวิจัยในวิทยานิพนธ์นี้)

| **กระบวนการทำงาน** | **งานวิจัยที่เกี่ยวข้อง** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** |
| **จุดตรวจสอบ** | | | | | | | | | | | | | | |
| มีการประยุกต์ใช้จุดตรวจสอบ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | ✓ | ✓ | ✓ | ✓ | ✓ |
| มีการบีบอัดไฟล์ข้อมูลแบบ Xdelta3-PA | | ✓ | | | | | | | | | | | | |
| เก็บจุดตรวจสอบบนระบบไฟล์แบบขนานหรือแบบกระจาย | ✓ | ✓ | | ✓ | | | | | | ✓ | ✓ | | | ✓ |
| ใช้ขั้นตอนวิธีเพื่อกำหนดจุดตรวจสอบ | | ✓ | | | | | | | | ✓ | ✓ | | | |
| กำหนดการสร้างจุดตรวจสอบด้วยตนเอง | ✓ | | ✓ | | | | | | | | | | ✓ | ✓ |
| ใช้เพจของหน่วยความจำ | | ✓ | | ✓ | ✓ | | | | | | | | | |
| ใช้ GPU เพื่อประมวลผลจุดตรวจสอบ | | | | ✓ | | | | | | | | | | |
| ใช้จุดตรวจสอบหลายระดับ | ✓ | ✓ | | | | | | | | ✓ | ✓ | | | |

ตารางที่ 2.3 ตารางสรุปเปรียบเทียบงานวิจัยที่เกี่ยวข้องกับการเพิ่มผลิตภาพของการทดสอบซอฟต์แวร์แบบกระจายโดยใช้จุดตรวจสอบ บทความที่เกี่ยวข้องประกอบไปด้วย “A” แทนงานวิจัยของ Moody A. และคณะ (2010) “B” แทนงานวิจัยของ Jangjaimon I. และ Tzeng N. F. (2013) “C” แทนงานวิจัยของ Zaharia M. และคณะ (2012b) “D” แทนงานวิจัยของ Ferreira K. B. และคณะ (2014) “E” แทนงานวิจัยของ Maruyama M. และคณะ (2005) “F” แทนงานวิจัยของ Dinh M. N. และคณะ (2011) “G” แทนงานวิจัยของ Schwartz-Narbonne D. และคณะ (2011) “H” แทนงานวิจัยของ Shetty A. และ Marshall N. (2014) “I” แทนวานวิจัยของ Gulzar M. A.และคณะ (2016) “J” แทนงานวิจัยของ Sharma P. และคณะ (2016) “K” แทนงานวิจัยของ Yan Y. และคณะ (2016) “L” แทนงานวิจัยของ Zhu W., Chen H. และ Hu F. “M” แทนงานวิจัยของ Bloomberg (2017) และ “N” แทนงานวิจัยเรื่อง การเพิ่มผลิตภาพของการทดสอบซอฟต์แวร์แบบกระจายโดยใช้จุดตรวจสอบ (ต่อ)

| กระบวนการทำงาน | งานวิจัยที่เกี่ยวข้อง | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G | H | I | J | K | L | M | N |
| **การประมวลผล** | | | | | | | | | | | | | | |
| ประมวลผลแบบขนาน | | | ✓ | | ✓ | ✓ | ✓ | | | | | | | |
| ประมวลผลแบบกระจาย | | | | | | ✓ | | | | ✓ | ✓ | | | ✓ |
| ใช้ซอฟต์แวร์ MPI | ✓ | ✓ | | ✓ | ✓ | ✓ | | | | | | | | |
| ใช้ซอฟต์แวร์ Spark | | | ✓ | | | | | ✓ | | ✓ | ✓ | ✓ | | ✓ |
| ใช้ RDD | | | ✓ | | | | | | | ✓ | ✓ | | | ✓ |
| ใช้ Dataset | | | | | | | | | | ✓ | ✓ | ✓ | | ✓ |
| **ตรวจแก้จุดบกพร่อง** | | | | | | | | | | | | | | |
| มีการประยุกต์ใช้การตรวจแก้จุดบกพร่อง | | | | | ✓ | ✓ | ✓ | | ✓ | | | | | |
| มีการดัดแปลง LLVM | | | | | | | ✓ | | | | | | | |
| การทดสอบซอฟต์แวร์ | | | | | | ✓ | ✓ | ✓ | ✓ | | | | | ✓ |
| การทดสอบซอฟต์แวร์โดยอาศัยจุดตรวจสอบ | | | | | | | | | | | | | ✓ | ✓ |

# บทที่ 3

# วิธีดำเนินการวิจัย

งานวิจัยนี้มีวัตถุประสงค์เพื่อประยุกต์ใช้จุดตรวจสอบเพื่อทดสอบซอฟต์แวร์ในระบบประมวลผลแบบกระจายที่มีการควบคุมคุณภาพ โดยกระบวนการทดสอบการทำงานของซอฟต์แวร์ในลักษณะคล้ายกับ Test-Driven Development กล่าวคือมีการแบ่งการทดสอบการทำงานออกเป็นกรณีย่อย ๆ ซึ่งผู้พัฒนาซอฟต์แวร์ที่จะสามารถทดสอบกรณีทดสอบที่เพิ่มเข้าในระบบหรือแก้ไขกรณีทดสอบเดิมโดยไม่มีความจำเป็นต้องประมวลผลใหม่ตั้งแต่ต้นทั้งหมด ระบบจะสามารถกู้คืนจุดตรวจสอบตำแหน่งที่เหมาะสมเพื่อลดเวลาและต้นทุนการพัฒนาซอฟต์แวร์ ในบทนี้จะกล่าวถึงกรอบแนวคิดการวิจัย วิธีการวิจัย เครื่องมือที่ใช้ในการวิจัย และกระบวนการต่าง ๆ ของการวิจัยโดยมีรายละเอียดดังนี้

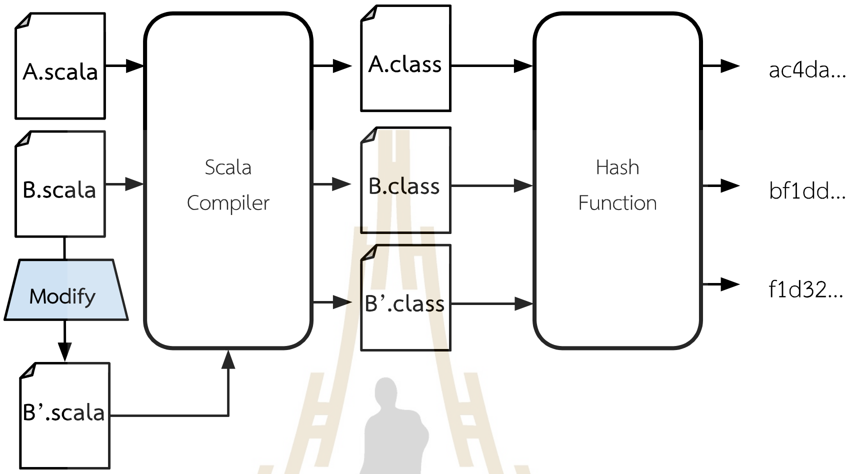
## 3.1 กรอบแนวคิดวิธีการวิจัย

เนื่องจากซอฟต์แวร์ Spark พัฒนาโดยใช้ภาษาโปรแกรม Scala ที่ต้องคอมไพล์ (Compile) ไฟล์ต้นรหัสให้เป็นรหัสไบต์ เพื่อที่จะให้อยู่ในรูปแบบที่สามารถปฏิบัติการได้บนระบบ JVM[3] เนื่องจากต้นรหัสต้องถูกคอมไพล์ทำให้เกิดไฟล์รหัสไบต์ หากพัฒนากรณีทดสอบแล้วแบ่งการทำงานเป็นกรณีย่อยโดยใช้ไฟล์ต้นรหัสแยกกันตามกรณีทดสอบและกำหนดข้อมูลนำเข้าและข้อมูลส่งออกที่เหมาะสมสำหรับแต่ละกรณีทดสอบ เมื่อระบบคอมไพล์ต้นรหัสแล้วจะได้ไฟล์รหัสไบต์อยู่คนละไฟล์กันตามที่แยกในแต่ละกรณีทดสอบ กระบวนการดังกล่าวเปิดโอกาสให้สามารถใช้

---

[3] ภาษาโปรแกรม Scala สามารถใช้งานได้กับกรอบงาน .NET แต่ประสิทธิภาพไม่ดีนักและไม่เป็นที่นิยม

ฟังก์ชันเข้ารหัสทางเดียวเพื่อตรวจสอบการเปลี่ยนแปลงของรหัสไบต์ ซึ่งหากมีการเปลี่ยนแปลงข้อมูลต้นรหัสแล้วคอมไพล์ใหม่จะได้รหัสไบต์ที่ต่างจากเดิม และเมื่อนำมาเข้าประมวลผลฟังก์ชันเข้ารหัสทางเดียวจะได้ค่าที่ต่างไปจากเดิม แม้ว่ารหัสไบต์จะเปลี่ยนเพียงเล็กน้อยก็ตาม

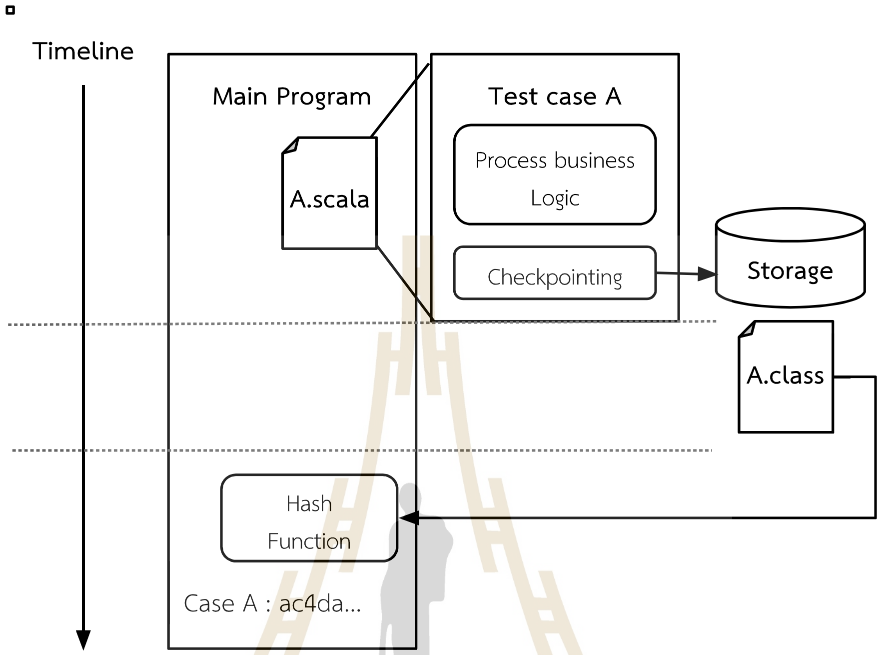


รูปที่ 3.1 แผนภาพแสดงกระบวนการตรวจสอบการแก้ไขไฟล์ต้นรหัสโดยใช้ไฟล์รหัสไบต์

จากรูปที่ 3.1 จะเห็นว่ามีไฟล์ต้นรหัส 2 ไฟล์ คือ ไฟล์ A.scala และไฟล์ B.scala เมื่อคอมไพล์แล้วจะได้ไฟล์รหัสไบต์ A.class และ B.class ตามลำดับ จากนั้นจึงประมวลผลฟังก์ชันเข้ารหัสทางเดียวจะได้รหัสที่มีความแตกต่างกันเนื่องจากไฟล์ต้นรหัสแตกต่างกัน และหากมีการแก้ไขต้นรหัสในไฟล์ B.scala ทำให้เปลี่ยนเป็นเป็นไฟล์ B’.scala แล้วนำเข้ากระบวนการคอมไพล์จะได้ไฟล์รหัสไบต์ B’.class และเมื่อประมวลผลฟังก์ชันทางเดียวก็จะพบว่ารหัสนั้นแตกต่างจากไฟล์ B เนื่องจากไฟล์มีการแก้ไข แต่ในกรณีของไฟล์ A ไม่มีการแก้ไขก็จะได้รหัสเดิม

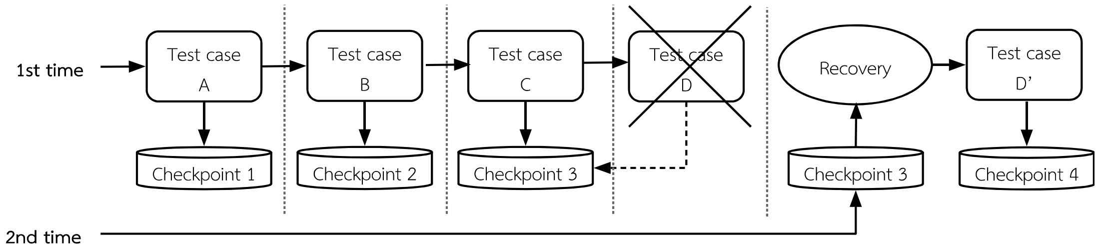
จากกระบวนการตรวจสอบความเปลี่ยนแปลงดังกล่าวข้างต้น และเมื่อแยกกรณีทดสอบออกเป็นส่วนของต้นรหัสออกเป็นลักษณะของคลาส (Class) หรือไฟล์ย่อย ๆ แล้ว ทำให้สามารถสร้างจุดตรวจสอบตามกรณีทดสอบแต่ละกรณีได้ ซึ่งกลไกการกู้คืนจุดตรวจสอบจะทำให้ไม่ต้องเริ่มประมวลผลใหม่ตั้งแต่ต้นหากมีการแก้ไขกรณีทดสอบในบางกรณี รูปที่ 3.2 ระบบทำงานอยู่บนโปรแกรมหลักจากนั้นภายในโปรแกรมหลักได้สร้างตัวแปรของกรณีทดสอบ A.scala ขึ้นมาแล้วสั่งให้กระบวนการประมวลผลภายใน A.scala เริ่มทำงานที่เป็นงานที่ต้องประมวลผลจริงเมื่อเสร็จแล้วระบบจะสร้างจุดตรวจสอบขึ้น และโปรแกรมหลักจะอ่านค่าของไฟล์ A.class ซึ่งเป็นไฟล์รหัสไบต์

ของไฟล์ A.scala จากนั้นจะเข้ารหัสทางเดียวจนกระทั่งได้ค่าเพื่อเอาไว้ตรวจสอบการเปลี่ยนแปลงของไฟล์ A.scala



รูปที่ 3.2 แผนภาพแสดงกระบวนการสร้างจุดตรวจสอบและประมวลผลไฟล์รหัสไบต์ให้เป็นค่ารหัสของฟังก์ชันเข้ารหัสทางเดียว

กรณีทดสอบจะมีได้หลายกรณีและสามารถกำหนดข้อมูลส่งออกของขั้นตอนก่อนหน้าซึ่งมักจะเป็นข้อมูลนำเข้าในขั้นตอนถัดไป หากเกิดข้อผิดพลาดหรือการแก้ไขกรณีทดสอบก็จะกระทบแค่กรณีนั้น ๆ ระบบสามารถกู้คืนจุดตรวจสอบได้จากขั้นตอนก่อนหน้าได้ทำให้ไม่ต้องประมวลผลงานซ้ำอีก ดังแสดงในรูปที่ 3.3 ระบบมีกรณีทดสอบ 4 กรณี คือ A, B, C และ D เมื่อระบบปฏิบัติการในแต่ละกรณีทดสอบเสร็จก็จะสร้างจุดตรวจสอบไว้โดยเขียนข้อมูลลงในแหล่งเก็บที่น่าเชื่อถือ และหากเกิดข้อผิดพลาดเกิดขึ้นหรือกรณีทดสอบทำงานไม่เป็นไปตามค่าที่คาดหวัง เช่น ในการปฏิบัติการครั้งแรกกรณีทดสอบ D ทำงานไม่เป็นไปตามค่าที่คาดหวังของระบบ เมื่อผู้ใช้ทำการปรับปรุงกรณีทดสอบ D แล้ว (แทนกรณีทดสอบที่ถูกแก้ไขนี้ด้วย D’) แล้วจึงทดลองปฏิบัติการอีกครั้ง จะพบว่าระบบข้ามการทำงานของกรณีทดสอบ A, B และ C ไปเนื่องจากไม่มีการแก้ไขกรณีทดสอบดังกล่าวในการปฏิบัติการครั้งที่สอง แต่ได้กู้คืนจุดตรวจสอบที่ 3 ซึ่งเป็นชุดข้อมูลหลังจากกรณี C ทำงานเสร็จสิ้น



รูปที่ 3.3 แผนภาพแสดงกระบวนการทดสอบกรณีทดสอบย่อยแล้วพบความผิดปกติ

## 3.2 การออกแบบและใช้งานจุดตรวจสอบของ Distributed Test Checkpointing (DTC)

ซอฟต์แวร์ Spark ได้บันทึกขั้นตอนการประมวลผล RDD เป็นสายตระกูลหรือที่รู้จักในชื่อแผนการประมวลผลเชิงตรรกะ (Logical Execution Plan) ซึ่งข้อมูลสายตระกูลดังกล่าวจะเก็บรวบรวมคำสั่งที่ถูกเรียกบน RDD ตัวนั้นไว้เพื่อใช้ประมวลผลเมื่อมีการเรียกใช้งานคำสั่งที่เป็นลักษณะการกระทำ อีกทั้งยังช่วยแก้ไขการประมวลผลในกรณีที่การประมวลผลเกิดข้อผิดพลาด ลักษณะการประมวลผลของ Spark โดยเมื่อคำสั่งการกระทำถูกเรียกบน RDD ใดแล้ว งานนั้นจะถูกส่งให้กับ DAG Scheduler เพื่อเปลี่ยนงานให้อยู่ในรูปของกราฟอวัฏจักรระบุทิศทาง (Directed Acyclic Graph) ซึ่งลักษณะของจุดยอดจะใช้แทนพาร์ทิชันและเส้นเชื่อมจะแทนการแปลงหลังจากนั้นแล้วระบบจะทำการแปลงการประมวลผลให้อยู่ในรูปของสเตจ (Stage) ซึ่งเป็นกลุ่มของตัวงานที่จะถูกแบ่งออกจากกันเมื่อมีลักษณะการขึ้นต่อกันแบบแคบ (Narrow Dependency) ซึ่งการประมวลผลการหาสเตจนั้นจะประมวลผลเริ่มจากจุดที่ถูกเรียกคำสั่งการกระทำย้อนกลับไปยังจุเริ่มต้นของ RDD แต่การประมวลผลข้อมูลจริงนั้นจะเริ่มประมวลผลจากจุดกำเนิด หลังจากที่ประมวลผลข้อมูลสเตจ

```
1        val data = sc.parallelize(Array(1,2,3,4,5))
2        val distData = data.map(x => (x,1))
3        distData.dtCheckpoint()
4        distData.count()
```

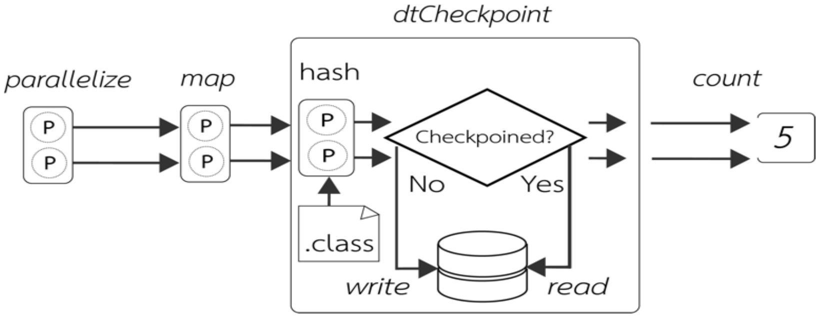
รูปที่ 3.4 แสดงตัวอย่างต้นรหัสที่มีการเรียกใช้งานกลไกสร้างจุดตรวจสอบของ DTC

จากลักษณะการประมวลผลที่กล่าวข้างต้นนั้นทำให้การเรียกใช้งานจุดตรวจสอบต้องทำการเรียกใช้งานก่อนที่จะมีการเรียกใช้งานการกระทำเกิดขึ้นกับ RDD ในรูปที่ 3.4 ที่แสดงให้เห็นการเรียกใช้งานจุดตรวจสอบของ DTC ซึ่งอธิบายได้ดังนี้

1) บรรทัดที่ 1 แสดงให้เห็นถึงการเรียกใช้งาน SparkContext ในค่าเริ่มต้นของ Spark กำหนดให้ชื่อ *sc* มีการเรียกใช้เมธอด (Method) ชื่อว่า *palallelize* ซึ่งเป็นเมธอตที่ใช้สำหรับทำให้ค่าพารามิเตอร์ (Parameter) ที่เมธอดนั้นรับค่ามาสามารถดำเนินการได้ในรูปแบบการประมวลผลแบบขนาน ในกรณีนี้คืออาเรย์ (Array) ของจำนวนเต็มบวก 1 ถึง 5
2) บรรทัดที่ 2 มีการประมวลผลข้อมูลผลลัพธ์ที่ได้จากบรรทัดที่ 1 ให้อยู่ในรูปแบบที่เป็นคู่ Key และ Value โดยจะดำเนินการกับทุกตัวข้อมูล
3) บรรทัดที่ 3 มีการเรียกใช้ *dtCheckpoint* ซึ่งเป็นเมธอดที่ใช้สำหรับการเรียกใช้งานจุดตรวจสอบในระบบ DTC จะเห็นว่าสามารถเรียกใช้ในรูปแบบเดียวกับจุดตรวจสอบดั้งเดิมของ Spark ทำให้ผู้ที่คุ้นเคยกับ Spark อยู่แล้วนั้นไม่ต้องเรียนรู้เพิ่มเติมในส่วนของการใช้งาน
4) บรรทัดที่ 4 แสดงถึงการเรียกใช้คำสั่งการกระทำ *count* ซึ่งเป็นคำสั่งที่ใช้นับจำนวนสมาชิกของข้อมูล

จากตัวอย่างจะเห็นว่าต้องมีการเรียกใช้การสร้างจุดตรวจสอบก่อนเนื่องจากกลไกการประมวล Stage ภายใน DAG Scheduler กระบวนการแปลงทั้งหมดก่อนจึงจะประมวลผลงานจริงเพื่อช่วยลดข้อผิดพลาดที่อาจจะเกิดขึ้นจากการประมวลผล

กระบวนการที่ DTC ใช้มีส่วนย่อย 2 ส่วนคือ **DtCheckpointing** ซึ่งเป็นส่วนที่ช่วยจัดการกระบวนการสร้างจุดตรวจสอบ ตรวจสอบการมีอยู่ของจุดตรวจสอบ และกู้คืนจุดตรวจสอบ ซึ่งอาศัยการทำงานรวมกับกลไก **Hashing an RDD** ซึ่งเป็นกลไกย่อยอีกส่วนหนึ่งของ DTC ที่ช่วยจัดการในการประมวลผลฟังก์ชันเข้ารหัสทางเดียวกับสายตระกูลและข้อมูลนำเข้าตามที่ผู้ใช้ได้มีการตั้งค่าเอาไว้ ซึ่งรายละเอียดจะได้อธิบายต่อไป



รูปที่ 3.5 แสดงกลไกการทำงานของ DtCheckpointing

### 3.2.1 กลไก DtCheckpointing

กลไกนี้จะถูกเรียกใช้งานเมื่อมีการเรียกเมธอด *dtCheckpoint* ซึ่งเป็นเมธอดที่อยู่ใน RDD และ DataSet (รวมถึง DataFrame ซึ่งมีลักษณะเดียวกับ DataSet[Row]) เมื่อการใช้งานถูกเรียกขึ้นจะเป็นการทำเครื่องหมายว่าจะมีการสร้างจุดตรวจสอบของ RDD หรือ DataSet นั่น ๆ ขึ้น แต่เนื่องจากกลไกการประมวลผลเกิดขึ้นหลังจากที่งานได้ถูกส่งไปให้ DAG Scheduler ประมวลผลเพื่อแยกสเตจออกจากกัน เนื่องจากการประมวลผลจริงจะเกิดขึ้นก็ต่อเมื่อมีการเรียกใช้คำสั่งการกระทำดังนั้นจึงสามารถวิเคราะห์กระบวนการทำงานก่อนที่จะประมวลผลได้ ในการสร้างเมธอด *dtCheckpoint* นี้ให้สามารถเรียกใช้งานได้โดยที่ผู้ใช้ต้องเรียนรู้น้อยที่สุด ผู้วิจัยจึงได้พัฒนากลไกของ DtCheckpointing ให้เป็นคลาสโดยปริยาย (Implicit Class) ของ RDD และ DataSet ซึ่งข้อดีคือจะทำให้หากซอฟต์แวร์ Spark ในรุ่นที่ใหม่กว่าเปิดเผยให้ใช้แล้วสามารถที่จะนำเอากรอบงาน DTC เชื่อมต่อเข้าไปกับ Spark รุ่นใหม่นั้นได้โดยตรงแทนที่การต้องแก้ไขต้นรหัสของ Spark และยังมีข้อดีคือผู้ใช้งานสามารถเรียกใช้งานเมธอดได้เสมือนว่าต้นรหัสถูกพัฒนาขึ้นมาจากผู้พัฒนาสายหลักของ Spark ซึ่งคุณสมบัตินี้มีอยู่ใน Scala รุ่น 2.10 ขึ้นไป

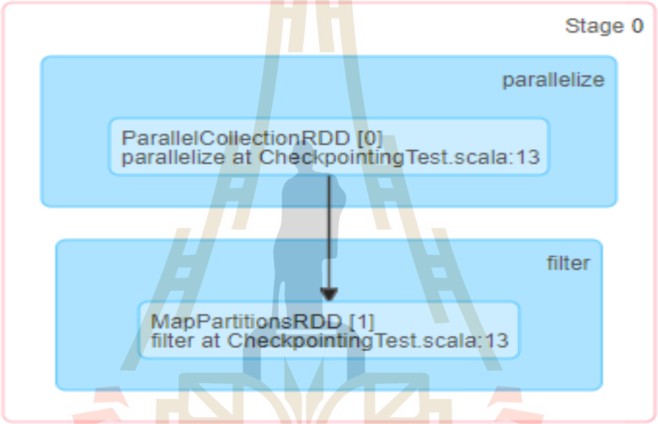
กลไกของ DtCheckpointing แสดงในรูปที่ 3.5 หากมีตัวแปรชนิด RDD หรือ DataSet ถูกเรียกใช้งานเมธอด *parallelize* และ *map* แล้วจากนั้นกลไก DtCheckpointing จะทำการเข้ารหัสทางเดียวกับพาร์ทิชันของข้อมูลที่อยู่ภายในตัวแปรในกรณีที่ผู้ใช้เลือกให้เข้ารหัสทางเดียวกับข้อมูล ซึ่งกำหนดได้ผ่านทางพารามิเตอร์ของเมธอด *dtCheckpoint* หากเรียกโดยส่งพารามิเตอร์ *dtCheckpoint(true)* แสดงว่าผู้ใช้ต้องการให้มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้าด้วย (ค่าเริ่มต้นจะไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า) เมื่อเสร็จแล้วจะเข้ารหัสทางเดียวกับรหัส

ไบต์ที่ถูกกรองข้อมูลที่ไม่ได้ใช้งานออกไป ในส่วนของการเข้ารหัสข้อมูลทางเดียวทั้งหมดเป็นความรับผิดชอบของกลไก Hashing an RDD แต่ส่วนของกลไก DtCheckpointing นั้นจะทำหน้าที่ตรวจสอบการมีอยู่ของจุดตรวจสอบต่อไป หากมีอยู่แล้วจะเรียกจุดตรวจสอบคืนกลับมาแต่หากไม่มีอยู่ก็จะสร้างจุดตรวจสอบขึ้นมาใหม่

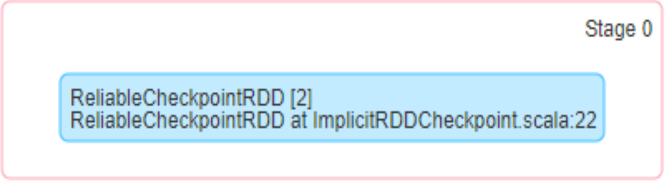
ตัวอย่างการทำงานประมวลผลจริงของกลไก DtCheckpointing ในรูปที่ 3.6 แสดงถึงต้นรหัสการคำนวณหาค่าพาย (Pi) แล้วมีการเรียกใช้งานเมธอด *dtCheckpoint* ในบรรทัดที่ 5 ส่วนในรูปที่ 3.7 แสดงกราฟอวัฏจักรระบุทิศทางที่ถูกประมวลผลครั้งแรกซึ่งจะเห็นว่ามี *parallelize* และ *filter* ถูกเรียกที่สเตจ 0 บนตัวแปร และ รูปที่ 3.8 แสดงให้เห็นกรณีที่มีการประมวลผลซ้ำงานเดิมแม้จะปิดโปรแกรมหรือทำให้ JVM หยุดทำงานไปแล้วแต่ระบบก็สามารถกู้คืนจุดตรวจสอบขึ้นมาได้ทำให้เห็นเป็นการกู้ข้อมูลคืนมาให้อยู่ในรูปตัวแปรของคลาส ReliableCheckpointRDD จากตัวอย่างต้นรหัสซึ่งจะเห็นว่ามีการคำนวณค่าพายโดยมีการสุ่มค่าเกิดขึ้นด้วยนั้น กรอบงาน DTC ยังช่วยทำให้ได้ค่าเดิมที่เคยถูกประมวลผลจนได้ผลลัพธ์ เช่น 3.1416076 ได้ค่านั้นอีกครั้ง แทนที่จะได้ค่าที่เปลี่ยนไปทุกครั้งระบบจึงสามารถทำซ้ำปัญหาซึ่งเป็นคุณสมบัติสำคัญที่ควรมีในเครื่องมือทดสอบซอฟต์แวร์ได้

```
val count =sc.parallelize(1 to (100000000*10)).filter { _ =>
  val x =math.random
  val y =math.random
  x*x +y*y < 1
}.dtCheckpoint()
 .count();
val pi =4.0 *count /(100000000*10)
```

รูปที่ 3.6 แสดงตัวอย่างต้นรหัสการประมวลผลหาค่าพาย



รูปที่ 3.7 แสดงกราฟอวัฏจักรระบุทิศทางของตัวแปรที่ถูกประมวลผลครั้งแรก



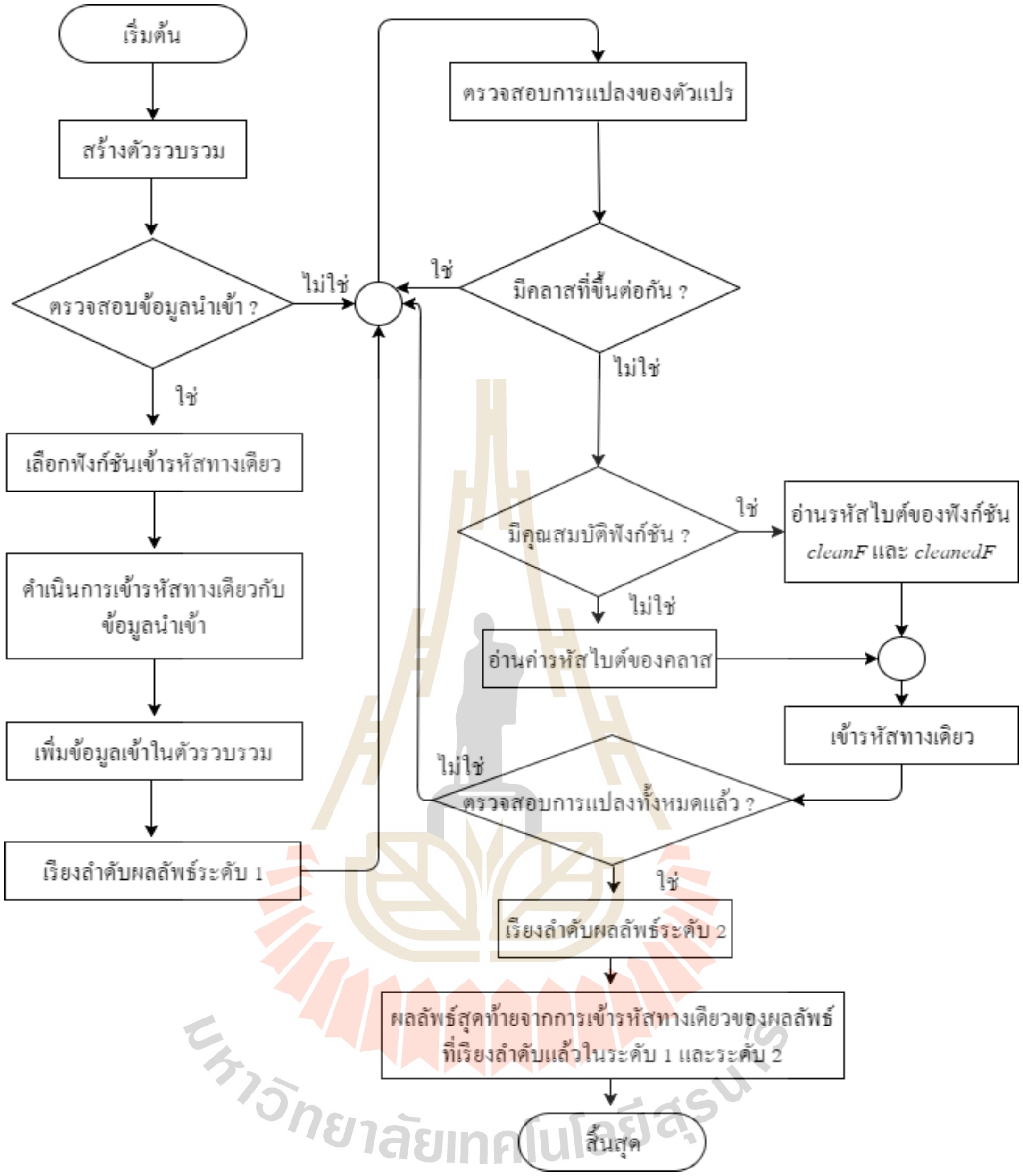
รูปที่ 3.8 แสดงกราฟอวัฏจักรระบุทิศทางของตัวแปรที่ถูกประมวลผลซ้ำ

### 3.2.1 กลไก Hashing an RDD

กลไกนี้ใช้ฟังก์ชันเข้ารหัสทางเดียวซึ่งสามารถใช้ตรวจสอบได้ว่าข้อมูลที่นำมาเข้ารหัสทางเดียวนั้นมีความเปลี่ยนแปลงเกิดขึ้นหรือไม่ได้ กรอบงาน DTC มีความสามารถที่จะเข้ารหัสทางเดียวด้วยขั้นตอนวิธี MD5, SHA-1 และ SHA-256 เนื่องจากแต่ละขั้นตอนวิธีมีความแตกต่างกันดังนั้นในวิทยานิพนธ์เล่มนี้จึงได้มีการทดสอบเปรียบเทียบกันด้วย และการทำการเข้ารหัสทางเดียวเป็นส่วนที่ทำให้กรอบงาน DTC ตรวจสอบการแปลงของตัวแปร RDD หรือ DataSet จากจุดที่เกิดการกระทำย้อนกลับไปยังจุดกำเนิดของตัวแปร

รูปที่ 3.9 แสดงกลไก Hashing an RDD โดยในขั้นตอนแรกนั้นระบบจะสร้างตัวแปรตัวรวมรวม (Accumulator) ขึ้นมาก่อนเพื่อให้สามารถใช้รวบรวมข้อมูลได้ทั่วทั้งคลัสเตอร์ของ Spark โดยชนิดของตัวแปรเป็น CollectionAccumulator ซึ่งมีชนิดทั่วไป (Generic) เป็นประเภท String ซึ่งหากปราศจากตัวรวบรวมแล้วระบบจะเข้าถึงตัวแปรได้เฉพาะใน JVM ของตัวเองเท่านั้น จากนั้นจะตรวจสอบพารามิเตอร์ของเมธอด *dtCheckpoint* ว่ามีค่าความจริงเป็นจริงหรือไม่ หากมีค่าความจริงเป็นจริงจะทำการเลือกฟังก์ชันเข้ารหัสทางเดียวที่ผู้ใช้กำหนดมาใช้และดำเนินการเข้ารหัสทางเดียวกับพาร์ทิชันของข้อมูลทุกตัวที่อยู่ใน RDD หรือ DataSet จากนั้นจะเพิ่มข้อมูลลงในตัวรวบรวม เมื่อการประมวลผลข้อมูลนำเข้าดำเนินการเรียบร้อยทุกพาร์ทิชันแล้วจะทำการเรียงลำดับข้อมูลจากน้อยไปมากในระดับที่ 1 เนื่องจากว่าข้อมูลที่ประมวลผลหาค่านั้นถูกกระจายออกไปทำบนโหนดทำงานและอาจจะเสร็จไม่พร้อมกันหากไม่เรียงลำดับก่อนจะทำให้การเข้ารหัสทางเดียวของข้อมูลในแต่ละครั้งนั้นได้ค่าไม่ตรงกันแม้จะเป็นข้อมูลชุดเดิม

กระบวนการหลังจากนั้นจะทำการตรวจสอบการแปลงของตัวแปร (หากผู้ใช้ไม่ตรวจสอบข้อมูลนำเข้าระบบจะข้ามมาทำงานที่นี่เลย) เมื่อตรวจสอบการแปลงแล้วจะได้ข้อมูลของคลาสที่มีการขึ้นต่อกันระบบจะเวียนบังเกิดไปจนถึงคลาสที่อยู่ชั้นในสุดก่อนจากนั้นจึงตรวจสอบว่ามีคุณสมบัติของฟังก์ชันอยู่หรือไม่ หากมีจะตรวจสอบว่าเป็น *cleanF* หรือ *cleanedF* ซึ่งเป็นตัวแปรที่ถูกล้าง Closure ซึ่งช่วยกรองความขึ้นต่อกันทำให้สามารถส่งฟังก์ชันไปประมวลผลบนคลัสเตอร์ได้อย่างถูกต้องและหากเป็นฟังก์ชันประเภทดังกล่าวแล้วจะมีการอ่านค่าของรหัสไบต์ขึ้นมาโดยการอ่านรหัสไบต์นั้นใช้เครื่องมือวิศวกรรมย้อนกลับ (Reverse Engineering Tool) ของกรอบงาน ASM จากนั้นจะทำการกรองข้อมูล *serialVersionUID*, *LOCALVARIABLE*, *LINENUMBER* และข้อมูลของฟังก์ชันไม่ระบุตัวตนบางส่วนออกไปแล้วจึงดำเนินการเข้ารหัสทางเดียว ส่วนในกรณีที่ไม่มีคุณสมบัติฟังก์ชันอยู่จะอ่านค่ารหัสไบต์แล้วเข้ารหัสทางเดียวเลย จากนั้นวนซ้ำดำเนินการข้างตนจนหมดแล้วจึงเรียงลำดับผลลัพธ์และเข้ารหัสทางเดียวกับผลลัพธ์ที่เรียงลำดับไว้แล้วของทั้งระดับที่ 1 และระดับที่ 2 จึงจะสิ้นสุดกระบวนการ



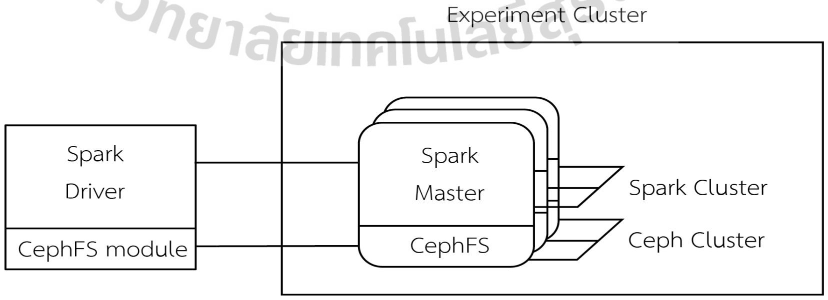
รูปที่ 3.9 แสดงผังงานของกลไก Hashing an RDD

## 3.3 เครื่องมือที่ใช้ในงานวิจัย

เครื่องมือที่ใช้ในงานวิจัยนี้ประกอบไปด้วย

1) เครื่องคอมพิวเตอร์สำหรับระบบคลัสเตอร์ 10 เครื่อง ซึ่งมีคุณสมบัติดังนี้
    - I) หน่วยประมวลผลกลาง : Intel(R) Core(TM) i5-4570 CPU @ 3.20 GHz
    - II) หน่วยความจำ : 4 กิกะไบต์
    - III) หน่วยเก็บข้อมูล : 1 เทราไบต์แบบ Hard-disk drive
2) เครื่องคอมพิวเตอร์สำหรับพัฒนาจำนวน 1 เครื่อง ซึ่งมีคุณสมบัติดังนี้
    - I) หน่วยประมวลผลกลาง : Intel(R) Xeon(R) E5-2650V3 CPU @ 2.30 GHz
    - II) หน่วยความจำ : 8 กิกะไบต์
    - III) หน่วยเก็บข้อมูล : 240 กิกะไบต์แบบ Solid-state drive
3) ระบบเชื่อมต่อเครือข่ายความสามารถผ่านข้อมูล 1 กิกะบิตต่อวินาทีต่อพอร์ต (Port)
4) ซอฟต์แวร์ที่ทำงานบนระบบคอมพิวเตอร์ รายละเอียดดังนี้
    - I) ระบบปฏิบัติการ Ubuntu 15.10
    - II) ซอฟต์แวร์ Spark
    - III) ซอฟต์แวร์ Ceph
    - IV) เครื่องมือ ASM

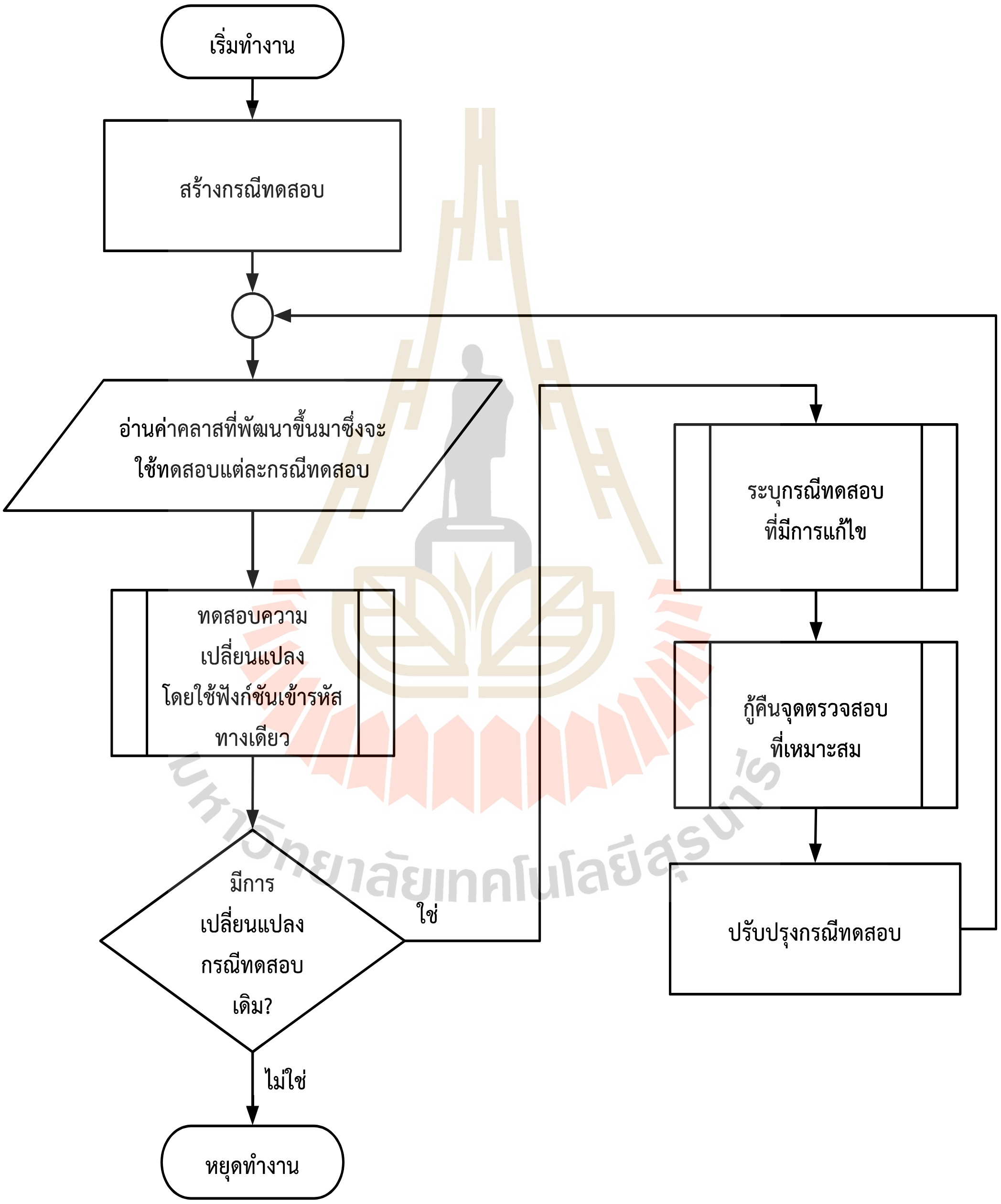
เครื่องมือที่เหล่านี้ถูกนำมาประกอบกันเป็นระบบคลัสเตอร์ของ Spark เพื่อประมวลผลและคลัสเตอร์ของ Ceph เพื่อเก็บข้อมูล จากนั้นเชื่อมต่อเครื่องคอมพิวเตอร์สำหรับพัฒนาผ่านระบบเครือข่ายไปยังคลัสเตอร์แสดงในรูปที่ 3.10



รูปที่ 3.10 ผังการเชื่อต่อกันเป็นคลัสเตอร์ของชุดทดสอบ

## 3.4 ผังงาน

ผังงานแสดงการทำงานของระบบการเพิ่มผลิตภาพของการทดสอบซอฟต์แวร์แบบกระจายโดยใช้จุดตรวจสอบที่นำเสนอในวิทยานิพนธ์ฉบับนี้แสดงในรูปที่ 3.11



รูปที่ 3.11 ผังงานแสดงการทำงานของระบบที่พัฒนาขึ้นประกอบวิทยานิพนธ์

# บทที่ 4

# ทดสอบและอภิปรายผล

ในบทนี้จะกล่าวถึงการตั้งค่าสำหรับการวัดผลและผลลัพธ์ที่เกิดขึ้นกับการทดสอบโดยใช้กรอบงาน DTC เพื่อสร้างจุดตรวจสอบรวมถึงการกู้คืนจุดตรวจสอบ ซึ่งจะได้อภิปรายถึงเวลาที่ใช้งานในแต่ละการตั้งค่า ข้อมูลที่ใช้ทดสอบ ขั้นตอนวิธีที่ใช้ทดสอบ และพื้นที่ที่ใช้ในการสร้างจุดตรวจสอบ

## 4.1 การเปรียบเทียบตัวเลือกการตั้งค่า

เมื่อกำหนดให้ No-Checkpoint คือไม่มีการใช้งานจุดตรวจสอบ, Spark Original คือเมื่อใช้จุดตรวจสอบดังเดิมของ Spark, Spark-flow คือเมื่อมีการใช้กรอบงาน Spark-flow และ DTC คือเมื่อให้กรอบงาน DTC จะแบ่งคุณสมบัติการทำงานได้ตามตารางที่ 4.1 ซึ่งจะเห็นว่า DTC นั้นไม่ต้องเพิ่มชั้นนามธรรม (Abstraction Layer) ขึ้นจากเดิม อีกทั้งยังเหมาะกับการทดสอบซอฟต์แวร์และสามารถทำงานบนคลัสเตอร์ได้ ส่วนในตารางที่ 4.2 นั้นแสดงความสามารถในการตั้งค่าที่เป็นไปได้ทั้งหมดของกรอบงานที่นำมาทดสอบเปรียบเทียบ ซึ่งจะเห็นว่า DTC นั้นมีตัวเลือกที่หลากหลายให้ผู้ใช้สามารถเลือกใช้งาน ในวิทยานิพนธ์เล่มนี้ก็จะแสดงให้เห็นการทดสอบต่อไป

ตารางที่ 4.1 แสดงคุณสมบัติที่แตกต่างกันของกรอบงานที่นำมาเปรียบเทียบ

| วิธีการ | รองรับการประมวลผลผิดพลาด | เพิ่มชั้นนามธรรม | ป้องกันการประมวลผลตั้งแต่ต้นใหม่ | เหมาะกับการทดสอบซอฟต์แวร์ | ทำงานบนคลัสเตอร์ |
|---|---|---|---|---|---|
| No-Checkpoint | - | - | - | - | ✓ |
| Spark Original | ✓ | - | ✓ | ✓(ไม่เหมาะ) | ✓ |
| Spark-flow | ✓ | ✓ | ✓ | ✓ | - |
| DTC | ✓ | - | ✓ | ✓ | ✓ |

ตารางที่ 4.2 แสดงความสามารถในการตั้งค่าของแต่ละกรอบงานที่นำมาทดสอบเปรียบเทียบ

| **วิธีการ** | **ประเภท** | | | **รูปแบบข้อมูล** | | | | **ฟังก์ชันเข้ารหัสทางเดียว** | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **RDD** | **Data-Set** | **DC** | **Java** | **Kryo** | **Avro** | **Parquet** | **MD5** | **SHA-1** | **SHA-256** |
| No-checkpoint | ✓ | ✓ | - | - | - | - | - | - | - | - |
| Spark Original | ✓ | ✓ | - | ✓ | - | - | - | - | - | - |
| Spark-flow | - | - | ✓ | - | - | - | ✓ | ✓ | - | - |
| DTC | ✓ | ✓ | - | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

## 4.2 วิธีการทดสอบ

การทดสอบจะดำเนินการโดยใช้การตั้งค่ากรณีทดสอบแบ่งเป็นกรณีดังนี้

1) กรณีทดสอบ 10 กรณีทดสอบต่อเนื่องโดยการใช้โปรแกรมนับคำที่ปรากฏมากกว่า 10 ล้านครั้งบนข้อมูล Wikipedia รหัสชุดข้อมูล enwiki-20170520-pages-logging.xml ขนาดไฟล์ 31 กิกะไบต์โดยทดสอบกับทั้ง RDD และ DataSet
2) กรณีทดสอบ 2 กรณีทดสอบต่อเนื่องจากนั้นจะปิดการทำงานของ JVM และทดสอบซ้ำ 5 ครั้งกับ 2 กรณีทดสอบเดิมเพื่อทำให้ทราบว่าระบบสามารถคงทนต่อการปิดการทำงานของโปรแกรมได้ ซึ่งการทดสอบขั้นตอนดังนี้
    I) โปรแกรมนับคำที่ปรากฏมากกว่า 10 ล้านครั้งบนข้อมูล Wikipedia รหัสชุดข้อมูล enwiki-20170520-pages-logging.xml ขนาดไฟล์ 31 กิกะไบต์โดยทดสอบกับทั้ง RDD และ DataSet
    II) โปรแกรมนับสามเหลี่ยม (Triangle count) บนข้อมูลของ Google Web Graph ขนาด 875,713 โหนดและ 5,105,039 เส้นเชื่อม ขนาดไฟล์ 73 เมกะไบต์ (Leskovec et al., 2009) โดยทดสอบกับ RDD
    III) โปรแกรม PageRank บนข้อมูลของ LiveJournal ขนาด 4,847,571 โหนดและ 68,993,773 เส้นเชื่อม ขนาดไฟล์ 1 กิกะไบต์ (Leskovec et al., 2009) โดยทดสอบกับ RDD
    IV) โปรแกรมคำนวณหาค่าพายขนาด 1,000,000,000 ครั้งโดยทดสอบกับ RDD

การทดสอบในกรณีทดสอบข้างต้นมีทั้งแบบกำหนดค่าเข้ารหัสทางเดียวกับข้อมูลนำเข้าและไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้า โดยมีการรวบรวมผลลัพธ์ของการประมวลผลทั้งด้านเวลาที่ใช้ไปและการใช้พื้นที่เก็บข้อมูล ซึ่งการวัดข้อมูลในวิทยานิพนธ์นี้ได้ตัดค่าความลำเอียง (Bias) ร้อยละ 20 ออกไปแล้ว รายละเอียดจะได้นำเสนอในหัวข้อต่อไป

## 4.3 อภิปรายผลการทดสอบ

จากที่ได้อธิบายวิธีการทดสอบในหัวข้อที่ 4.1 และหัวข้อที่ 4.2 แล้วนั้น ในหัวข้อนี้จะได้กล่าวถึงผลลัพธ์ที่ได้จากการทดสอบ

### 4.3.1 กรณีทดสอบ 10 กรณีทดสอบต่อเนื่องโดยการใช้โปรแกรมนับคำแบบ RDD

การทดสองนี้แบ่งกรณีทดสอบออกเป็น 10 กรณีทดสอบย่อยและประมวลผลต่อเนื่องกันทั้ง 10 กรณีทดสอบ โดยที่ไม่มีการปิดการทำงานของ JVM ระหว่างทดสอบโปรแกรมนับคำที่ปรากฏมากกว่า 10 ล้านครั้ง กรอบงาน DTC ได้ผลลัพธ์ในกรณีที่ไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้าจะใช้เวลาการประมวลผลในกรณีทดสอบแรกใกล้เคียงกับการประมวลผลของกลไกจุดตรวจสอบดั้งเดิมของ Spark โดยการตั้งค่าที่ทำให้ DTC ใช้เวลามากที่สุดคือ DTC-SHA-1-Java ซึ่งใช้เวลา 636 วินาทีเท่ากับการใช้จุดตรวจสอบดั้งเดิมของ Spark (แสดงในตารางที่ 4.3) และในกรณีที่ดีที่สุดคือการใช้ DTC-SHA-1-Kryo ใช้เวลาที่ 534 วินาทีซึ่งใช้เวลาน้อยกว่ากลไกจุดตรวจสอบดั้งเดิมของ Spark อยู่ 102 วินาทีคิดเป็นร้อยละ 16 ที่สามารถลดการใช้เวลาลงได้ (แสดงในตารางที่ 4.5) ส่วนในกรณี No-checkpoint ซึ่งเป็นกรณีทดสอบอ้างอิงโดยไม่ใช้งานจุดตรวจสอบนั้นใช้เวลา 136 วินาที (แสดงในตารางที่ 4.3) ดังนั้นหากเปรียบเทียบกรณีที่ใช้กรอบงาน DTC ทุกการตั้งค่าและจุดตรวจสอบดั้งเดิมของ Spark จะพบว่าใช้เวลามากกว่ากรณี No-checkpoint ถึง 4.7 เท่า แต่ก็พบว่าในกรณีทดสอบถัดมานั้นจุดตรวจสอบดั้งเดิมของ Spark ยังใช้เวลาใกล้เคียงในกรณีทดสอบแรก แต่กับกรอบงาน DTC นั้นสามารถลดการใช้เวลาได้อย่างมาก ดังแนวโน้มที่แสดงในรูปที่ 4.1

ในกรณีที่ตั้งค่าใช้ฟังก์ชันเข้ารหัสทางเดียวกับข้อมูลนำเข้าด้วยนั้นจะพบว่าในกรณีทดสอบแรกยังใช้เวลาสูงกว่ากรณี No-checkpoint และกรณีใช้จุดตรวจสอบดั้งเดิมของ Spark โดยพบว่าจุดตรวจสอบที่ใช้เวลามาที่สุดคือ DTC-SHA-1-Kryo โดยใช้เวลา 908 วินาที (แสดงในตารางที่ 4.6) และการตั้งค่าที่ใช้เวลาน้อยที่สุดคือ DTC-SHA-256-Java ใช้เวลาที่ 790 วินาที (แสดงในตารางที่ 4.3) ซึ่งใช้เวลามากกว่าจุดตรวจสอบดั้งเดิมของ Spark อยู่ร้อยละ 24 แต่ในกรณีทดสอบถัดมาก็พบว่าสามารถเวลาที่ใช้ได้เป็นอย่างมากดังแสดงในรูปที่ 4.2 นอกจากนี้ยังพบว่าหากเปรียบเทียบการใช้พื้นที่จัดเก็บข้อมูลกับจุดตรวจสอบดั้งเดิมของ Spark หากรูปแบบข้อมูลเป็นแบบ

Java สามารถลดการใช้พื้นที่เก็บข้อมูลได้ถึง 10 เท่า และหากรูปแบบข้อมูลเป็น Kryo สามารถลดพื้นที่เก็บข้อมูลได้ถึง 19.7 เท่า ดังแสดงในตารางที่ 4.7

ตารางที่ 4.3 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องในตัวแปรแบบ RDD และไม่เข้ารหัสทางเดียวข้อมูลนำเข้า โดยใช้รูปแบบของข้อมูลที่สร้างจุดตรวจสอบเป็น Java

| **กรณีทดสอบที่** | **No-Checkpoint (วินาที)** | **Spark Original (วินาที)** | **DTC (วินาที)** | | |
|---|---|---|---|---|---|
| | | | MD5 | SHA-1 | SHA-256 |
| 1 | 136 | 636 | 616 | 636 | 611 |
| 2 | 120 | 594 | 1 | 1 | 1 |
| 3 | 144 | 652 | 1 | 1 | 1 |
| 4 | 180 | 706 | 1 | 0 | 0 |
| 5 | 204 | 753 | 1 | 1 | 0 |
| 6 | 189 | 679 | 0 | 0 | 0 |
| 7 | 179 | 674 | 1 | 1 | 2 |
| 8 | 172 | 638 | 0 | 2 | 0 |
| 9 | 163 | 688 | 0 | 0 | 0 |
| 10 | 155 | 666 | 1 | 0 | 0 |

ตารางที่ 4.4 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องในตัวแปรแบบ RDD และเข้ารหัสทางเดียวข้อมูลนำเข้า โดยใช้รูปแบบของข้อมูลที่สร้างจุดตรวจสอบเป็น Java

| **กรณีทดสอบที่** | **No-Checkpoint (วินาที)** | **Spark Original (วินาที)** | **DTC (วินาที)** | | |
|---|---|---|---|---|---|
| | | | MD5 | SHA-1 | SHA-256 |
| 1 | 136 | 636 | 852 | 852 | 790 |
| 2 | 120 | 594 | 84 | 84 | 86 |
| 3 | 144 | 652 | 82 | 84 | 85 |
| 4 | 180 | 706 | 84 | 83 | 84 |
| 5 | 204 | 753 | 82 | 82 | 84 |
| 6 | 189 | 679 | 87 | 87 | 90 |
| 7 | 179 | 674 | 87 | 91 | 88 |
| 8 | 172 | 638 | 83 | 83 | 84 |
| 9 | 163 | 688 | 81 | 81 | 84 |
| 10 | 155 | 666 | 82 | 82 | 83 |

ตารางที่ 4.5 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องในตัวแปรแบบ RDD และไม่เข้ารหัสทางเดียวข้อมูลนำเข้า โดยใช้รูปแบบของข้อมูลที่สร้างจุดตรวจสอบเป็น Kryo

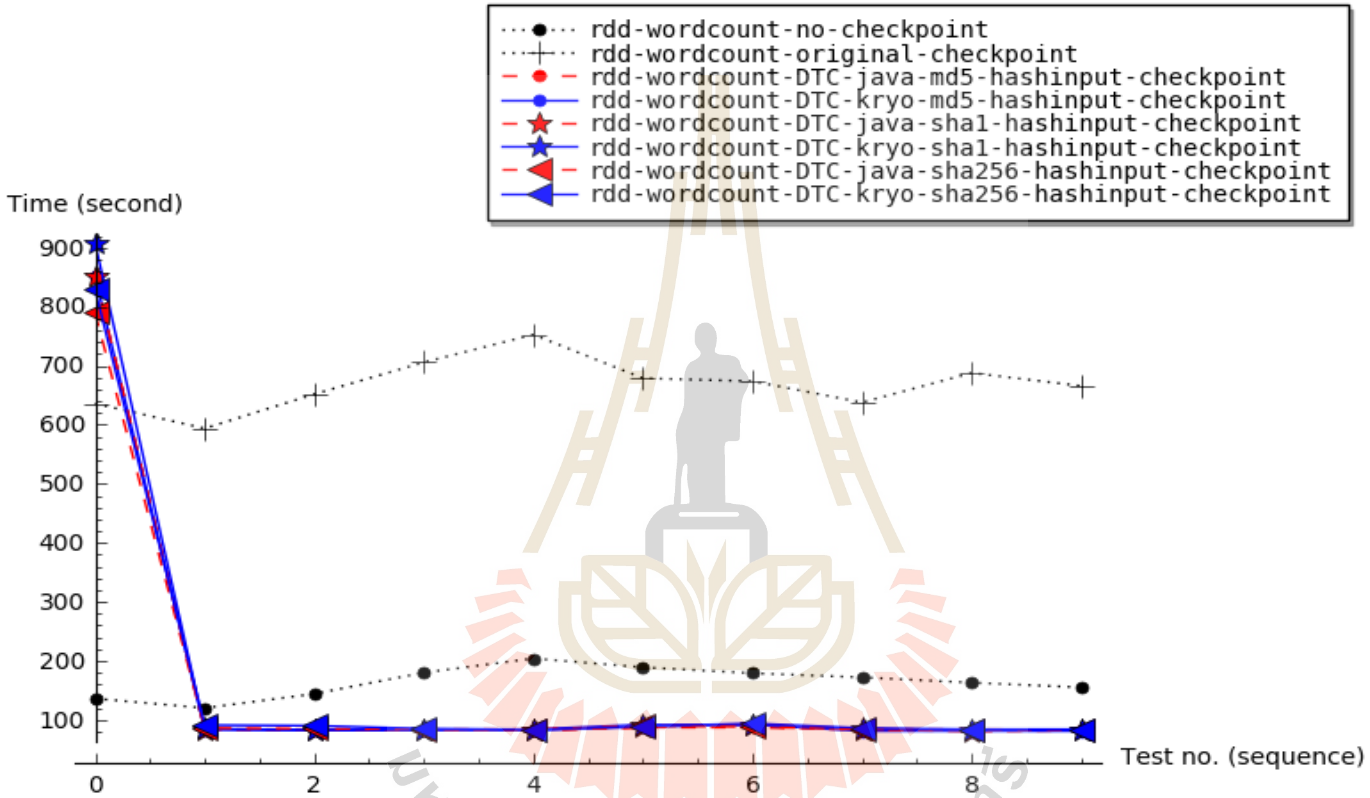
| **กรณีทดสอบที่** | **No-Checkpoint (วินาที)** | **Spark Original (วินาที)** | **DTC (วินาที)** | | |
|---|---|---|---|---|---|
| | | | MD5 | SHA-1 | SHA-256 |
| 1 | 136 | 636 | 617 | 534 | 625 |
| 2 | 120 | 594 | 1 | 1 | 1 |
| 3 | 144 | 652 | 0 | 1 | 1 |
| 4 | 180 | 706 | 0 | 1 | 1 |
| 5 | 204 | 753 | 0 | 0 | 0 |
| 6 | 189 | 679 | 2 | 0 | 0 |
| 7 | 179 | 674 | 1 | 2 | 0 |
| 8 | 172 | 638 | 1 | 0 | 3 |
| 9 | 163 | 688 | 0 | 0 | 1 |
| 10 | 155 | 666 | 0 | 1 | 2 |

ตารางที่ 4.6 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องในตัวแปรแบบ RDD และเข้ารหัสทางเดียวข้อมูลนำเข้า โดยใช้รูปแบบของข้อมูลที่สร้างจุดตรวจสอบเป็น Kryo

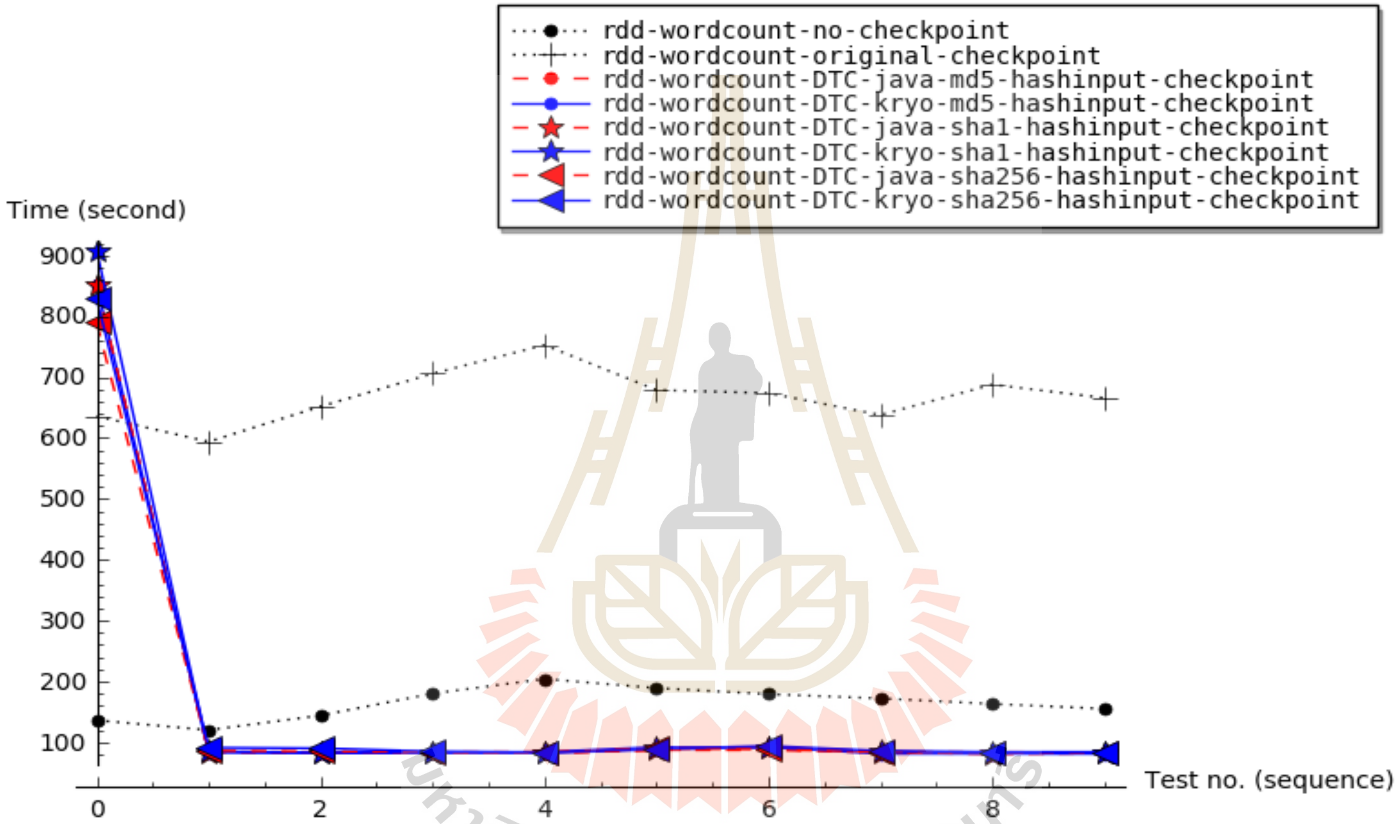
| **กรณีทดสอบที่** | **No-Checkpoint (วินาที)** | **Spark Original (วินาที)** | **DTC (วินาที)** | | |
|---|---|---|---|---|---|
| | | | MD5 | SHA-1 | SHA-256 |
| 1 | 136 | 636 | 852 | 908 | 829 |
| 2 | 120 | 594 | 83 | 85 | 91 |
| 3 | 144 | 652 | 82 | 83 | 90 |
| 4 | 180 | 706 | 82 | 84 | 85 |
| 5 | 204 | 753 | 82 | 84 | 83 |
| 6 | 189 | 679 | 88 | 92 | 89 |
| 7 | 179 | 674 | 92 | 91 | 94 |
| 8 | 172 | 638 | 81 | 85 | 86 |
| 9 | 163 | 688 | 81 | 82 | 84 |
| 10 | 155 | 666 | 81 | 82 | 84 |

ตารางที่ 4.7 แสดงการใช้พื้นที่เก็บข้อมูลจุดตรวจสอบโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องในตัวแปรแบบ RDD

| **วิธีการ** | **พื้นที่เก็บข้อมูล (เมกะไบต์)** |
|---|---|
| No-Checkopoint | 0 |
| Spark Original | 9.870 |
| DTC-MD5-Java | 0.987 |
| DTC-SHA-1-Java | 0.987 |
| DTC-SHA-256-Java | 0.987 |
| DTC-MD5-Kryo | 0.501 |
| DTC-SHA-1-Kryo | 0.501 |
| DTC-SHA-256-Kryo | 0.501 |



รูปที่ 4.1 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องกับตัวแปรแบบ RDD โดยไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้า



รูปที่ 4.2 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องกับตัวแปรแบบ RDD โดยเข้ารหัสทางเดียวกับข้อมูลนำเข้า

### 4.3.2 กรณีทดสอบ 10 กรณีทดสอบต่อเนื่องโดยการใช้โปรแกรมนับคำแบบ DataSet

การทดสองนี้แบ่งกรณีทดสอบออกเป็น 10 กรณีทดสอบย่อยและประมวลผลต่อเนื่องกันทั้ง 10 กรณีทดสอบโดยที่ไม่มีการปิดการทำงานของ JVM ระหว่างทดสอบโปรแกรมนับคำที่ปรากฏมากกว่า 10 ล้านครั้ง กรอบงาน DTC ได้ผลลัพธ์ในกรณีที่ไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้าจะใช้เวลาการประมวลผลในกรณีทดสอบแรกใกล้เคียงกับการประมวลผลของกลไกจุดตรวจสอบดั้งเดิมของ Spark โดยการตั้งค่าที่ทำให้ DTC ใช้เวลามากที่สุดคือ DTC-SHA-1-Parquet ซึ่งใช้เวลา 780 วินาทีสูงกว่าการใช้จุดตรวจสอบดั้งเดิมของ Spark ที่ใช้เวลา 701 วินาทีและกรอบงาน Spark-flow ที่ใช้เวลา 752 วินาที (แสดงในตารางที่ 4.8) และในกรณีที่ดีที่สุดคือการใช้ DTC-MD5-Parquet ใช้เวลาที่ 606 วินาทีซึ่งใช้เวลาน้อยกว่ากลไกจุดตรวจสอบดั้งเดิมของ Spark อยู่ 95 วินาทีคิดเป็นร้อยละ 14 ที่สามารถลดการใช้เวลาลงได้ และหากเทียบกับกรอบงาน Spark-flow นั้น DTC สามารถลดการใช้เวลาได้มากกว่าถึงร้อยละ 24 ส่วนในกรณี No-checkpoint ซึ่งเป็นกรณีทดสอบอ้างอิงโดยไม่ใช้งานจุดตรวจสอบนั้นใช้เวลา 134 วินาที (แสดงในตารางที่ 4.8) ดังนั้นหากเปรียบเทียบกรณีที่ใช้กรอบงาน DTC ทุกการตั้งค่าและจุดตรวจสอบดั้งเดิมของ Spark จะพบว่าใช้เวลามากกว่ากรณี No-checkpoint ถึง 4.5 เท่า แต่ก็พบว่าในกรณีทดสอบถัดมานั้นจุดตรวจสอบดั้งเดิมของ Spark และกรอบงาน Spark-flow ยังใช้เวลาใกล้เคียงในกรณีทดสอบแรก แต่กับกรอบงาน DTC นั้นสามารถลดการใช้เวลาได้อย่างมาก ดังแนวโน้มที่แสดงในรูปที่ 4.3

ในกรณีที่ตั้งค่าใช้ฟังก์ชันเข้ารหัสทางเดียวกับข้อมูลนำเข้าด้วยนั้นจะพบว่าในกรณีทดสอบแรกยังใช้เวลาสูงกว่ากรณี No-checkpoint และกรณีใช้จุดตรวจสอบดั้งเดิมของ Spark โดยพบว่าจุดตรวจสอบที่ใช้เวลามาที่สุดคือ DTC-SHA-256-Avro โดยใช้เวลา 1,052 วินาที (แสดงในตารางที่ 4.11) และการตั้งค่าที่ใช้เวลาน้อยที่สุดคือ DTC-SHA-256-Paquet ใช้เวลาที่ 995 วินาที (แสดงในตารางที่ 4.9) ซึ่งใช้เวลามากกว่าจุดตรวจสอบดั้งเดิมของ Spark อยู่ร้อยละ 42 แต่ในกรณีทดสอบถัดมาก็พบว่าสามารถเวลาที่ใช้ได้เป็นอย่างมากดังแสดงในรูปที่ 4.4 นอกจากนี้ยังพบว่าหากเปรียบเทียบการใช้พื้นที่จัดเก็บข้อมูลกับจุดตรวจสอบดั้งเดิมของ Spark หากรูปแบบข้อมูลเป็นแบบ Parquet และ Avro สามารถลดพื้นที่เก็บข้อมูลได้ถึง 10 เท่าและรูปแบบข้อมูลแบบ Avro ใช้ข้อมูลน้อยกว่าเล็กน้อย ดังแสดงในตารางที่ 4.12 จะเห็นว่ากรอบงาน Spark-flow ใช้พื้นที่เก็บข้อมูลใกล้เคียงกับกลไกสร้างจุดตรวจสอบดังเดิมของ Spark

ตารางที่ 4.8 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องในตัวแปรแบบ DataSet และไม่เข้ารหัสทางเดียวข้อมูลนำเข้า โดยใช้รูปแบบของข้อมูลที่สร้างจุดตรวจสอบเป็น Parquet

| กรณีทดสอบที่ | No-Checkpoint (วินาที) | Spark Original (วินาที) | Spark-flow (วินาที) | DTC (วินาที) | | |
|---|---|---|---|---|---|---|
| | | | | MD5 | SHA-1 | SHA-256 |
| 1 | 134 | 701 | 752 | 606 | 780 | 674 |
| 2 | 118 | 663 | 590 | 1 | 1 | 1 |
| 3 | 145 | 727 | 615 | 0 | 0 | 0 |
| 4 | 208 | 876 | 623 | 0 | 0 | 0 |
| 5 | 270 | 996 | 677 | 0 | 0 | 0 |
| 6 | 229 | 901 | 709 | 0 | 0 | 0 |
| 7 | 231 | 892 | 699 | 0 | 0 | 0 |
| 8 | 225 | 760 | 671 | 0 | 0 | 0 |
| 9 | 204 | 727 | 641 | 0 | 0 | 0 |
| 10 | 180 | 877 | 679 | 0 | 0 | 0 |

ตารางที่ 4.9 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องในตัวแปรแบบ DataSet และเข้ารหัสทางเดียวข้อมูลนำเข้า โดยใช้รูปแบบของข้อมูลที่สร้างจุดตรวจสอบเป็น Parquet

| กรณีทดสอบที่ | No-Checkpoint (วินาที) | Spark Original (วินาที) | Spark-flow (วินาที) | DTC (วินาที) | | |
|---|---|---|---|---|---|---|
| | | | | MD5 | SHA-1 | SHA-256 |
| 1 | 134 | 701 | 752 | 1004 | 1035 | 995 |
| 2 | 118 | 663 | 590 | 103 | 102 | 99 |
| 3 | 145 | 727 | 615 | 94 | 92 | 101 |
| 4 | 208 | 876 | 623 | 88 | 93 | 90 |
| 5 | 270 | 996 | 677 | 87 | 87 | 88 |
| 6 | 229 | 901 | 709 | 90 | 93 | 91 |
| 7 | 231 | 892 | 699 | 90 | 90 | 90 |
| 8 | 225 | 760 | 671 | 92 | 87 | 85 |
| 9 | 204 | 727 | 641 | 88 | 99 | 100 |
| 10 | 180 | 877 | 679 | 102 | 94 | 96 |

ตารางที่ 4.10 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องในตัวแปรแบบ DataSet และไม่เข้ารหัสทางเดียวข้อมูลนำเข้า โดยใช้รูปแบบของข้อมูลที่สร้างจุดตรวจสอบเป็น Avro

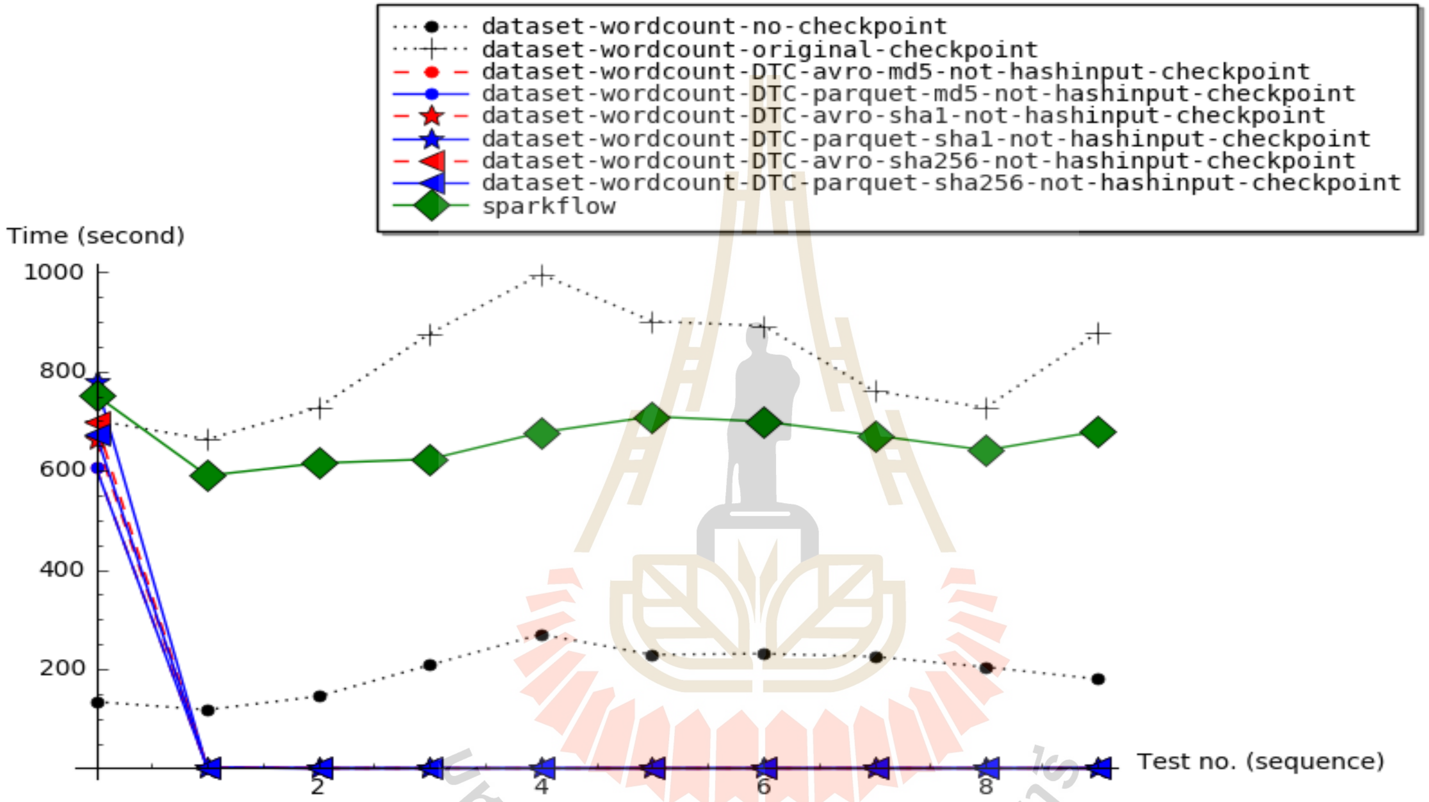
| **กรณีทดสอบที่** | **No-Checkpoint (วินาที)** | **Spark Original (วินาที)** | **Spark-flow (วินาที)** | **DTC (วินาที)** | | |
|---|---|---|---|---|---|---|
| | | | | MD5 | SHA-1 | SHA-256 |
| 1 | 134 | 701 | 752 | 607 | 663 | 699 |
| 2 | 118 | 663 | 590 | 2 | 2 | 2 |
| 3 | 145 | 727 | 615 | 0 | 0 | 0 |
| 4 | 208 | 876 | 623 | 0 | 0 | 0 |
| 5 | 270 | 996 | 677 | 0 | 0 | 0 |
| 6 | 229 | 901 | 709 | 0 | 0 | 0 |
| 7 | 231 | 892 | 699 | 0 | 0 | 0 |
| 8 | 225 | 760 | 671 | 0 | 0 | 0 |
| 9 | 204 | 727 | 641 | 0 | 0 | 0 |
| 10 | 180 | 877 | 679 | 0 | 0 | 0 |

ตารางที่ 4.11 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องในตัวแปรแบบ DataSet และเข้ารหัสทางเดียวข้อมูลนำเข้า โดยใช้รูปแบบของข้อมูลที่สร้างจุดตรวจสอบเป็น Avro

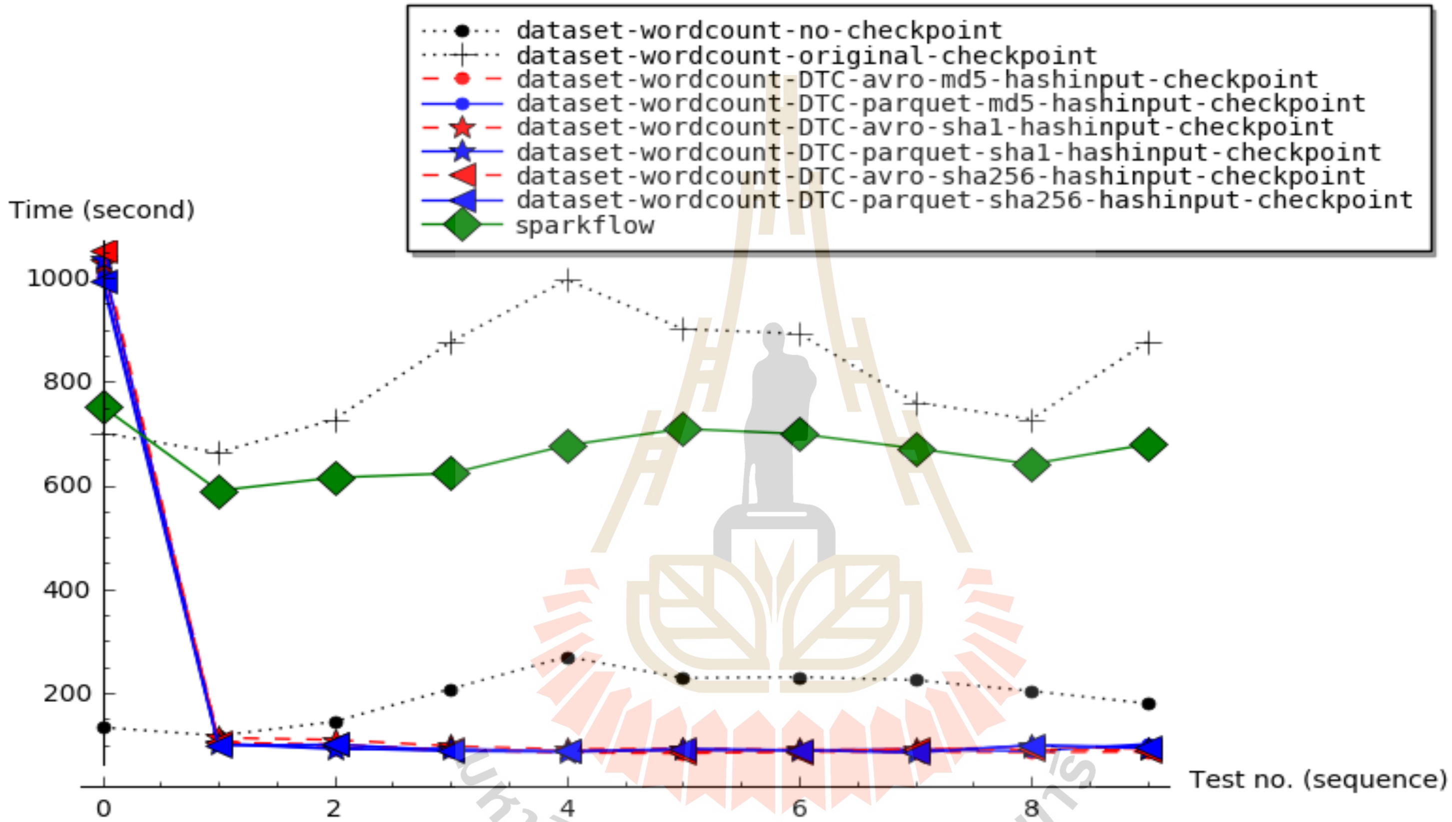
| **กรณีทดสอบที่** | **No-Checkpoint (วินาที)** | **Spark Original (วินาที)** | **Spark-flow (วินาที)** | **DTC (วินาที)** | | |
|---|---|---|---|---|---|---|
| | | | | MD5 | SHA-1 | SHA-256 |
| 1 | 134 | 701 | 752 | 1003 | 1028 | 1052 |
| 2 | 118 | 663 | 590 | 108 | 114 | 105 |
| 3 | 145 | 727 | 615 | 94 | 110 | 101 |
| 4 | 208 | 876 | 623 | 90 | 98 | 91 |
| 5 | 270 | 996 | 677 | 85 | 92 | 89 |
| 6 | 229 | 901 | 709 | 84 | 93 | 87 |
| 7 | 231 | 892 | 699 | 87 | 91 | 87 |
| 8 | 225 | 760 | 671 | 87 | 91 | 93 |
| 9 | 204 | 727 | 641 | 86 | 97 | 91 |
| 10 | 180 | 877 | 679 | 87 | 93 | 90 |

ตารางที่ 4.12 แสดงการใช้พื้นที่เก็บข้อมูลจุดตรวจสอบโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องในตัวแปรแบบ DataSet

| **วิธีการ** | **พื้นที่เก็บข้อมูล (เมกะไบต์)** |
|---|---|
| No-Checkopoint | 0 |
| Spark Original | 9.860 |
| Spark-flow | 9.930 |
| DTC-MD5-Parquet | 0.993 |
| DTC-SHA-1-Parquet | 0.993 |
| DTC-SHA-256-Parquet | 0.993 |
| DTC-MD5-Avro | 0.987 |
| DTC-SHA-1-Avro | 0.987 |
| DTC-SHA-256-Avro | 0.987 |



รูปที่ 4.3 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมนับคำ 10 กรณีทดสอบ ต่อเนื่องกับตัวแปรแบบ DataSet โดยไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้า



รูปที่ 4.4 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมนับคำ 10 กรณีทดสอบต่อเนื่องกับตัวแปรแบบ DataSet โดยเข้ารหัสทางเดียวกับข้อมูลนำเข้า

### 4.3.3 กรณีทดสอบ 2 กรณีทดสอบต่อเนื่องแล้วปิดการทำงานของ JVM โดยการใช้โปรแกรมนับคำแบบ RDD

การทดสองนี้แบ่งกรณีทดสอบออกเป็น 2 กรณีทดสอบต่อเนื่องกันหลังจากนั้นแล้วจะปิดการทำงานของ JVM แล้วจึงเริ่มทดสอบซ้ำจำนวน 5 ครั้งเพื่อตรวจสอบการทำงานในกรณีที่มีการปิดโปรแกรม โดยจะใช้โปรแกรมนับคำที่ปรากฏมากกว่า 10 ล้านครั้งโดยใช้ตัวแปรแบบ RDD ทดสอบเปรียบเทียบการตั้งค่า No-checkpoint กลไกสร้างจุดตรวจสอบดั้งเดิมของ Spark และกรอบงาน DTC ที่ตั้งค่าแบบต่าง ๆ ผลของการใช้เวลากรณีไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าแสดงในตารางที่ 4.13 และกรณีเข้ารหัสทางเดียวกับข้อมูลนำเข้าแสดงในตารางที่ 4.14 ซึ่งจะสังเกตแนวโน้มได้ตามรูปที่ 4.5 และรูปที่ 4.6 ซึ่งเป็นแผนภาพของการไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าและเข้ารหัสทางเดียวกับข้อมูลนำเข้าตามลำดับ

หากสังเกตผลลัพธ์ก็จะทราบแนวโน้มที่เกิดขึ้น กล่าวคือในกรณีที่การทดสอบใช้กรอบงาน DTC จะสามารถลดเวลาในการประมวลผลในรอบถัดมาได้อย่างมากโดยสามารถทำงานได้ดีในกรณีถัดมาแม้ว่าจะเคยปิดการทำงานขของ JVM ไปแล้ว และกรอบงาน DTC ยังใช้พื้นที่ในการเก็บตรวจสอบน้อยกว่าจุดตรวจสอบดั้งเดิมของ Spark สูงสุดถึง 19.7 เท่าเช่นเดียวกับการประมวลผล 10 กรณีทดสอบต่อเนื่องดังแสดงในตารางที่ 4.15

ตารางที่ 4.13 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD และไม่เข้ารหัสทางเดียวข้อมูลนำเข้า

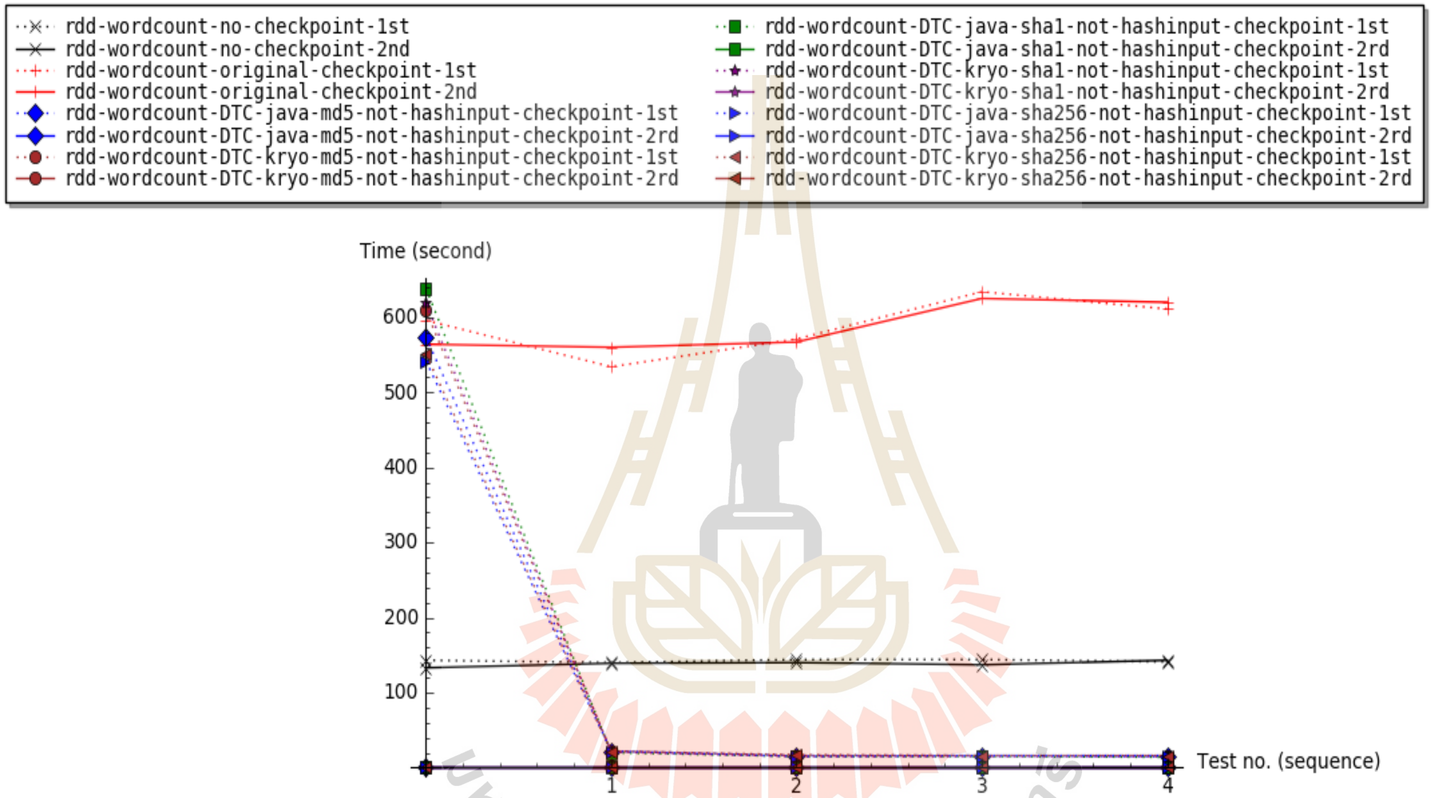
| **การทดสอบ** | **ครั้งที่ 1 (วินาที)** | **ครั้งที่ 2 (วินาที)** | **ครั้งที่ 3 (วินาที)** | **ครั้งที่ 4 (วินาที)** | **ครั้งที่ 5 (วินาที)** |
|---|---|---|---|---|---|
| No-checkpoint | | | | | |
| กรณีทดสอบ 1 | 143 | 140 | 144 | 144 | 141 |
| กรณีทดสอบ 2 | 133 | 139 | 140 | 137 | 143 |
| Spark Original | | | | | |
| กรณีทดสอบ 1 | 596 | 534 | 571 | 634 | 611 |
| กรณีทดสอบ 2 | 564 | 560 | 567 | 625 | 620 |
| DTC-MD5-Java | | | | | |
| กรณีทดสอบ 1 | 574 | 22 | 16 | 16 | 15 |
| กรณีทดสอบ 2 | 1 | 1 | 1 | 1 | 1 |
| DTC-MD5-Kryo | | | | | |
| กรณีทดสอบ 1 | 609 | 22 | 16 | 16 | 16 |
| กรณีทดสอบ 2 | 1 | 1 | 1 | 1 | 1 |
| DTC-SHA-1-Java | | | | | |
| กรณีทดสอบ 1 | 639 | 20 | 15 | 15 | 14 |
| กรณีทดสอบ 2 | 1 | 1 | 1 | 1 | 1 |
| DTC-SHA-1-Kryo | | | | | |
| กรณีทดสอบ 1 | 626 | 21 | 15 | 15 | 15 |
| กรณีทดสอบ 2 | 1 | 1 | 1 | 1 | 1 |
| DTC-SHA-256-Java | | | | | |
| กรณีทดสอบ 1 | 542 | 22 | 15 | 16 | 16 |
| กรณีทดสอบ 2 | 1 | 1 | 1 | 1 | 1 |
| DTC-SHA-256-Kryo | | | | | |
| กรณีทดสอบ 1 | 551 | 22 | 17 | 16 | 16 |
| กรณีทดสอบ 2 | 1 | 1 | 1 | 1 | 1 |

ตารางที่ 4.14 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD และเข้ารหัสทางเดียวข้อมูลนำเข้า

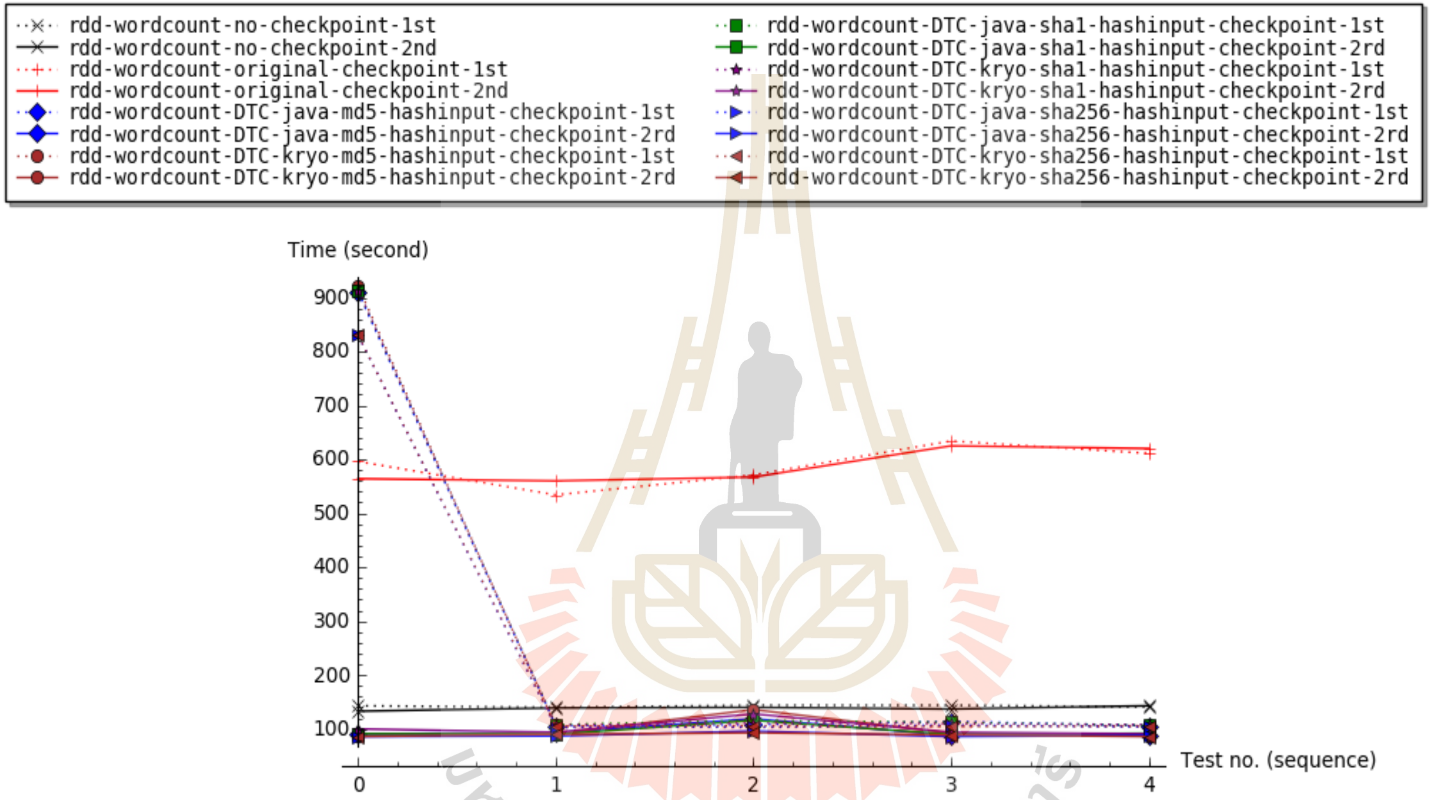
| **การทดสอบ** | **ครั้งที่ 1 (วินาที)** | **ครั้งที่ 2 (วินาที)** | **ครั้งที่ 3 (วินาที)** | **ครั้งที่ 4 (วินาที)** | **ครั้งที่ 5 (วินาที)** |
|---|---|---|---|---|---|
| No-checkpoint | | | | | |
| กรณีทดสอบ 1 | 143 | 140 | 144 | 144 | 141 |
| กรณีทดสอบ 2 | 133 | 139 | 140 | 137 | 143 |
| Spark Original | | | | | |
| กรณีทดสอบ 1 | 596 | 534 | 571 | 634 | 611 |
| กรณีทดสอบ 2 | 564 | 560 | 567 | 625 | 620 |
| DTC-MD5-Java | | | | | |
| กรณีทดสอบ 1 | 910 | 106 | 108 | 114 | 106 |
| กรณีทดสอบ 2 | 91 | 93 | 119 | 88 | 88 |
| DTC-MD5-Kryo | | | | | |
| กรณีทดสอบ 1 | 921 | 105 | 103 | 105 | 106 |
| กรณีทดสอบ 2 | 91 | 91 | 136 | 92 | 89 |
| DTC-SHA-1-Java | | | | | |
| กรณีทดสอบ 1 | 913 | 108 | 116 | 112 | 107 |
| กรณีทดสอบ 2 | 91 | 90 | 116 | 90 | 92 |
| DTC-SHA-1-Kryo | | | | | |
| กรณีทดสอบ 1 | 914 | 107 | 111 | 110 | 108 |
| กรณีทดสอบ 2 | 100 | 94 | 128 | 94 | 91 |
| DTC-SHA-256-Java | | | | | |
| กรณีทดสอบ 1 | 830 | 103 | 104 | 107 | 104 |
| กรณีทดสอบ 2 | 86 | 87 | 96 | 86 | 88 |
| DTC-SHA-256-Kryo | | | | | |
| กรณีทดสอบ 1 | 830 | 105 | 106 | 105 | 105 |
| กรณีทดสอบ 2 | 86 | 90 | 93 | 89 | 86 |

ตารางที่ 4.15 แสดงการใช้พื้นที่เก็บข้อมูลจุดตรวจสอบโปรแกรมนับคำ 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD

| **วิธีการ** | **พื้นที่เก็บข้อมูล (เมกะไบต์)** |
|---|---|
| No-Checkopoint | 0 |
| Spark Original | 9.870 |
| DTC-MD5-Java | 0.987 |
| DTC-SHA-1-Java | 0.987 |
| DTC-SHA-256-Java | 0.987 |
| DTC-MD5-Kryo | 0.501 |
| DTC-SHA-1-Kryo | 0.501 |
| DTC-SHA-256-Kryo | 0.501 |



รูปที่ 4.5 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมนับคำ 2 กรณีทดสอบต่อเนื่อง ที่มีการปิด JVM กับตัวแปรแบบ RDD โดยไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้า



รูปที่ 4.6 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมนับคำ 2 กรณีทดสอบต่อเนื่อง ที่มีการปิด JVM กับตัวแปรแบบ RDD โดยเข้ารหัสทางเดียวกับข้อมูลนำเข้า

### 4.3.4 กรณีทดสอบ 2 กรณีทดสอบต่อเนื่องแล้วปิดการทำงานของ JVM โดยการใช้โปรแกรมนับคำแบบ DataSet

การทดสองนี้แบ่งกรณีทดสอบออกเป็น 2 กรณีทดสอบต่อเนื่องกันหลังจากนั้นแล้วจะปิดการทำงานของ JVM แล้วจึงเริ่มทดสอบซ้ำจำนวน 5 ครั้งเพื่อตรวจสอบการทำงานในกรณีที่มีการปิดโปรแกรม โดยจะใช้โปรแกรมนับคำที่ปรากฏมากกว่า 10 ล้านครั้งโดยใช้ตัวแปรแบบ DataSet ทดสอบเปรียบเทียบการตั้งค่า No-checkpoint กลไกสร้างจุดตรวจสอบดั้งเดิมของ Spark และกรอบงาน DTC ที่ตั้งค่าแบบต่าง ๆ ผลของการใช้เวลากรณีไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าแสดงในตารางที่ 4.16 และกรณีเข้ารหัสทางเดียวกับข้อมูลนำเข้าแสดงในตารางที่ 4.17 ซึ่งจะสังเกตแนวโน้มได้ตามรูปที่ 4.7 และรูปที่ 4.8 ซึ่งเป็นแผนภาพของการไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าและเข้ารหัสทางเดียวกับข้อมูลนำเข้าตามลำดับ

หากสังเกตผลลัพธ์ก็จะทราบแนวโน้มที่เกิดขึ้น กล่าวคือในกรณีที่การทดสอบใช้กรอบงาน DTC จะสามารถลดเวลาในการประมวลผลในรอบถัดมาได้อย่างมากโดยสามารถทำงานได้ดีในกรณีถัดมาแม้ว่าจะเคยปิดการทำงานขของ JVM ไปแล้ว และกรอบงาน DTC ยังใช้พื้นที่ในการเก็บตรวจสอบน้อยกว่าจุดตรวจสอบดั้งเดิมของ Spark สูงสุดถึง 10 เท่า จากตารางที่ 4.18 จะเห็นว่ากรอบงาน Spark-flow ประหยัดพื้นที่ได้ในระดับหนึ่งเท่านั้นและยังไม่มีกลไกที่จะทราบว่ากรณีทดสอบถัดมามีการเรียกใช้ตัวแปร DataSet เดียวกันทำให้ระบบประมวลผลซ้ำทั้งที่มีการสร้างจุดตรวจสอบไว้แล้ว แต่อย่างไรก็ตามกรอบงาน Spark-flow สามารถตรวจสอบได้หากกรณีทดสอบนั้นถูกประมวลผลใหม่หลังจากที่เคยปิด JVM ไปแล้วดังแสดงในตารางที่ 4.16 และตารางที่ 4.17

ตารางที่ 4.16 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ DataSet และไม่เข้ารหัสทางเดียวข้อมูลนำเข้า

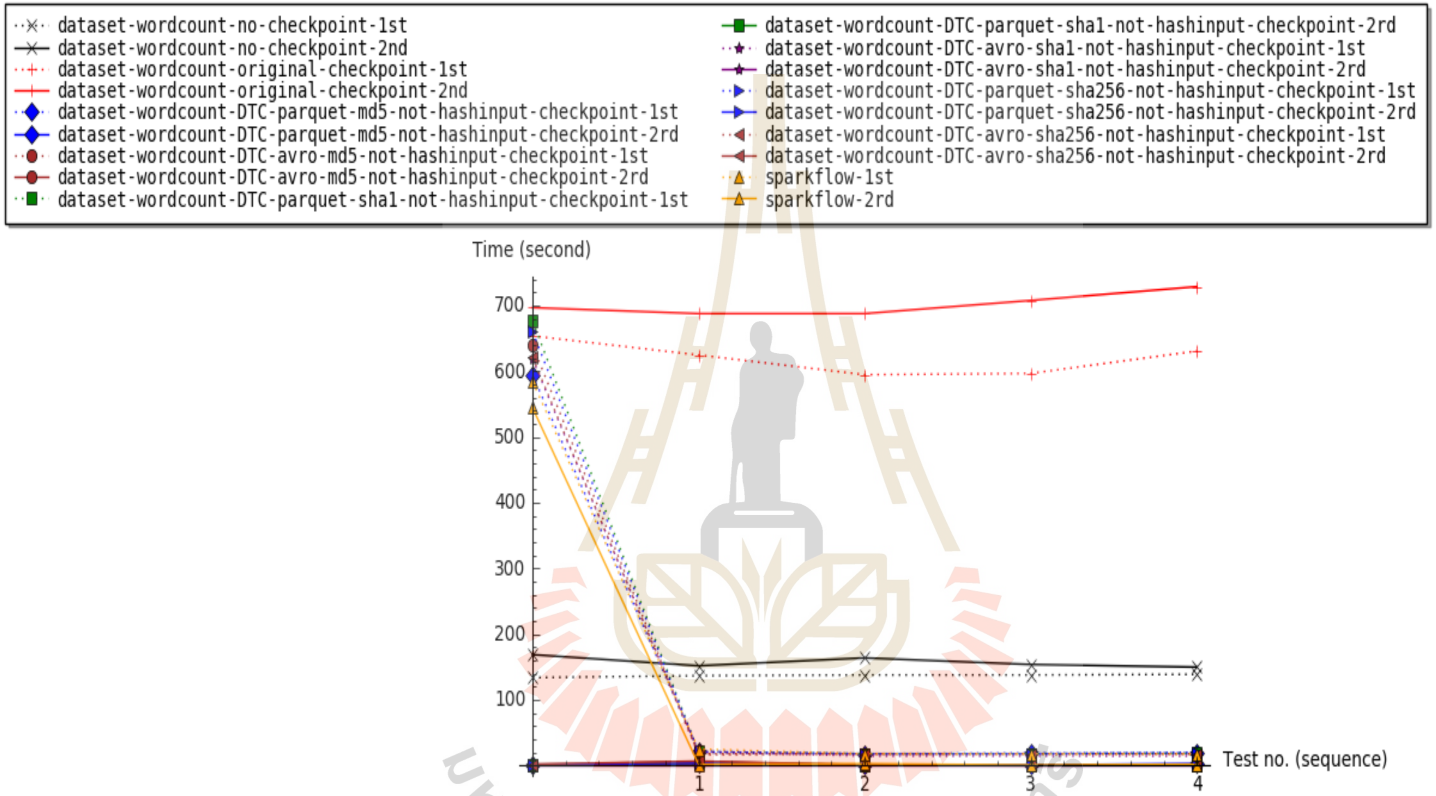
| **การทดสอบ** | **ครั้งที่ 1 (วินาที)** | **ครั้งที่ 2 (วินาที)** | **ครั้งที่ 3 (วินาที)** | **ครั้งที่ 4 (วินาที)** | **ครั้งที่ 5 (วินาที)** |
|---|---|---|---|---|---|
| No-checkpoint | | | | | |
| กรณีทดสอบ 1 | 134 | 137 | 138 | 138 | 139 |
| กรณีทดสอบ 2 | 169 | 152 | 164 | 154 | 150 |
| Spark Original | | | | | |
| กรณีทดสอบ 1 | 654 | 625 | 595 | 597 | 631 |
| กรณีทดสอบ 2 | 697 | 688 | 688 | 708 | 729 |
| DTC-MD5-Parquet | | | | | |
| กรณีทดสอบ 1 | 595 | 22 | 18 | 19 | 19 |
| กรณีทดสอบ 2 | 1 | 3 | 1 | 1 | 3 |
| DTC-MD5-Avro | | | | | |
| กรณีทดสอบ 1 | 640 | 18 | 18 | 18 | 18 |
| กรณีทดสอบ 2 | 2 | 6 | 1 | 1 | 1 |
| DTC-SHA-1-Parquet | | | | | |
| กรณีทดสอบ 1 | 677 | 21 | 18 | 18 | 19 |
| กรณีทดสอบ 2 | 1 | 4 | 1 | 1 | 1 |
| DTC-SHA-1-Avro | | | | | |
| กรณีทดสอบ 1 | 674 | 18 | 19 | 18 | 18 |
| กรณีทดสอบ 2 | 2 | 6 | 1 | 1 | 1 |
| DTC-SHA-256-Parquet | | | | | |
| กรณีทดสอบ 1 | 661 | 21 | 18 | 18 | 20 |
| กรณีทดสอบ 2 | 1 | 4 | 1 | 1 | 2 |
| DTC-SHA-256-Avro | | | | | |
| กรณีทดสอบ 1 | 622 | 17 | 18 | 18 | 18 |
| กรณีทดสอบ 2 | 2 | 7 | 1 | 0 | 1 |
| Spark-flow | | | | | |
| กรณีทดสอบ 1 | 585 | 25 | 18 | 18 | 18 |
| กรณีทดสอบ 2 | 545 | 2 | 3 | 1 | 2 |

ตารางที่ 4.17 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับคำ 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ DataSet และเข้ารหัสทางเดียวข้อมูลนำเข้า

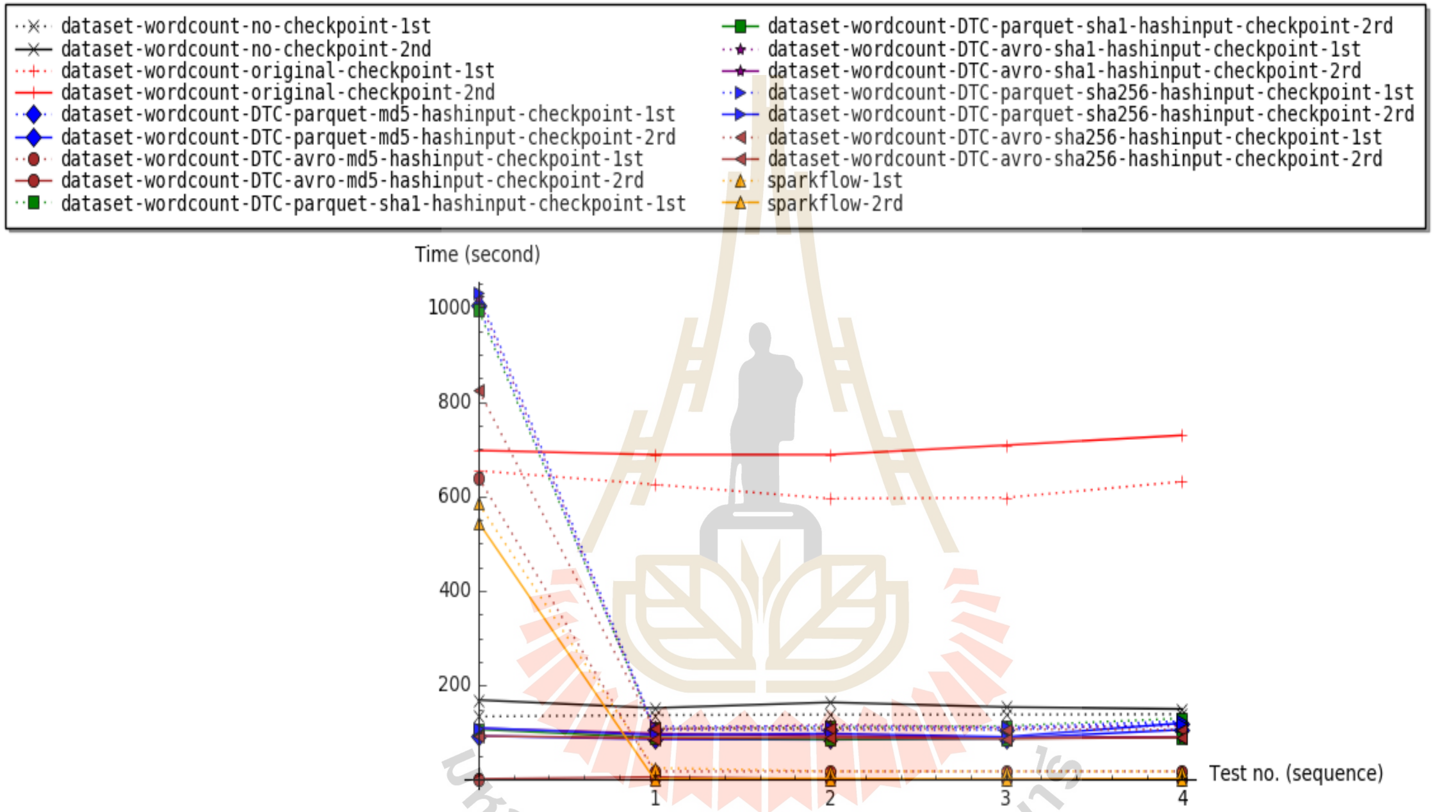
| **การทดสอบ** | **ครั้งที่ 1 (วินาที)** | **ครั้งที่ 2 (วินาที)** | **ครั้งที่ 3 (วินาที)** | **ครั้งที่ 4 (วินาที)** | **ครั้งที่ 5 (วินาที)** |
|---|---|---|---|---|---|
| No-checkpoint | | | | | |
| กรณีทดสอบ 1 | 134 | 137 | 138 | 138 | 139 |
| กรณีทดสอบ 2 | 169 | 152 | 164 | 154 | 150 |
| Spark Original | | | | | |
| กรณีทดสอบ 1 | 654 | 625 | 595 | 597 | 631 |
| กรณีทดสอบ 2 | 697 | 688 | 688 | 708 | 729 |
| DTC-MD5-Parquet | | | | | |
| กรณีทดสอบ 1 | 1003 | 106 | 106 | 106 | 119 |
| กรณีทดสอบ 2 | 93 | 86 | 85 | 88 | 106 |
| DTC-MD5-Avro | | | | | |
| กรณีทดสอบ 1 | 1029 | 108 | 108 | 108 | 108 |
| กรณีทดสอบ 2 | 95 | 89 | 92 | 91 | 87 |
| DTC-SHA-1-Parquet | | | | | |
| กรณีทดสอบ 1 | 994 | 108 | 111 | 114 | 130 |
| กรณีทดสอบ 2 | 107 | 89 | 86 | 88 | 90 |
| DTC-SHA-1-Avro | | | | | |
| กรณีทดสอบ 1 | 1021 | 107 | 117 | 111 | 114 |
| กรณีทดสอบ 2 | 109 | 98 | 93 | 89 | 91 |
| DTC-SHA-256-Parquet | | | | | |
| กรณีทดสอบ 1 | 1032 | 113 | 113 | 111 | 124 |
| กรณีทดสอบ 2 | 110 | 95 | 98 | 91 | 120 |
| DTC-SHA-256-Avro | | | | | |
| กรณีทดสอบ 1 | 825 | 108 | 108 | 107 | 106 |
| กรณีทดสอบ 2 | 93 | 88 | 90 | 87 | 90 |
| Spark-flow | | | | | |
| กรณีทดสอบ 1 | 585 | 25 | 18 | 18 | 18 |
| กรณีทดสอบ 2 | 545 | 2 | 3 | 1 | 2 |

ตารางที่ 4.18 แสดงการใช้พื้นที่เก็บข้อมูลจุดตรวจสอบโปรแกรมนับคำ 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ DataSet

| **วิธีการ** | **พื้นที่เก็บข้อมูล (เมกะไบต์)** |
|---|---|
| No-Checkopoint | 0 |
| Spark Original | 9.860 |
| Spark-flow | 1.986 |
| DTC-MD5-Java | 0.987 |
| DTC-SHA-1-Java | 0.987 |
| DTC-SHA-256-Java | 0.987 |
| DTC-MD5-Kryo | 0.501 |
| DTC-SHA-1-Kryo | 0.501 |
| DTC-SHA-256-Kryo | 0.501 |



รูปที่ 4.7 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมนับคำ 2 กรณีทดสอบต่อเนื่อง ที่มีการปิด JVM กับตัวแปรแบบ DataSet โดยไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้า



รูปที่ 4.8 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมนับคำ 2 กรณีทดสอบต่อเนื่อง ที่มีการปิด JVM กับตัวแปรแบบ DataSet โดยเข้ารหัสทางเดียวกับข้อมูลนำเข้า

### 4.3.5 กรณีทดสอบ 2 กรณีทดสอบต่อเนื่องแล้วปิดการทำงานของ JVM โดยการใช้โปรแกรมนับสามเหลี่ยมแบบ RDD

การทดสองนี้แบ่งกรณีทดสอบออกเป็น 2 กรณีทดสอบต่อเนื่องกันหลังจากนั้นแล้วจะปิดการทำงานของ JVM แล้วจึงเริ่มทดสอบซ้ำจำนวน 5 ครั้งเพื่อตรวจสอบการทำงานในกรณีที่มีการปิดโปรแกรม โดยจะใช้โปรแกรมนับสามเหลี่ยมบนข้อมูลของ Google Web Graph ขนาด 875,713 โหนดและ 5,105,039 เส้นเชื่อมโดยทดสอบกับตัวแปรแบบ RDD ทดสอบเปรียบเทียบการตั้งค่า No-checkpoint กลไกสร้างจุดตรวจสอบดั้งเดิมของ Spark และกรอบงาน DTC ที่ตั้งค่าแบบต่าง ๆ ผลของการใช้เวลากรณีไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าแสดงในตารางที่ 4.19 และกรณีเข้ารหัสทางเดียวกับข้อมูลนำเข้าแสดงในตารางที่ 4.20 ซึ่งจะสังเกตแนวโน้มได้ตามรูปที่ 4.9 และรูปที่ 4.10 ซึ่งเป็นแผนภาพของการไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าและเข้ารหัสทางเดียวกับข้อมูลนำเข้าตามลำดับ

จากตารางเวลาผลลัพธ์ที่ได้ออกมานั้นหากเป็นกรณีที่ไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้าจะสังเกตเห็นว่าสามารถลดเวลาที่ใช้ในการประมวลผลกรณีทดสอบลงได้สูงสุดถึง 7.3 เท่าเมื่อเทียบกับกรณีทดสอบแรกในครั้งแรกดังแสดงในตารางที่ 4.19 ที่มีการใช้กรอบงาน DTC ตั้งค่าเข้ารหัสทางเดียวด้วยขั้นตอนวิธีแบบ MD5 หรือ SHA-1 และใช้รูปแบบข้อมูล Java ซึ่งสอดคล้องกับผลลัพธ์ที่ได้อภิปรายในหัวข้อก่อนหน้า แต่ในส่วนกรณีที่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้านั้นจะพบว่าใช้เวลาใกล้เคียงกันทั้ง No-checkpoint ซึ่งไม่ใช้งานจุดตรวจสอบ จุดตรวจสอบดั้งเดิมของ Spark และกรอบงาน DTC ทุกการตั้งค่า จากการวิเคราะห์พบว่าเมื่อมีการอ่านพาร์ทิชันของข้อมูลขึ้นมากรอบงาน Spark จะพยายามแปลงข้อมูลให้อยู่ในรูปของวัตถุของคลาส ซึ่งในกรณีนี้คือคลาส ShippableVertexPartition ซึ่งอยู่ในแพคเกจ (Package) org.apache.spark.graphx.impl ทำให้กรอบงาน DTC ไม่สามารถตรวจสอบข้อมูลภายในได้เนื่องจากวัตถุจะถูกสร้างบนหน่วยความจำใหม่ทุกครั้ง

หากเปรียบเทียบพื้นที่เก็บข้อมูลที่ใช้ในจุดตรวจสอบในกรณีที่ใช้จุดตรวจสอบดั้งเดิมของ Spark กับกรอบงาน DTC กรณีที่เข้ารหัสข้อมูลนำเข้าโดยใช้รูปแบบข้อมูลแบบ Java นั้นพบว่าสามารถพื้นที่ได้ถึงร้อยละ 42 และในกรณีที่ดีที่สุดของกรอบงาน DTC คือตั้งค่ารูปแบบข้อมูลแบบ Kryo และไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าสามารถลดการใช้พื้นที่ได้ถึง 50 เท่า ดังแสดงในตารางที่ 4.21

ตารางที่ 4.19 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับสามเหลี่ยม 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD และไม่เข้ารหัสทางเดียวข้อมูลนำเข้า

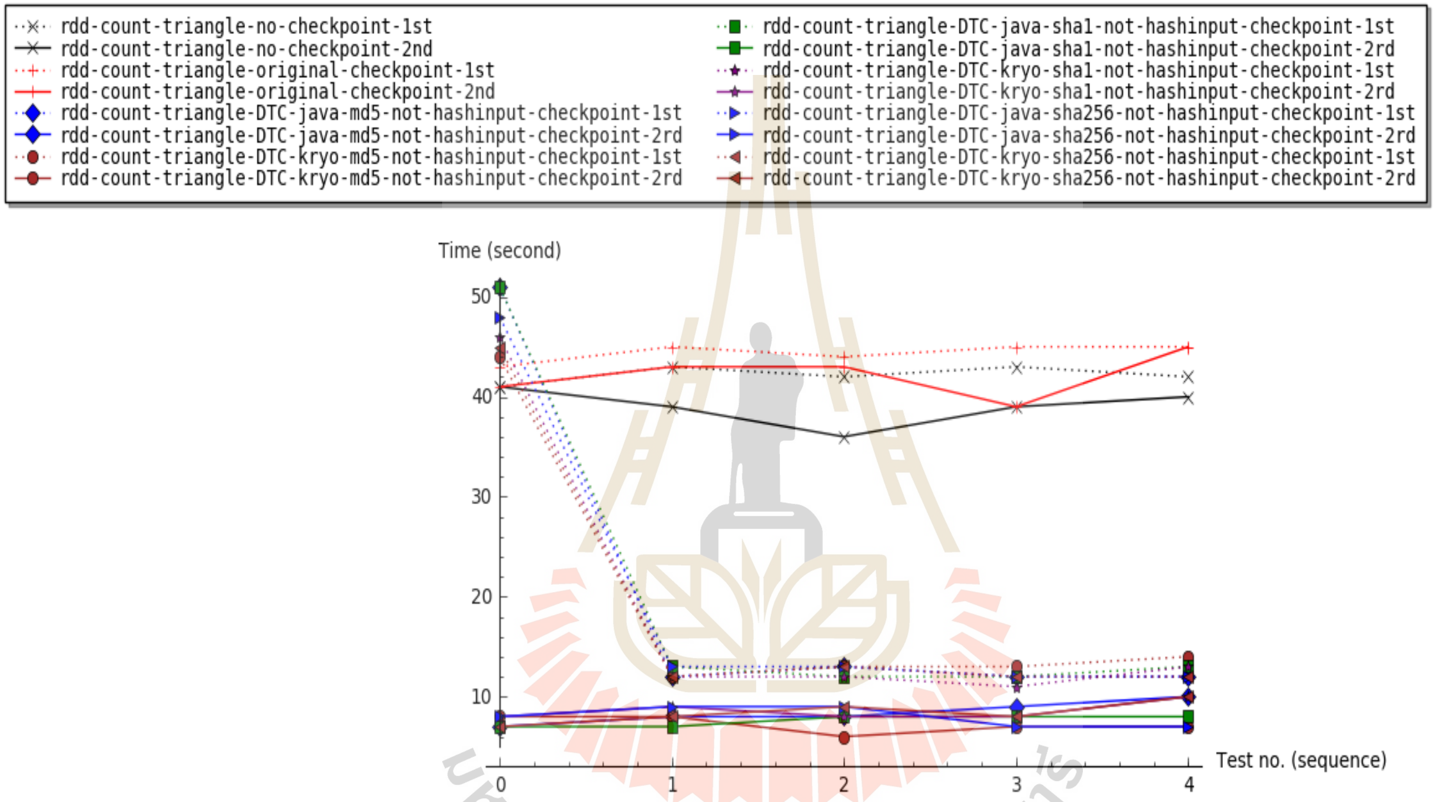
| **การทดสอบ** | **ครั้งที่ 1 (วินาที)** | **ครั้งที่ 2 (วินาที)** | **ครั้งที่ 3 (วินาที)** | **ครั้งที่ 4 (วินาที)** | **ครั้งที่ 5 (วินาที)** |
|---|---|---|---|---|---|
| No-checkpoint | | | | | |
| กรณีทดสอบ 1 | 41 | 43 | 42 | 43 | 42 |
| กรณีทดสอบ 2 | 41 | 39 | 36 | 39 | 40 |
| Spark Original | | | | | |
| กรณีทดสอบ 1 | 43 | 45 | 44 | 45 | 45 |
| กรณีทดสอบ 2 | 41 | 43 | 43 | 39 | 45 |
| DTC-MD5-Java | | | | | |
| กรณีทดสอบ 1 | 51 | 12 | 13 | 12 | 12 |
| กรณีทดสอบ 2 | 7 | 8 | 8 | 9 | 10 |
| DTC-MD5-Kryo | | | | | |
| กรณีทดสอบ 1 | 44 | 12 | 13 | 13 | 14 |
| กรณีทดสอบ 2 | 8 | 8 | 6 | 7 | 7 |
| DTC-SHA-1-Java | | | | | |
| กรณีทดสอบ 1 | 51 | 13 | 12 | 12 | 13 |
| กรณีทดสอบ 2 | 7 | 7 | 8 | 8 | 8 |
| DTC-SHA-1-Kryo | | | | | |
| กรณีทดสอบ 1 | 46 | 12 | 12 | 11 | 13 |
| กรณีทดสอบ 2 | 8 | 9 | 8 | 8 | 10 |
| DTC-SHA-256-Java | | | | | |
| กรณีทดสอบ 1 | 48 | 13 | 13 | 12 | 12 |
| กรณีทดสอบ 2 | 8 | 9 | 9 | 7 | 7 |
| DTC-SHA-256-Kryo | | | | | |
| กรณีทดสอบ 1 | 45 | 12 | 13 | 12 | 12 |
| กรณีทดสอบ 2 | 7 | 8 | 9 | 8 | 10 |

ตารางที่ 4.20 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมนับสามเหลี่ยม 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD และเข้ารหัสทางเดียวข้อมูลนำเข้า

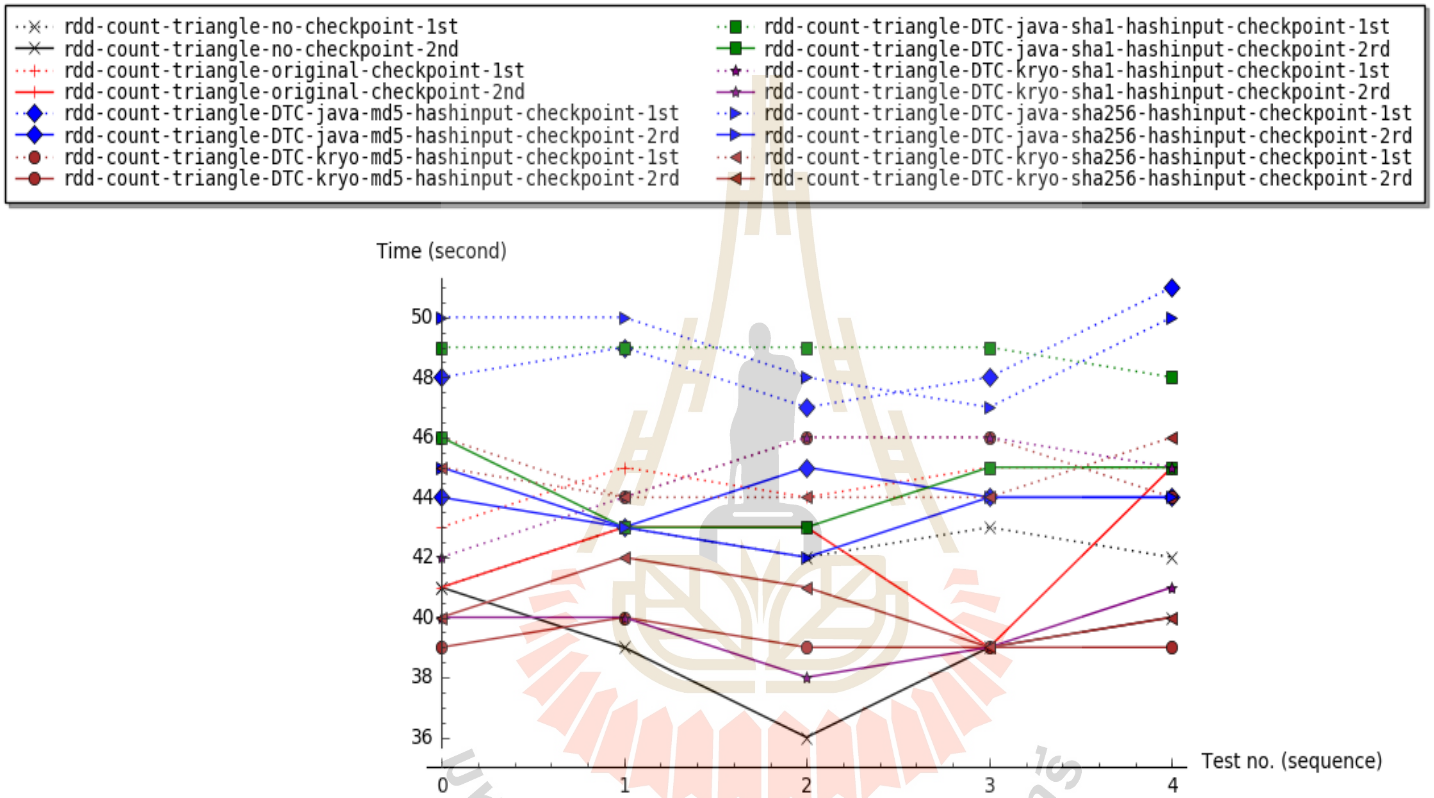
| **การทดสอบ** | **ครั้งที่ 1 (วินาที)** | **ครั้งที่ 2 (วินาที)** | **ครั้งที่ 3 (วินาที)** | **ครั้งที่ 4 (วินาที)** | **ครั้งที่ 5 (วินาที)** |
|---|---|---|---|---|---|
| No-checkpoint | | | | | |
| กรณีทดสอบ 1 | 41 | 43 | 42 | 43 | 42 |
| กรณีทดสอบ 2 | 41 | 39 | 36 | 39 | 40 |
| Spark Original | | | | | |
| กรณีทดสอบ 1 | 43 | 45 | 44 | 45 | 45 |
| กรณีทดสอบ 2 | 41 | 43 | 43 | 39 | 45 |
| DTC-MD5-Java | | | | | |
| กรณีทดสอบ 1 | 48 | 49 | 47 | 48 | 51 |
| กรณีทดสอบ 2 | 44 | 43 | 45 | 44 | 44 |
| DTC-MD5-Kryo | | | | | |
| กรณีทดสอบ 1 | 46 | 44 | 46 | 46 | 44 |
| กรณีทดสอบ 2 | 39 | 40 | 39 | 39 | 39 |
| DTC-SHA-1-Java | | | | | |
| กรณีทดสอบ 1 | 49 | 49 | 49 | 49 | 48 |
| กรณีทดสอบ 2 | 46 | 43 | 43 | 45 | 45 |
| DTC-SHA-1-Kryo | | | | | |
| กรณีทดสอบ 1 | 42 | 44 | 46 | 46 | 45 |
| กรณีทดสอบ 2 | 40 | 40 | 38 | 39 | 41 |
| DTC-SHA-256-Java | | | | | |
| กรณีทดสอบ 1 | 50 | 50 | 48 | 47 | 50 |
| กรณีทดสอบ 2 | 45 | 43 | 42 | 44 | 44 |
| DTC-SHA-256-Kryo | | | | | |
| กรณีทดสอบ 1 | 45 | 44 | 44 | 44 | 46 |
| กรณีทดสอบ 2 | 40 | 42 | 41 | 39 | 40 |

ตารางที่ 4.21 แสดงการใช้พื้นที่เก็บข้อมูลจุดตรวจสอบโปรแกรมนับสามเหลี่ยม 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD

| **วิธีการ** | **พื้นที่เก็บข้อมูล (เมกะไบต์)** |
|---|---|
| No-Checkopoint | 0 |
| Spark Original | 345 |
| DTC-MD5-Java แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 250 |
| DTC-SHA-1-Java แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 250 |
| DTC-SHA-256-Java แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 250 |
| DTC-MD5-Java แบบไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 25 |
| DTC-SHA-1-Java แบบไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 25 |
| DTC-SHA-256-Java แบบไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 25 |
| DTC-MD5-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 69 |
| DTC-SHA-1-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 69 |
| DTC-SHA-256-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 69 |
| DTC-MD5-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 6.9 |
| DTC-SHA-1-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 6.9 |
| DTC-SHA-256-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 6.9 |



รูปที่ 4.9 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมนับสามเหลี่ยม 2 กรณีทดสอบต่อเนื่อง ที่มีการปิด JVM กับตัวแปรแบบ RDD โดยไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้า



รูปที่ 4.10 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมนับสามเหลี่ยม 2 กรณีทดสอบต่อเนื่องที่มีการปิด JVM กับตัวแปรแบบ RDD โดยเข้ารหัสทางเดียวกับข้อมูลนำเข้า

### 4.3.6 กรณีทดสอบ 2 กรณีทดสอบต่อเนื่องแล้วปิดการทำงานของ JVM โดยการใช้โปรแกรม PageRank แบบ RDD

การทดสองนี้แบ่งกรณีทดสอบออกเป็น 2 กรณีทดสอบต่อเนื่องกันหลังจากนั้นแล้วจะปิดการทำงานของ JVM แล้วจึงเริ่มทดสอบซ้ำจำนวน 5 ครั้งเพื่อตรวจสอบการทำงานในกรณีที่มีการปิดโปรแกรม โดยจะใช้โปรแกรม PageRank บนข้อมูล LiveJournal ขนาด 4,847,571 โหนดและ 68,993,773 เส้นเชื่อมโดยทดสอบกับตัวแปรแบบ RDD ทดสอบเปรียบเทียบการตั้งค่า No-checkpoint กลไกสร้างจุดตรวจสอบดั้งเดิมของ Spark และกรอบงาน DTC ที่ตั้งค่าแบบต่าง ๆ ผลของการใช้เวลากรณีไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าแสดงในตารางที่ 4.22 และกรณีเข้ารหัสทางเดียวกับข้อมูลนำเข้าแสดงในตารางที่ 4.23 ซึ่งจะสังเกตแนวโน้มได้ตามรูปที่ 4.11 และรูปที่ 4.12 ซึ่งเป็นแผนภาพของการไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าและเข้ารหัสทางเดียวกับข้อมูลนำเข้าตามลำดับ

จากตารางเวลาผลลัพธ์ที่ได้ออกมานั้นหากเป็นกรณีที่ไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้าจะสังเกตเห็นว่าสามารถลดเวลาที่ใช้ในการประมวลผลกรณีทดสอบลงได้สูงสุดถึง 229 เท่าเมื่อเทียบกับกรณีทดสอบแรกในครั้งแรกดังแสดงในตารางที่ 4.22 ที่มีการใช้กรอบงาน DTC ตั้งค่าเข้ารหัสทางเดียวด้วยขั้นตอนวิธีแบบ SHA-256 และใช้รูปแบบข้อมูล Kryo ซึ่งสอดคล้องกับผลลัพธ์ที่ได้อภิปรายในหัวข้อก่อนหน้าเนื่องจากในกรณีทดสอบที่สองใช้เวลาน้อยมาก (น้อยกว่า 1 วินาที) แต่ในส่วนกรณีที่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้านั้นจะพบว่าใช้เวลาใกล้เคียงกันระหว่างจุดตรวจสอบดั้งเดิมของ Spark และกรอบงาน DTC ทุกการตั้งค่า จากการวิเคราะห์พบว่าเมื่อมีการอ่านพาร์ทิชันของข้อมูลขึ้นมากรอบงาน Spark จะพยายามแปลงข้อมูลให้อยู่ในรูปของวัตถุของคลาส ซึ่งในกรณีนี้คือคลาส ShippableVertexPartition ซึ่งอยู่ในแพคเกจ org.apache.spark.graphx.impl ทำให้กรอบงาน DTC ไม่สามารถตรวจสอบข้อมูลภายในได้เนื่องจากวัตถุจะถูกสร้างบนหน่วยความจำใหม่ทุกครั้ง

หากเปรียบเทียบพื้นที่เก็บข้อมูลที่ใช้ในจุดตรวจสอบในกรณีที่ใช้จุดตรวจสอบดั้งเดิมของ Spark กับกรอบงาน DTC กรณีที่เข้ารหัสข้อมูลนำเข้าโดยใช้รูปแบบข้อมูลแบบ Java นั้นพบว่าไม่สามารถลดการใช้พื้นที่ได้ และในกรณีที่ดีที่สุดของกรอบงาน DTC คือตั้งค่ารูปแบบข้อมูลแบบ Kryo และไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าสามารถลดการใช้พื้นที่ได้ถึง 17 เท่า ดังแสดงในตารางที่ 4.24

ตารางที่ 4.22 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรม PageRank 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD และไม่เข้ารหัสทางเดียวข้อมูลนำเข้า

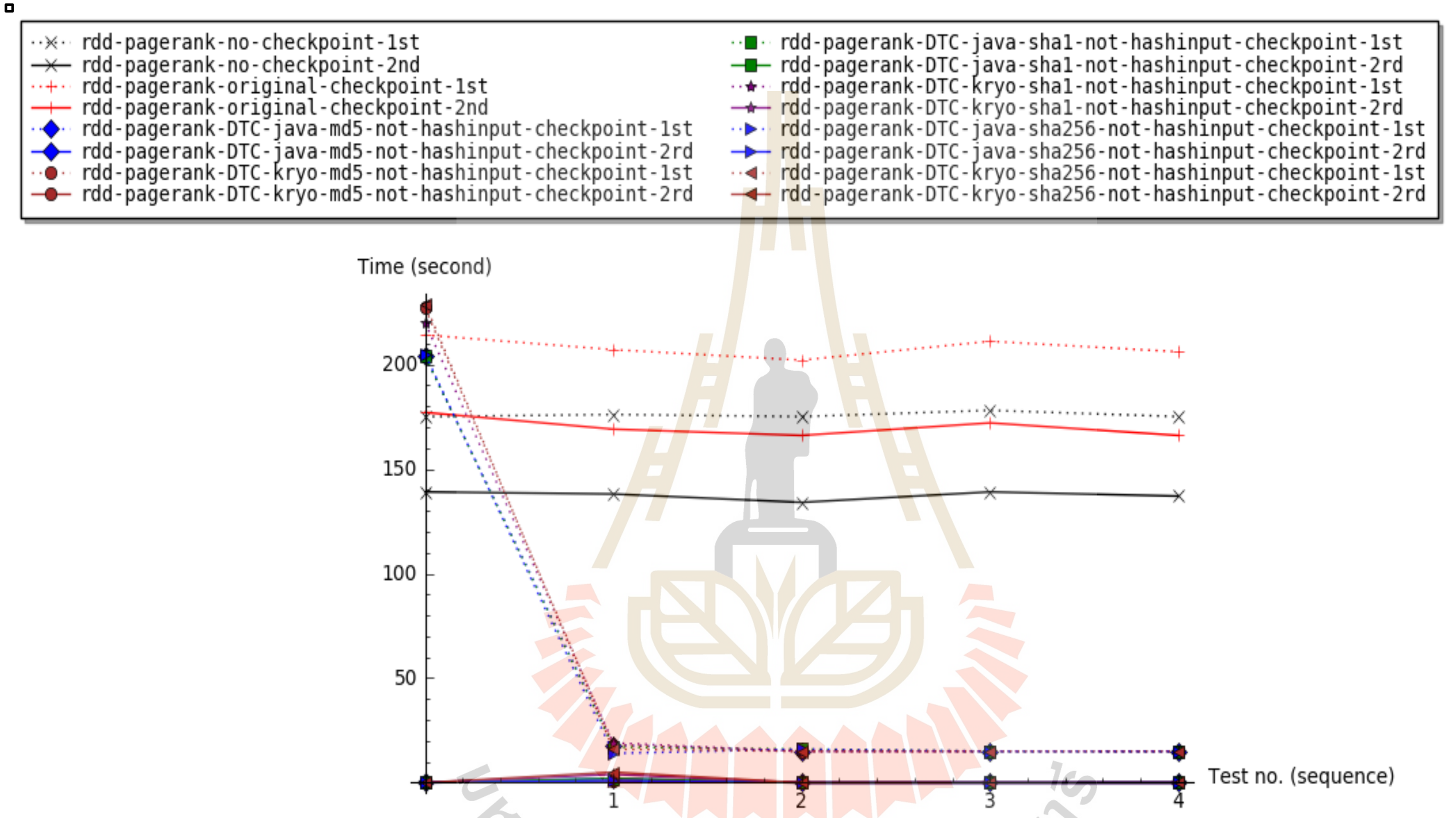
| **การทดสอบ** | **ครั้งที่ 1 (วินาที)** | **ครั้งที่ 2 (วินาที)** | **ครั้งที่ 3 (วินาที)** | **ครั้งที่ 4 (วินาที)** | **ครั้งที่ 5 (วินาที)** |
|---|---|---|---|---|---|
| No-checkpoint | | | | | |
| กรณีทดสอบ 1 | 175 | 176 | 175 | 178 | 175 |
| กรณีทดสอบ 2 | 139 | 138 | 134 | 139 | 137 |
| Spark Original | | | | | |
| กรณีทดสอบ 1 | 214 | 207 | 202 | 211 | 206 |
| กรณีทดสอบ 2 | 177 | 169 | 166 | 172 | 166 |
| DTC-MD5-Java | | | | | |
| กรณีทดสอบ 1 | 204 | 18 | 15 | 15 | 15 |
| กรณีทดสอบ 2 | 0 | 2 | 0 | 0 | 0 |
| DTC-MD5-Kryo | | | | | |
| กรณีทดสอบ 1 | 227 | 18 | 15 | 15 | 15 |
| กรณีทดสอบ 2 | 0 | 1 | 0 | 0 | 0 |
| DTC-SHA-1-Java | | | | | |
| กรณีทดสอบ 1 | 204 | 17 | 16 | 15 | 15 |
| กรณีทดสอบ 2 | 0 | 2 | 0 | 0 | 0 |
| DTC-SHA-1-Kryo | | | | | |
| กรณีทดสอบ 1 | 220 | 19 | 15 | 15 | 15 |
| กรณีทดสอบ 2 | 0 | 4 | 0 | 0 | 0 |
| DTC-SHA-256-Java | | | | | |
| กรณีทดสอบ 1 | 205 | 14 | 16 | 15 | 15 |
| กรณีทดสอบ 2 | 0 | 1 | 0 | 0 | 0 |
| DTC-SHA-256-Kryo | | | | | |
| กรณีทดสอบ 1 | 229 | 16 | 15 | 15 | 15 |
| กรณีทดสอบ 2 | 0 | 5 | 0 | 0 | 0 |

ตารางที่ 4.23 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรม PageRank 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD และเข้ารหัสทางเดียวข้อมูลนำเข้า

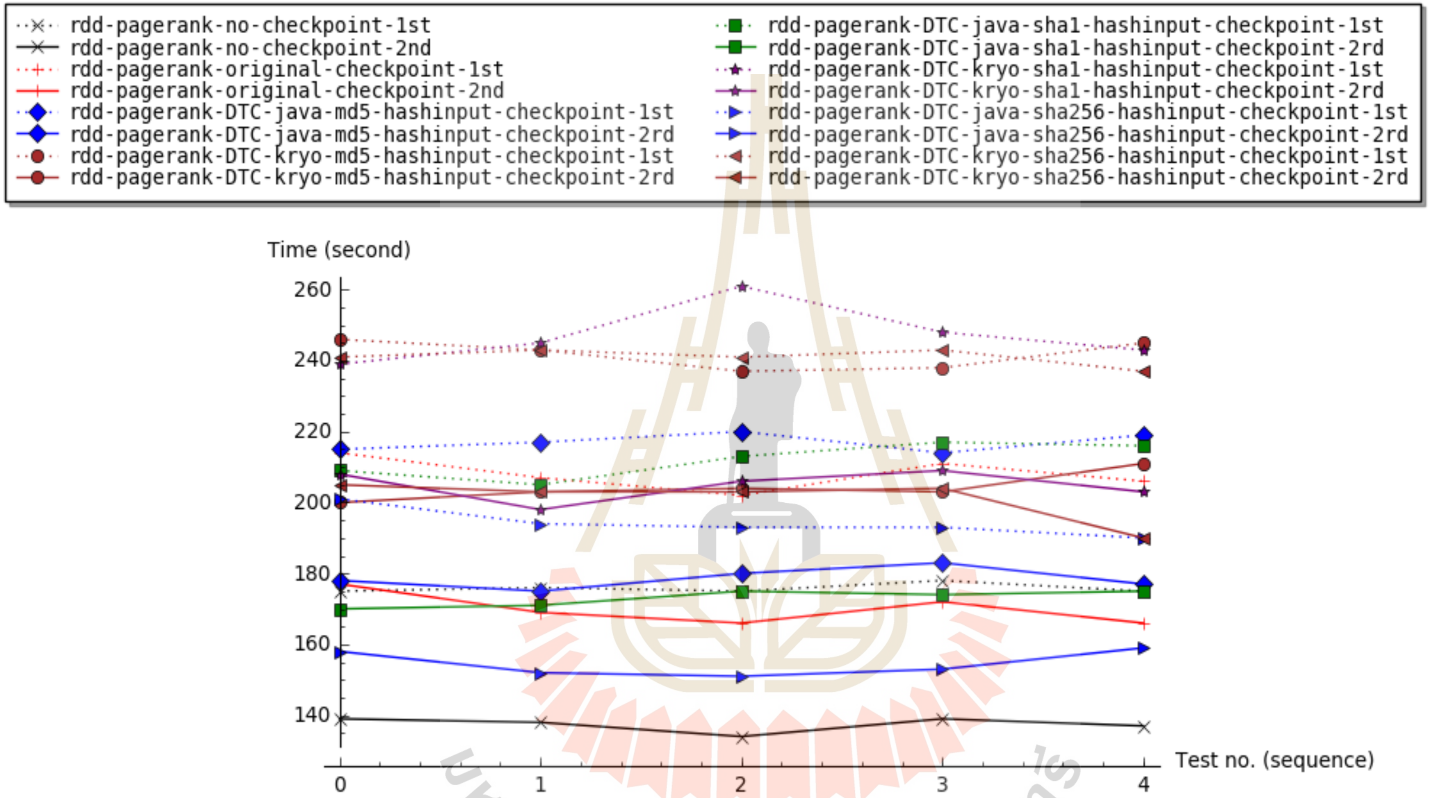
| การทดสอบ | ครั้งที่ 1 (วินาที) | ครั้งที่ 2 (วินาที) | ครั้งที่ 3 (วินาที) | ครั้งที่ 4 (วินาที) | ครั้งที่ 5 (วินาที) |
|---|---|---|---|---|---|
| No-checkpoint | | | | | |
| กรณีทดสอบ 1 | 175 | 176 | 175 | 178 | 175 |
| กรณีทดสอบ 2 | 139 | 138 | 134 | 139 | 137 |
| Spark Original | | | | | |
| กรณีทดสอบ 1 | 214 | 207 | 202 | 211 | 206 |
| กรณีทดสอบ 2 | 177 | 169 | 166 | 172 | 166 |
| DTC-MD5-Java | | | | | |
| กรณีทดสอบ 1 | 215 | 217 | 220 | 214 | 219 |
| กรณีทดสอบ 2 | 178 | 175 | 180 | 183 | 177 |
| DTC-MD5-Kryo | | | | | |
| กรณีทดสอบ 1 | 246 | 243 | 237 | 238 | 245 |
| กรณีทดสอบ 2 | 200 | 203 | 204 | 203 | 211 |
| DTC-SHA-1-Java | | | | | |
| กรณีทดสอบ 1 | 209 | 205 | 213 | 217 | 216 |
| กรณีทดสอบ 2 | 170 | 171 | 175 | 174 | 175 |
| DTC-SHA-1-Kryo | | | | | |
| กรณีทดสอบ 1 | 239 | 245 | 261 | 248 | 243 |
| กรณีทดสอบ 2 | 208 | 198 | 206 | 209 | 203 |
| DTC-SHA-256-Java | | | | | |
| กรณีทดสอบ 1 | 201 | 194 | 193 | 193 | 190 |
| กรณีทดสอบ 2 | 158 | 152 | 151 | 153 | 159 |
| DTC-SHA-256-Kryo | | | | | |
| กรณีทดสอบ 1 | 241 | 243 | 241 | 243 | 237 |
| กรณีทดสอบ 2 | 205 | 203 | 203 | 204 | 190 |

ตารางที่ 4.24 แสดงการใช้พื้นที่เก็บข้อมูลจุดตรวจสอบโปรแกรม PageRank 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD

| **วิธีการ** | **พื้นที่เก็บข้อมูล (กิโลไบต์)** |
|---|---|
| No-Checkopoint | 0 |
| Spark Original | 340 |
| DTC-MD5-Java แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 340 |
| DTC-SHA-1-Java แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 340 |
| DTC-SHA-256-Java แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 340 |
| DTC-MD5-Java แบบไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 34 |
| DTC-SHA-1-Java แบบไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 34 |
| DTC-SHA-256-Java แบบไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 34 |
| DTC-MD5-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 200 |
| DTC-SHA-1-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 200 |
| DTC-SHA-256-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 200 |
| DTC-MD5-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 20 |
| DTC-SHA-1-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 20 |
| DTC-SHA-256-Kryo แบบมีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า | 20 |



รูปที่ 4.11 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรม PageRank 2 กรณีทดสอบต่อเนื่อง ที่มีการปิด JVM กับตัวแปรแบบ RDD โดยไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้า



รูปที่ 4.12 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรม PageRank 2 กรณีทดสอบต่อเนื่องที่มีการปิด JVM กับตัวแปรแบบ RDD โดยเข้ารหัสทางเดียวกับข้อมูลนำเข้า

### 4.3.7 กรณีทดสอบ 2 กรณีทดสอบต่อเนื่องแล้วปิดการทำงานของ JVM โดยการใช้โปรแกรม PageRank แบบ RDD

การทดสองนี้แบ่งกรณีทดสอบออกเป็น 2 กรณีทดสอบต่อเนื่องกันหลังจากนั้นแล้วจะปิดการทำงานของ JVM แล้วจึงเริ่มทดสอบซ้ำจำนวน 5 ครั้งเพื่อตรวจสอบการทำงานในกรณีที่มีการปิดโปรแกรม โดยจะใช้คำนวณหาค่าพายขนาด 1,000,000,000 ครั้งโดยทดสอบกับตัวแปรแบบ RDD ทดสอบเปรียบเทียบการตั้งค่า No-checkpoint กลไกสร้างจุดตรวจสอบดั้งเดิมของ Spark และกรอบงาน DTC ที่ตั้งค่าแบบต่าง ๆ ผลของการใช้เวลากรณีไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าแสดงในตารางที่ 4.25 และกรณีเข้ารหัสทางเดียวกับข้อมูลนำเข้าแสดงในตารางที่ 4.26 ซึ่งจะสังเกตแนวโน้มได้ตามรูปที่ 4.13 และรูปที่ 4.14 ซึ่งเป็นแผนภาพของการไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้าและเข้ารหัสทางเดียวกับข้อมูลนำเข้าตามลำดับ

จากตารางเวลาผลลัพธ์ที่ได้ออกมานั้นหากเป็นกรณีที่ไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้าจะสังเกตเห็นว่าสามารถลดเวลาที่ใช้ในการประมวลผลกรณีทดสอบลงได้สูงสุดถึง 38.6 เท่าเมื่อเทียบกับกรณีทดสอบแรกในครั้งแรกดังแสดงในตารางที่ 4.25 ที่มีการใช้กรอบงาน DTC ตั้งค่าเข้ารหัสทางเดียวด้วยขั้นตอนวิธีแบบ MD5 และใช้รูปแบบข้อมูล Kryo แต่ในส่วนกรณีที่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้านั้นจะพบว่าสามาลดเวลาได้สูงสุด 18 เท่ามีการใช้กรอบงาน DTC ตั้งค่าเข้ารหัสทางเดียวด้วยขั้นตอนวิธีแบบ MD5 และใช้รูปแบบข้อมูล Java ดังแสดงในตารางที่ 4.26

หากเปรียบเทียบพื้นที่เก็บข้อมูลที่ใช้ในจุดตรวจสอบในกรณีที่ใช้จุดตรวจสอบดั้งเดิมของ Spark กับกรอบงาน DTC กรณีใช้รูปแบบข้อมูลแบบ Java นั้นพบว่าสามารถลดการใช้พื้นที่ได้ 9.9 เท่า และในกรณีที่ดีที่สุดของกรอบงาน DTC คือตั้งค่ารูปแบบข้อมูลแบบ Kryo สามารถลดการใช้พื้นที่ได้ถึง 17.9 เท่า ดังแสดงในตารางที่ 4.27

ตารางที่ 4.25 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมคำนวณหาค่าพาย 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD และไม่เข้ารหัสทางเดียวข้อมูลนำเข้า

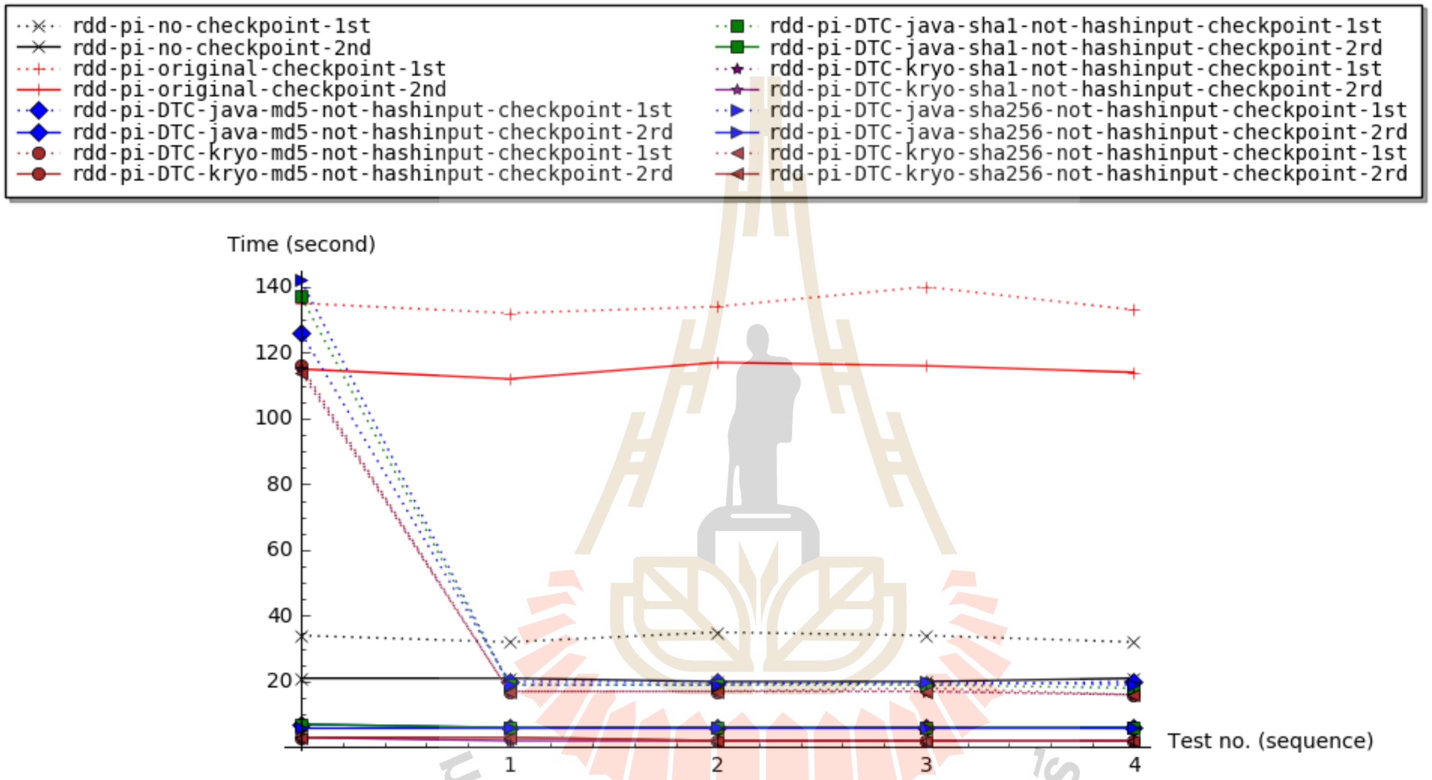
| การทดสอบ | ครั้งที่ 1 (วินาที) | ครั้งที่ 2 (วินาที) | ครั้งที่ 3 (วินาที) | ครั้งที่ 4 (วินาที) | ครั้งที่ 5 (วินาที) |
|---|---|---|---|---|---|
| No-checkpoint | | | | | |
| กรณีทดสอบ 1 | 34 | 32 | 35 | 34 | 32 |
| กรณีทดสอบ 2 | 21 | 21 | 20 | 20 | 21 |
| Spark Original | | | | | |
| กรณีทดสอบ 1 | 114 | 132 | 134 | 140 | 133 |
| กรณีทดสอบ 2 | 115 | 112 | 117 | 116 | 114 |
| DTC-MD5-Java | | | | | |
| กรณีทดสอบ 1 | 126 | 20 | 20 | 19 | 20 |
| กรณีทดสอบ 2 | 7 | 6 | 6 | 6 | 6 |
| DTC-MD5-Kryo | | | | | |
| กรณีทดสอบ 1 | 116 | 17 | 17 | 18 | 16 |
| กรณีทดสอบ 2 | 3 | 3 | 2 | 2 | 2 |
| DTC-SHA-1-Java | | | | | |
| กรณีทดสอบ 1 | 137 | 19 | 19 | 19 | 18 |
| กรณีทดสอบ 2 | 7 | 6 | 6 | 6 | 6 |
| DTC-SHA-1-Kryo | | | | | |
| กรณีทดสอบ 1 | 115 | 17 | 17 | 17 | 16 |
| กรณีทดสอบ 2 | 3 | 2 | 2 | 2 | 2 |
| DTC-SHA-256-Java | | | | | |
| กรณีทดสอบ 1 | 142 | 19 | 19 | 20 | 19 |
| กรณีทดสอบ 2 | 6 | 6 | 6 | 6 | 6 |
| DTC-SHA-256-Kryo | | | | | |
| กรณีทดสอบ 1 | 114 | 17 | 17 | 17 | 16 |
| กรณีทดสอบ 2 | 3 | 3 | 2 | 2 | 2 |

ตารางที่ 4.26 แสดงเวลาที่ใช้ในการทดสอบเปรียบเทียบเงื่อนไขโปรแกรมคำนวณหาค่าพาย 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD และเข้ารหัสทางเดียวข้อมูลนำเข้า

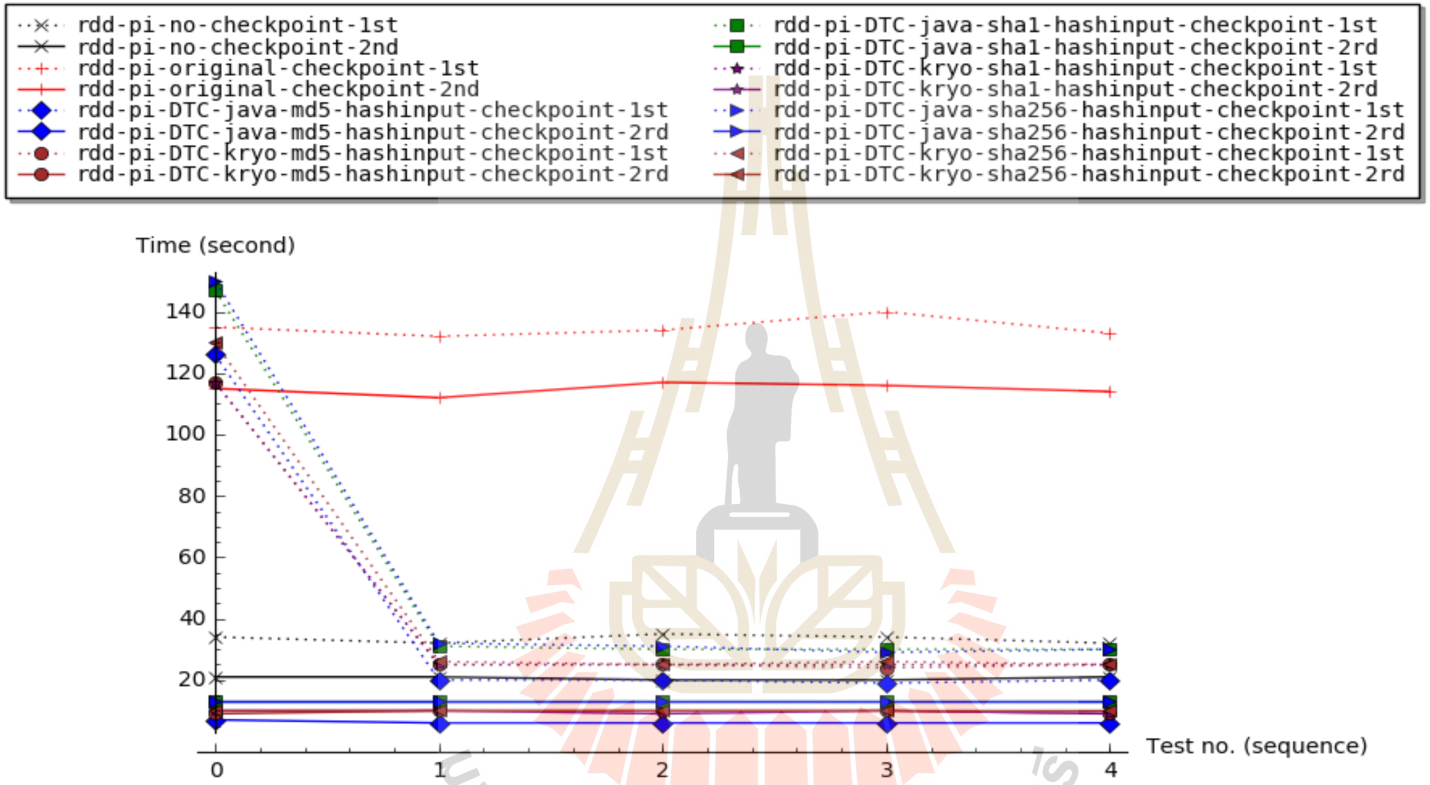
| **การทดสอบ** | **ครั้งที่ 1 (วินาที)** | **ครั้งที่ 2 (วินาที)** | **ครั้งที่ 3 (วินาที)** | **ครั้งที่ 4 (วินาที)** | **ครั้งที่ 5 (วินาที)** |
|---|---|---|---|---|---|
| No-checkpoint | | | | | |
| กรณีทดสอบ 1 | 34 | 32 | 35 | 34 | 32 |
| กรณีทดสอบ 2 | 21 | 21 | 20 | 20 | 21 |
| Spark Original | | | | | |
| กรณีทดสอบ 1 | 135 | 132 | 134 | 140 | 133 |
| กรณีทดสอบ 2 | 115 | 112 | 117 | 116 | 114 |
| DTC-MD5-Java | | | | | |
| กรณีทดสอบ 1 | 126 | 20 | 20 | 19 | 20 |
| กรณีทดสอบ 2 | 7 | 6 | 6 | 6 | 6 |
| DTC-MD5-Kryo | | | | | |
| กรณีทดสอบ 1 | 117 | 25 | 25 | 24 | 25 |
| กรณีทดสอบ 2 | 9 | 10 | 10 | 10 | 9 |
| DTC-SHA-1-Java | | | | | |
| กรณีทดสอบ 1 | 147 | 31 | 30 | 30 | 30 |
| กรณีทดสอบ 2 | 13 | 13 | 13 | 13 | 13 |
| DTC-SHA-1-Kryo | | | | | |
| กรณีทดสอบ 1 | 117 | 25 | 25 | 25 | 25 |
| กรณีทดสอบ 2 | 10 | 10 | 9 | 10 | 9 |
| DTC-SHA-256-Java | | | | | |
| กรณีทดสอบ 1 | 150 | 32 | 31 | 29 | 30 |
| กรณีทดสอบ 2 | 13 | 13 | 13 | 13 | 13 |
| DTC-SHA-256-Kryo | | | | | |
| กรณีทดสอบ 1 | 130 | 26 | 25 | 26 | 25 |
| กรณีทดสอบ 2 | 10 | 10 | 10 | 10 | 10 |

ตารางที่ 4.27 แสดงการใช้พื้นที่เก็บข้อมูลจุดตรวจสอบโปรแกรมคำนวณหาค่าพาย 2 กรณีทดสอบต่อเนื่องที่มีการปิดการทำงาน JVM กับตัวแปรแบบ RDD

| **วิธีการ** | **พื้นที่เก็บข้อมูล (เมกะไบต์)** |
|---|---|
| No-Checkopoint | 0 |
| Spark Original | 39.5 |
| DTC-MD5-Java | 4.0 |
| DTC-SHA-1-Java | 4.0 |
| DTC-SHA-256-Java | 4.0 |
| DTC-MD5-Kryo | 2.2 |
| DTC-SHA-1-Kryo | 2.2 |
| DTC-SHA-256-Kryo | 2.2 |



รูปที่ 4.13 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมคำนวณหาค่าพาย 2 กรณีทดสอบ ต่อเนื่องที่มีการปิด JVM กับตัวแปรแบบ RDD โดยไม่เข้ารหัสทางเดียวกับข้อมูลนำเข้า



รูปที่ 4.14 แผนภาพเปรียบเทียบเวลาที่ใช้ในการประมวลผลโปรแกรมคำนวณหาค่าพาย 2 กรณีทดสอบ ต่อเนื่องที่มีการปิด JVM กับตัวแปรแบบ RDD โดยเข้ารหัสทางเดียวกับข้อมูลนำเข้า

# บทที่ 5

# สรุปผลการวิจัยและข้อเสนอแนะ

จากความก้าวหน้าของเทคโนโลยีการเก็บข้อมูลและอุปกรณ์อิเล็กทรอนิกส์ที่มีความสามารถในการเก็บรวบรวมข้อมูลได้หลากหลายรูปแบบ ทำให้เกิดยุคสมัยของข้อมูลที่ถูกเรียกว่ายุคของข้อมูลขนาดใหญ่ ซึ่งก่อให้เกิดการนำเสนอเครื่องมือเพื่อประมวลผลข้อมูลขนาดใหญ่เหล่านี้ หนึ่งในนั้นคือกรอบงาน Spark ซึ่งนำเอาลักษณะตัวแบบโปรแกรม MapReduce มาใช้เพื่อประมวลผล แต่อย่างไรก็ตามพบว่าเทคนิคการทดสอบซอฟต์แวร์ที่ใช้อยู่กับการพัฒนาซอฟต์แวร์ปกตินั้นไม่เหมาะกับการทดสอบซอฟต์แวร์ที่ใช้ประมวลผลข้อมูลขนาดใหญ่

งานวิจัยในวิทยานิพนธ์นี้นำเสนอกรอบงาน DTC ซึ่งเป็นกรอบงานที่ใช้ทดสอบซอฟต์แวร์เพื่อให้ผู้ใช้หรือนักพัฒนาซอฟต์แวร์สามารถทดสอบซอฟต์แวร์เชิงหน่วยบน Spark เพื่อควบคุมคุณภาพของซอฟต์แวร์ไว้ได้ และเมื่อสามารถทดสอบเชิงหน่วยได้ก็สามารถนำไปทดสอบในระดับอื่น เช่น การทดสอบแบบถดถอยได้ โดยที่บทสรุปของงานวิจัยจะอยู่หัวข้อที่ 5.1 และข้อเสนอแนะสำหรับงานวิจัยนี้จะอยู่หัวข้อที่ 5.2

## 5.1 สรุปผลการวิจัย

จากการทดสอบการใช้งานของกรอบงาน DTC เปรียบเทียบการใช้งานกับกรณีที่ไม่ใช้งานจุดตรวจสอบเลย, ใช้งานกลไกสร้างจุดตรวจสอบดั้งเดิมของ Spark จะพบว่าการตั้งค่ากรอบงาน DTC นั้นไม่ว่าจะตั้งขั้นตอนวิธีเป็น MD5, SHA-1 หรือ SHA-256 ก็ได้ผลลัพธ์ที่ใกล้เคียงกันในด้านเวลาการประมวลผล โดยกรณีทดสอบกรณีแรกนั้นจะสูงแต่ในกรณีทดสอบถัดมานั้นจะใช้เวลาการประมวลผลลดลงอย่างมาก ต่างกับกรณีที่ไม่ใช้จุดตรวจสอบใดเลยและใช้กลไกการสร้างจุดตรวจสอบดั้งเดิมของ Spark ที่ใช้เวลาประมวลผลใกล้เคียงกันในทุกกรณี ขณะการใช้พื้นที่สร้างจุดตรวจสอบนั้นกลไกการสร้างจุดตรวจสอบดั้งเดิมของ Spark จะใช้พื้นที่มากที่สุด แต่หากใช้กรอบงาน DTC นั้นพื้นที่ที่ใช้จะขึ้นอยู่กับรูปแบบข้อมูล หากเป็นตัวแปรแบบ RDD การตั้งค่าเป็นรูปแบบ

ข้อมูลแบบ Java จะใช้พื้นที่เก็บข้อมูลมากกว่าการตั้งค่ารูปแบบข้อมูลเป็นแบบ Kryo และหากเป็นตัวแปรแบบ DataSet การตั้งค่ารูปแบบข้อมูลแบบ Avro จะใช้พื้นที่น้อยกว่ารูปแบบข้อมูลแบบ Parquet

กรณีที่เปรียบเทียบกับกรอบงาน Spark-flow จะพบว่ากรอบงาน DTC นั้นมีความสามารถที่เหนือกว่าทั้งในด้านการใช้เวลาการประมวลผลและพื้นที่เก็บข้อมูลที่น้อยกว่า ทั้งนี้เนื่องจากระบบวิเคราะห์รหัสไบต์ของกรอบงาน Spark-flow สามารถตรวจจับได้เพียงในระดับการประมวลผลใหม่หลังจากที่มีการปิดการทำงานของ JVM ไป แต่จะไม่สามารถใช้ในกรณีที่มีหลายกรณีทดสอบต่อเนื่องกันเนื่องจากไม่มีการกรองข้อมูลส่วนที่ไม่จำเป็นออก พบว่ากรอบงาน DTC ในกรณีทดสอบแรกนั้นจะใช้เวลาใกล้เคียงกับกรอบงาน Spark-flow หากไม่มีการเข้ารหัสทางเดียวกับข้อมูลนำเข้า แต่ในกรณีทดสอบที่สองและกรณีทดสอบถัดมาจะพบว่ากรอบงาน DTC จะใช้เวลาและพื้นที่เก็บข้อมูลน้อยกว่าอย่างมาก

ด้วยลักษณะที่สามารถใช้ตัวแปรทั้งแบบ RDD และ DataSet ซ้ำในหลาย ๆ กรณีทดสอบทั้งกรณีทดสอบที่ต่อเนื่องหรือมีการปิดการทำงานของ JVM ความสามารถในการทำซ้ำข้อผิดพลาดได้ (Re-produce Error) ที่สามารถทำซ้ำแม้กระทั่งข้อมูลที่สุ่ม รวมไปความความเร็วในการประมวลผลและการใช้พื้นที่เก็บข้อมูล ทำให้กรอบงาน DTC เหมาะสมอย่างยิ่งในการทดสอบซอฟต์แวร์ที่จะใช้ประมวลผลกับข้อมูลที่มีขนาดใหญ่ โดยที่นักพัฒนาไม่จำเป็นต้องเลือกกลุ่มข้อมูลตัวแทนขึ้นมาทดสอบซึ่งอาจทำให้ข้อบกพร่องนั้นหลุดรอดกรณีทดสอบไป

## 5.2 ข้อเสนอแนะ

การพัฒนากรอบงาน DTC นั้นยังมีข้อที่ควรปรับปรุงดังเช่นกรณีที่ไม่สามารถอ่านข้อมูลได้อย่างถูกต้องในกรณีที่ข้อมูลนำเข้าถูกอ่านเข้ามาโดยกรอบงาน Spark แล้วถูกแปลงเป็นวัตถุของคลาส แทนที่จะเป็นพาร์ทิชันของข้อมูลตัวอักษรปกติ และความสามารถของกรอบงานที่ดัดแปลงไปใช้ช่วยการประมวลผลงานจริงที่ไม่ใช้การทดสอบซอฟต์แวร์แทนกลไกสร้างจุดตรวจสอบดั้งเดิมของ Spark ก็สามารถทำได้

นอกจากนั้นยังมีกลไกอื่น เช่น กลไกการแคชซึ่งเป็นกลไกที่ช่วยให้ข้อมูล RDD หรือ DataSet สามารถค้างอยู่บนหน่วยความจำ กลไก Persist ที่สามารถกำหนดให้ข้อมูล RDD หรือ DataSet สามารถค้างอยู่บนหน่วยความจำหรือมีการเขียนลงแหล่งเก็บข้อมูล กลไกการบีบอัดข้อมูลของรูปแบบเก็บข้อมูลแบบ Avro และกลไก Garbage Collection ที่มีให้เลือกใช้หลายกลไก ซึ่ง

อาจจะช่วยให้การประมวลผลนั้นยังไม่ถูกทดสอบในงานวิจัยนี้ก็สามารถนำมาต่อยอดงานวิจัยให้การประมวลผลทำได้รวดเร็วขึ้นและใช้พื้นที่ในการสร้างจุดตรวจสอบลดลง

# รายการอ้างอิง

โทมัส สเตอริ่ง และ กษิดิศ ชาญเชี่ยว (2012). **การใช้งานระบบคอมพิวเตอร์สมรรถนะสูงเบื้องต้น.** Retrieved May 14, 2016, from http://vasabilab.cs.tu.ac.th/books/7c396f8f

ธนา หงษ์สุวรรณ. (n.d.). **Message Authentication and Digital Signature.** Retrieved April 12, 2016, from http://www.msit.mut.ac.th/member/filemanager/share_file/bon/security/Digital%20Signature.doc

Alvisi, L., & Marzullo, K. (1995, May). **Message logging: Pessimistic, optimistic, and causal.** In Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on (pp. 229-236). IEEE.

Alvisi, L., Rao, S., Husain, S. A., De Mel, A., & Elnozahy, E. (1999, June). **An analysis of communication-induced checkpointing.** In ftcs (p. 242). IEEE.

Anthony, S. (2012). **What can you do with a supercomputer?** Retrieved May 14, 2016, from http://www.extremetech.com/extreme/122159-what-can-you-do-with-a-supercomputer

Baldoni, R., Hélary, J. M., Mostefaoui, A., & Raynal, M. (1997, June). **A communication-induced checkpointing protocol that ensures rollback-dependency trackability.** In Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on (pp. 68-77). IEEE.

Beck, K. (2003). **Test-driven development: by example.** Addison-Wesley Professional.

Bloomberg. (2017, January 05). **Bloomberg/spark-flow.** Retrieved November 16, 2017, from https://github.com/bloomberg/spark-flow

Cao, G., & Singhal, M. (1998). **On coordinated checkpointing in distributed systems. Parallel and Distributed Systems**, IEEE Transactions on, 9(12), 1213-1225.

Chandy, K. M., & Lamport, L. (1985). **Distributed snapshots: determining global states of distributed systems.** ACM Transactions on Computer Systems (TOCS), 3(1), 63-75.

Dean, J., & Ghemawat, S. (2008). **MapReduce: Simplified Data Processing on Large Clusters.** Communications of the ACM - 50th Anniversary Issue: 1958 - 2008, 51(1), 107–113.

Deconinck, G., & Lauwereins, R. (1997, July). **User-triggered checkpointing: system-independent and scalable application recovery**. In Computers and Communications, 1997. Proceedings., Second IEEE Symposium on (pp. 418-423). IEEE.

Deshpande, N., & Kamalapur, S. (2008). **Fault Torelence.** In Distributed System (2nd ed., pp. 5–52). Technical Publications Pune.

Dinh, M. N., Abramson, D., Kurniawan, D., Jin, C., Moench, B., & DeRose, L. (2011, May**). Assertion based parallel debugging.** In Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on (pp. 63-72). IEEE.

Elnozahy, E. N., & Zwaenepoel, W. (1994, June). **On the use and implementation of message logging.** In Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on (pp. 298-307). IEEE.

Ferreira, K. B., Riesen, R., Brighwell, R., Bridges, P., & Arnold, D. (2011). **Libhashckpt: Hash-based Incremental Checkpointing Using GPU's.** In Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface (pp. 272–281). Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=2042476.2042507

Ferreira, K. B., Riesen, R., Bridges, P., Arnold, D., & Brightwell, R. (2014). **Accelerating incremental checkpointing for extreme-scale computing.** Future Generation Computer Systems, 30, 66-77.

Gulzar, M. A., Interlandi, M., Condie, T., & Kim, M. (2016, November). **Bigdebug: Interactive debugger for big data analytics in apache spark**. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 1033-1037). ACM.

Hamill, P. (2004). **Unit Test Frameworks: Tools for High-Quality Software Development.** " O'Reilly Media, Inc.".

IBM. (n.d.). **What is big data? [Business].** Retrieved March 11, 2016, from http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html

Jangjaimon, I., & Tzeng, N. F. (2013, May). **Adaptive incremental checkpointing via delta compression for networked multicore systems.** In Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on (pp. 7-18). IEEE.

Jaggi, P. K., & Singh, A. K. (2011). **Message efficient global snapshot recording using a self stabilizing spanning tree in a MANET.** International Journal of Communication Networks and Information Security, 3(3), 247.

Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). **Learning Spark: Lightning-Fast Big Data Analysis** (1st ed.). O'Reilly Media.

Kang, S. J., Lee, S. Y., & Lee, K. M. (2015). **Performance comparison of OpenMP, MPI, and mapreduce in practical problems.** Advances in Multimedia, 2015, 7.

Koo, R., & Toueg, S. (1987). **Checkpointing and rollback-recovery for distributed systems.** Software Engineering, IEEE Transactions on, (1), 23-31.

Kshemkalyani, A. D., Raynal, M., & Singhal, M. (1995). **An introduction to snapshot algorithms in distributed computing.** Distributed systems engineering, 2(4), 224.

Kuleshov, E. (2007). **Using the ASM framework to implement common Java bytecode transformation patterns.** Aspect-Oriented Software Development.

Kumar, P., & Khunteta, A. (2010). **A Minimum Process Coordinated Checkpointing Protocol for Mobile Distributed Systems.** International Journal of Computer Science Issues, 7(3), 23-32.

Lai, T. H., & Yang, T. H. (1987). **On distributed snapshots.** Information Processing Letters, 25(3), 153-158.

Laskowski, J. (2016). **Mastering Apache Spark.** Retrieved May 15, 2016, from https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-rdd-checkpointing.html

Leskovec, J., Lang, K. J., Dasgupta, A., & Mahoney, M. W. (2009). **Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters.** Internet Mathematics, 6(1), 29-123.

Maruyama, M., Tsumura, T., & Nakashima, H. (2005). **Parallel Program Debugging based on Data-Replay** (pp. 151–156). Presented at the International Conference on Parallel and Distributed Computing Systems,.

Komorowski, M. (2014, March 9). **a history of storage cost (update).** Retrieved March 12, 2016, from http://www.mkomo.com/cost-per-gigabyte-update

Meneses, E., Mendes, C. L., & Kalé, L. V. (2010, May). **Team-based message logging: Preliminary results.** In Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on (pp. 697-702). IEEE.

Moody, A., Bronevetsky, G., Mohror, K., & De Supinski, B. R. (2010, November**). Design, modeling, and evaluation of a scalable multi-level checkpointing system.** In High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for (pp. 1-11). IEEE.

Netzer, R. H., & Xu, J. (1995). **Necessary and sufficient conditions for consistent global snapshots.** IEEE Transactions on Parallel & Distributed Systems, (2), 165-169.

Plank, J. S., Beck, M., Kingsley, G., & Li, K. (1994). **Libckpt: Transparent checkpointing under unix.** Computer Science Department.

Pressman, R. S. (2010).**Software Engineering*: *A Practitioner's Approach** (7th ed.). McGraw-Hill.

Quinn, M.J. (2003). **Parallel Programming in C with MPI and OpenMP**(1st ed.). McGraw-Hill Science/Engineering/Math.

Rusu, C., Grecu, C., & Anghel, L. (2008, January). **Coordinated versus uncoordinated checkpoint recovery for network-on-chip based systems.** In Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on (pp. 32-37). IEEE.

Saridakis, T. (2003, September). **Design patterns for checkpoint-based rollback recovery**. In Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP).

Schwartz-Narbonne, D., Liu, F., Pondicherry, T., August, D., & Malik, S. (2011). **Parallel assertions for debugging parallel programs** (pp. 181–190). Presented at the Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on. http://doi.org/10.1109/MEMCOD.2011.5970525

Sharma, P., Guo, T., He, X., Irwin, D., & Shenoy, P. (2016, April). **Flint: batch-interactive data-intensive processing on transient servers.** In Proceedings of the Eleventh European Conference on Computer Systems (p. 6). ACM.

Shetty, A., & Marshall, N. (2014). **Testing Spark: Best Practices.** Presented at the Spark Summit 2014. Retrieved from https://spark-summit.org/2014/talk/testing-spark-best-practices

Shoro, A. G., & Soomro, T. R. (2015). **Big Data Analysis: Apache Spark Perspective.** Global Journal of Computer Science and Technology, 15(1).

Smoak, C., Widel, M., & Truong, S. (2012). **Checksum Please: A Way to Ensure Data Integrity.** In *PharmaSUG-2012*.

Spillner, A., Linz, T., & Schaefer, H. (2014). **Software testing foundations: a study guide for the certified tester exam.** Rocky Nook, Inc..

Sterling, T. L. (2002). **Beowulf cluster computing with Linux.** MIT press.

Sudha, & Nisha. (2015). **A Survey of Various Fault Tolerance Checkpointing Algorithms in Distributed System.** International Journal of Advanced Networking and Applications, 7(2), 2682–2689.

Wikipedia. (2016, April 10). **SHA-1** [Encyclopedia]. Retrieved April 12, 2016, from https://en.wikipedia.org/wiki/SHA-1

Xu, L. (2015a). **Spark Internals: Cache and Checkpoint.** Retrieved May 15, 2016, from https://github.com/JerryLead/SparkInternals/blob/master/markdown/english/6-CacheAndCheckpoint.md

Xu, L. (2015b). **Spark Internals: Overview.** Retrieved March 23, 2016, from https://github.com/JerryLead/SparkInternals/blob/master/markdown/english/1-Overview.md

Yan, Y., Gao, Y., Chen, Y., Guo, Z., Chen, B., & Moscibroda, T. (2016, October). **Tr-spark: Transient computing for big data analytics.** In Proceedings of the Seventh ACM Symposium on Cloud Computing (pp. 484-496). ACM.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). **Spark: Cluster Computing with Working Sets.** In Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (pp. 10–10). USENIX Association.

Zaharia, M., Chowdhury, M., Tathagata, D., Ankur, D., Ma, J., McCauley, M., … Stoica, I. (2012a). **Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.** In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (pp. 2–2). USENIX Association.

Zaharia, M., Das, T., Li, H., Shenker, S., & Stoica, I. (2012b). **Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters.** USENIX. Retrieved from https://www.usenix.org/conference/hotcloud12/workshop-program/presentation/zaharia

Zaharia, M. (2015). **Apache Spark: Preparing for the Next Wave of Reactive Big Data.**

Zhu, W., Chen, H., & Hu, F. (2016, January). **ASC: Improving spark driver performance with automatic spark checkpoint.** In Advanced Communication Technology (ICACT), 2016 18th International Conference on (pp. 607-611). IEEE.

# ภาคผนวก ก

## รหัสต้นฉบับของกรอบงาน Distributed Test Checkpointing

```
package org.apache.spark.cprdd

import java.io.{PrintWriter, StringWriter}

import com.typesafe.config.ConfigException
import org.apache.spark.rdd.RDD
import org.apache.spark.util.CollectionAccumulator
import org.objectweb.asm.ClassReader
import org.objectweb.asm.tree.{ClassNode, MethodNode}
import org.objectweb.asm.util.{Textifier, TraceMethodVisitor}
import th.ac.sut.aiyara.sparktest.utils.ConfigSpark

import scala.collection.immutable.::
import scala.reflect.ClassTag

trait HashRDD extends ConfigSpark{
  def traceMethod(mnode:MethodNode):String ={
    var strwrt:StringWriter =new StringWriter()
    if (mnode.name !="<init>"&& mnode.name !="<cinit>"){
      strwrt.append("Method name :"+mnode.name)
      val p =new Textifier
      val tracer=new TraceMethodVisitor(p)
      mnode.accept(tracer)
      val w =new PrintWriter(strwrt)
      p.print(w)
      w.flush()
    }
    strwrt.toString
  }

  def getClassFunctionHash(_class:Class[_ <:AnyRef]):String ={
    import scala.collection.JavaConversions._
    val classReader =getASMClassReader(_class)
    val cn:ClassNode =new ClassNode

    classReader.accept(cn,0)
    var readStringFromBytecode:StringBuilder =new StringBuilder()
    for(method <-cn.methods.toList){
      val traced =traceMethod(method.asInstanceOf[MethodNode])
      readStringFromBytecode.append(traced)
    }

    val clsName =
_class.getName.replaceAll("\\.","\\\\/").replaceAll("\\$","\\\\\\$")

    var arrayOfStringInBytecodeLineByLine =readStringFrom-
Bytecode.toString().split("\\n")
    arrayOfStringInBytecodeLineByLine =filterLin-
eNumber(arrayOfStringInBytecodeLineByLine)
  arrayOfStringInBytecodeLineByLine =filterLocalVaria-
```

```
ble(arrayOfStringInBytecodeLineByLine)
    arrayOfStringInBytecodeLineByLine = removeReferences-
Class(clsName,arrayOfStringInBytecodeLineByLine)
    removeAnnonymousFunction(arrayOfStringInBytecodeLineByLine)
  }

  def filterLocalVariable(readStringFromBytecode: Array[String]): Ar-
ray[String] = {
    readStringFromBytecode.filter(x => !x.trim.startsWith("LOCALVARIABLE"))
  }

  def filterLineNumber(readStringFromBytecode: Array[String]): Ar-
ray[String] = {
    readStringFromBytecode.filter(x => !x.trim.startsWith("LINENUMBER"))
  }

  def removeReferencesClass(clsName: String, readStringFromBytecode: Ar-
ray[String]): Array[String] = {
    val removed = readStringFromBytecode.map(x =>
x.replaceAll(clsName, ""))
    removed
  }

  def removeAnnonymousFunction(readStringFromBytecode: Array[String]):
String = {
    readStringFrom-
Bytecode.mkString.replaceAll("\\$\\$anonfun\\$apply\\$\\d", "")
  }

  def getASMClassReader(cls: Class[_]): ClassReader = {
      val clsName = cls.getName.replaceFirst("^.*\\.", "") + ".class"
      val resourceStream = cls.getResourceAsStream(clsName)
      new ClassReader(resourceStream)
  }

  def getCleanedFucntionClass(head: (String, AnyRef)): String = {
    val s = getFieldsInObject(head._2)
  .filter(p => p._1.startsWith("cleanF") || p._1.startsWith("cleanedF"))
  .map(x => getClassFunctionHash(x._2.getClass))
  .mkString("|")
    hashToString(s)
  }

  def getDependencyHash[T:ClassTag](rddForRecursion: RDD[T]): String = {
    val listElementsInObject = getFieldsInObject(rddForRecursion)
    def recurringHead(listElementsInObject: List[(String, AnyRef)], acc:
List[String]): List[String] = listElementsInObject match {
      case head :: tails => {
        val hashFromClass: String = head._1 match {
    case "prev" => getDependencyHash(head._2.asInstanceOf[RDD[T]])
```

```
          case "f"=> getCleanedFucntionClass(head)
          case _ => getClassHash(head._2.getClass)
        }
        recurringHead(tails,acc :+hashFromClass)
      }
      case Nil => acc
    }
    recurringHead(listElementsInObject,List()).sorted.mkString
  }

  protected def getRootRdd[T:ClassTag](checkRootRdd: RDD[T]): RDD[T]= {
    def recurringHead(rddHead: RDD[T]): RDD[T]= {
      val retRddHead: RDD[T]=rddHead.firstParent match {
    case firstParentRdd: RDD[T]=> {
          if(checkPreviousRdd(firstParentRdd)){
            recurringHead(firstParentRdd)
          }else {
            firstParentRdd
          }
        }
        case _ => rddHead
      }
      retRddHead
    }

    def checkPreviousRdd[U: ClassTag](firstParentRdd: RDD[U]): Boolean = {
      val listElementsInObject =getFieldsInObject(firstParentRdd)
      listElementsInObject.map(x => x._1).contains("prev")
    }

    val rootRdd =recurringHead(checkRootRdd)
    rootRdd
  }

  protected def getSignature[T: ClassTag](rddForSignature:
RDD[T],hashInput: Boolean =false): String ={
    val hashArray =new CollectionAccumulator[String]
    if(hashInput){
      val sc =rddForSignature.sparkContext
      sc.register(hashArray, "Hash array accumulator")
      val hasher =hash match {
        case "sha1"=> "SHA-1"
        case "sha256"=> "SHA-256"
        case _ => "MD5"
      }
      getRootRdd(rddForSignature).foreachPartition(p => {
        val partitionData =p.mkString
        val hash =java.security.MessageDigest.getInstance(hasher)
        hash.update(partitionData.getBytes())
        val hashal =new java.math.BigInteger(1, hash.digest()).toString(16)
        hashArray.add(hashal)
```

```
   })
  }else{
   hashArray.add("<NOT_HASH_INPUT_PARTITION>")
  }
  val sortedSignature =hashArray
 .value
 .toArray
 .sortWith(_.asInstanceOf[String]< _.asInstanceOf[String])
 .mkString
  val hashed =getDependencyHash(rddForSignature)
  val hashLv1 =(hashed +sortedSignature).getBytes
  val hashLv2 =hashToString(hashLv1)
  hashLv2
 }

 def getFieldsInObject(f:AnyRef):List[(String,AnyRef)]={
  val fields =f.getClass.getDeclaredFields
  val prepareField =fields
 .filter(_.getName !="serialVersionUID")
 .map(_f => {
    _f.setAccessible(true)
 (_f.getName,_f.get(f))
   }).filter(_ !=null)
  prepareField.toList
 }

 def getClassHash(_class:Class[_]):String ={
  val className =_class.getName.replaceFirst("^.*\\.", "")+".class"
  val resourceStream =_class.getResourceAsStream(className)
  var hashed =""
  if (resourceStream !=null){
   val byteArrayOfClass =
org.apache.commons.io.IOUtils.toByteArray(resourceStream)
  hashed =hashToString(byteArrayOfClass)
  }
  hashed
 }

 def hashToString(stringOfClass:String):String ={
  val byteOfClass =stringOfClass.getBytes()
  hashToString(byteOfClass)
}

 def hashToString(byteOfClass:Array[Byte]):String ={
  var hashedString:String =""
  try {
    hash match {
    case "sha1"=> hashedString =sha1hasher(byteOfClass)
    case "sha256"=> hashedString =sha256hasher(byteOfClass)
  case _ => hashedString =md5hasher(byteOfClass)
```

```
      }
    } catch {
      case e:ConfigException => {
        LoggerObj.log.warn(s"Hash message digest not set :
${e.getMessage} use MD5 by default")
        hashedString = md5hasher(byteOfClass)
     }
    }
 hashedString
  }

  def sha256hasher(byteOfClass: Array[Byte]): String = {
    val md = java.security.MessageDigest.getInstance("SHA-256")
    md.digest(byteOfClass).map("%02x".format(_)).mkString
  }

  def sha1hasher(byteOfClass: Array[Byte]): String = {
   val md = java.security.MessageDigest.getInstance("SHA-1")
    md.digest(byteOfClass).map("%02x".format(_)).mkString
  }

  def md5hasher(byteOfClass: Array[Byte]): String = {
    val md = java.security.MessageDigest.getInstance("MD5")
    md.digest(byteOfClass).map("%02x".format(_)).mkString
  }
}
```

```
package org.apache.spark.cprdd

import org.apache.hadoop.fs.Path
import org.apache.spark.sql.{Dataset, Encoder}
import th.ac.sut.aiyara.sparktest.utils.ConfigSpark
import com.databricks.spark.avro._

import scala.reflect.ClassTag

/**
 * Created by Bhuridech Sudsee.
 */
trait ImplicitDatasetCheckpoint {

  implicit class ImplicitDatasetCheckpoint[T: ClassTag](val ds:
Dataset[T]) extends HashRDD with ConfigSpark {
    var encoder: Encoder[T] = _

    def dtCheckpoint(_encoder: Encoder[T], hashInput: Boolean = false): Da-
taset[T] = {
      encoder = _encoder
      val serializerResult = keySerializer match {
  //case "json"=> doCheckpointWithJson(hashInput)
```

```
     case "avro" => doCheckpointWithAvro(hashInput)
        case "avro-compress" => doCheckpointWithAvroCompres-
sion(hashInput)
        case _ => doCheckpointWithParquet(hashInput)
      }
      serializerResult
    }

    def doCheckpointWithAvroCompression(hashInput: Boolean): Dataset[T] = {
      spark.conf.set("spark.sql.avro.compression.codec", "deflate")
      spark.conf.set("spark.sql.avro.deflate.level", avroOption)

      val cpDir = getPath("avro-compress",hashInput)
      var checkpointedDS = ds
     if (checkPathExists(cpDir)) {
        LoggerObj.log.debug("Read file")
        checkpointedDS = spark.read.avro(cpDir).as(encoder)
      } else {
        checkpointedDS.write.avro(cpDir)
      }
      checkpointedDS
    }

    def doCheckpointWithAvro(hashInput: Boolean): Dataset[T] = {
      val cpDir = getPath("avro",hashInput)
      var checkpointedDS = ds
      if (checkPathExists(cpDir)) {
        LoggerObj.log.debug("Read file")
        checkpointedDS = spark.read.avro(cpDir).as[T](encoder)
      } else {
       LoggerObj.log.debug("Write file")
        checkpointedDS.write.avro(cpDir)
      }
      checkpointedDS
    }

    def doCheckpointWithJson(hashInput: Boolean): Dataset[T] = {
      val cpDir = getPath("json",hashInput)
      var checkpointedDS = ds
      if (checkPathExists(cpDir)) {
        LoggerObj.log.debug("Read file")
        checkpointedDS = spark.read.json(cpDir).as(encoder)
      } else {
        LoggerObj.log.debug("Write file")
        checkpointedDS.write.json(cpDir)
      }
      checkpointedDS
    }

    def doCheckpointWithParquet(hashInput: Boolean): Dataset[T] = {
      val cpDir = getPath("parquet",hashInput)
```

```
      var checkpointedDS = ds
      if (checkPathExists(cpDir)) {
        LoggerObj.log.debug("Read file")
        checkpointedDS = spark.read.parquet(cpDir).as(encoder)
      } else {
        LoggerObj.log.debug("Write file")
        checkpointedDS.write.parquet(cpDir)
      }
      checkpointedDS
    }

    def getPath(postfix: String, hashInput: Boolean): String = {
      val sparktestPath = sc.getConf.get("sparktest.test.lineage.path")
      val cpDir: String = sparktestPath + getSignature[T](ds.rdd, hashInput)
+"-"+postfix
      cpDir
    }

    private def checkPathExists(_path: String): Boolean = {
  val path = new Path(_path)
      val fs = path.getFileSystem(sc.hadoopConfiguration)
      fs.exists(path)
    }
  }
}
```

# ภาคผนวก ข

## บทความทางวิชาการที่ได้รับการตีพิมพ์เผยแพร่ในระหว่างการศึกษา

# บทความทางวิชาการที่ได้รับการตีพิมพ์เผยแพร่ในระหว่างการศึกษา

Sudsee, B., & Kaewkasi, C. (2018). **A Productivity Improvement of Distributed Software Testing Using Checkpoint.** International Conference on Advanced Communications Technology (ICACT), pp. 78–84.

# A Productivity Improvement of Distributed Software Testing using Checkpoint

Bhuridech Sudsee, Chanwit Kaewkasi
*School of Computer Engineering*
*Suranaree University of Technology, Nakhon Ratchasrima, Thailand, 30000*
m5741861@g.sut.ac.th, chanwit@sut.ac.th

***Abstract*—The advancement of storage technologies and the fast-growing number of generated data have made the world moved into the Big Data era. In this past, we had many data mining tools but they are inadequate to process Data-Intensive Scalable Computing workloads. The Apache Spark framework is a popular tool designed for Big Data processing. It leverages in-memory processing techniques that make Spark up to 100 times faster than Hadoop. Testing this kind of Big Data program is time consuming. Unfortunately, developers lack a proper testing framework, which cloud help assure quality of their data-intensive processing programs, while saving development time.**

**We propose *Distributed Test Checkpointing (DTC) for Apache Spark*. DTC applies unit testing to the Big Data software development life cycle and reduce time spent for each testing loop with checkpoint. From the experimental results, we found that in the subsequence rounds of unit testing, DTC dramatically speed the testing time up to 450-500% faster. In case of storage, DTC can cut unnecessary data off and make the storage 19.7 times saver than the original checkpoint of Spark.**

***Keywords*—Distributed Checkpointing; Apache Spark; Big Data Testing; Software Testing;**

## I. INTRODUCTION

The increasing and diversity of electronic devices, sensors, IoT devices and the fast-growing numbers of Internet users have been generating tremendous amount of data recently. They are not only the large amount of data but their structures are also complex as well. This complexity makes the traditional data mining tools inadequate to manage today's data [1].

The MapReduce [2] programming model has induced the development of many frameworks such as Apache Hadoop [4], Map-reduce-merge [5] and Apache Spark [6], which aim to process data intensive tasks. Developers only need to rewrite their programming logic in the form of *map* and *reduce* functions in order to process data on a MapReduce framework. These functions will be automatically managed by the framework's default configuration. This mechanism makes the MapReduce framework easy to use. At its simplest form, a MapReduce program usually starts by a *map* function creating key/value pairs from the input. These intermediate key/value pairs are then passed to a *reduce* function to produce the final results. The MapReduce model is parallel by nature. It is designed to allow developers to run MapReduce programs for high performance computing jobs using a commodity cluster, built from low-cost hardwares. With this kind of the cluster architecture, we can handle massive amount of data and process them on numerous cluster nodes without a single point of failure [3].

Although the MapReduce model is easy to use for software development, but it is quite tricky to test software written by the MapReduce model. Software testing is a vital part of the development process. Testing is usually 25-50% of the overall cost [8]. We found that the current mechanism is not enough to assure quality for Big Data processing programs. Unit testing is a software testing technique which properly leads to better levels of quality. However, tools like Scalatest[9] or jUnit[10] have their own limitations to use with a MapReduce framework like Spark. For example, SparkContext and SparkSession objects must be instantiated only once for each running Java Virtual Machine (JVM) to avoid unexpected testing results [12]. Spark-testing-base [11] also does not have a testing mechanism for Spark. Without modification, it cannot work on a Spark cluster because if its inability to distribute class files across worker nodes. There aforementioned techniques are not suitable for Spark simply because they are not designed to test programs that distributelly process large amount of data.

Test-driven development (TDD) is a software development technique that helps developers to focus on writing a specific test at a time. It additionally allows code improvement while preserving correctness according to the specification. TDD workflow consists of the following steps, (1) writing a minimum test (2) writing codes to just make the test passed, and (3) refactoring to remove unnecessary codes while still making the current test passed [13]. We call these steps a TDD workflow herein this paper. Applying TDD to data intensive programs is difficult due to the nature of workloads, which need to process on a cluster. So, developers require a special tool to help shorten each loop of the TDD workflow.

Spark has *cache*, *persist* and *checkpoint* methods to help mitigate job failure. These mechanisms however do not help software testing process much. The main reason is that a cluster state cached or persisted by them does not survive across runs of JVMs. A cluster state saved by the *checkpoint* method does survive on disk but unfortunately it cannot be retrieved back by a newly started JVM [14, 15].

In this paper, we present Distributed Test Checkpointing (DTC), a technique that leverages the checkpoint technique to enhance software testing for data intensive jobs. With DTC, developers can increase productivity when testing their software on a distributed cluster repeatedly. DTC applied a hash function on each data partition of a Resilient Distributed Datasets (RDD) [18] to use an identifier. Modification of an RDD or a Dataset can be traced by the hashed number. The testcase that uses the RDD is also hashed at the bytecode level. Combining these techniques, DTC is found to reduce testing time and storage required by checkpointing significantly compared to the original Spark's checkpointing technique.

The remaining of this paper is organized as followed. Section II discusses related works, including Apache Spark. Section III presents the design and internal mechanism of DTC. Section IV presents the system architecture of the cluster used by our experiments, and the experimental results. This paper then ends with conclusion and future works in Section V.

## II. BACKGROUND AND RELATED WORK

### A. Apache Spark

Spark is a data intensive processing framework focusing on in-memory data processing [6], which is implemented in the form of Resilient Distributed Dataset (RDD) [18]. RDD is designed to take care of the data flow and handle the processing mechanism. An RDD could be created using one of the following methods (1) reading data from file (2) parallelizing collection in the driver program (3) transforming from another RDD (4) and by transforming back from a persisted RDD [6]. An RDD comprises with two kinds of command, *transformations* and *actions*. A transformation command transforms an RDD to another RDD. These commands are *map*, *filter* and *groupByKey*, for example. Another set of commands are actions, which are *collect* and *count*, for example. An RDD keeps all previous transformation inside itself. This direct acyclic graph of transformation is known as *lineage*. The beginning of the real computation occurs only when an action is called. This is the lazy evaluation nature of Spark.

A mechanism for failure recovery that helps an RDD to resume the processing without re-computation from scratch are methods such as *cache*, *persist* and *checkpoint*. The *cache* method uses persistency at MEMORY_ONLY, while the *persist* method has several levels of persistency. The checkpoint method, in contrast, uses the technique which save data onto a reliable storage, such as HDFS, Amazon S3 or Ceph. An RDD is usually cached or persisted during its computation to avoid re-computation previous steps [15].

The checkpoint technique is also applicable for Spark Streaming because it truncates the internal lineage, so the RDD does not need to knowledge of its parent. However, this mechanism is not designed for software testing. The re-computation is still required to start from the beginning when the testcase is re-run. The rerunning of the testcase destroys a Block Manager inside an Executor. This Block Manage is responsible for keeping cached and persisted data. The new Driver program and the testcase therefore is not able to access the location of checkpoints.

In addition, Spark has introduced the Dataframe API in 1.3 and Dataset in 1.6. Both abstractions can be used interchangeably because Dataset[Row] is the type safer version of DataFrame. A dataset is also convertible to an RDD. In the case of DTC proposed in this paper, we read and write data directly without triggering any computation of related RDDs.

### B. Debugging framework for Spark

A technique used to improve quality of the software is debugging. Developers usually debug to observe certain set of variables they are interested. However, in the Data-intensive Scalable Computing (DISC), the debugging process is difficult as data are computed distributedly on a cluster.

*BigDebug [7]* is a tool designed to helps Spark's developers deal with debugging a Big Data program. There is a downside that the tool requires user's interaction during the debugging process. Those interactions make the debugging more difficult than those of normal programs because the Big Data programs are distributed by nature. Moreover, a BigDebug program cannot tackle the problem when the RDD being debug requires changes. The whole debugging process needs to start over in that case. In case of the developer changing codes on-the-fly, the RDD will become in-consistent as some partitions of the RDD has been processed by the old version of codes, while other partitions will be processed by the new codes. BigDebug support Spark up to 1.2.1 as the time writing.

### C. Checkpoint implementation for Spark

Researchers have been employed the checkpoint of Spark in many ways to improve its efficiency, as follows.

*Flint [26]* was created atop the original checkpoint technique of Spark. It aims at applying checkpoint and store their data on transient instances to reduce the VM usage cost. A transient instance in a kind of low-cost computing unit, which can be recalled anytime by its cloud provider. Flint solves this problem by writing an RDD's partitions to an HDFS, which is operated on on-demand instances. We found that this implementation lacks a mechanism to prevent re-calculation when JVM is terminated. In addition, their checkpoint will be saved automatically so developers need to prepare a huge amount of space in order to prevent the full of storage, which can lead to the failure of the whole system.

*TR-Spark [27]* implements the similar approach as Flint. The difference is that TR-Spark allows fined-granularity checkpoints at task-level. By leveraging this level of checkpoints, the storage usage cloud be reduced in comparison to checkpoint the whole RDD. However, TR-Spark makes it difficult to use as developers need to collect the information of VM failure to let it know the failure probability. TR-Spark does not deal with changes of the Driver program.

*Automatic Spark Checkpointing (ASC) [25]* was designed to help analyze the trade-off between RDD checkpointing and its restore. ASC performs this computation by estimating them from an RDD lineage. Nevertheless, this technique does not support checkpoint across JVM termination. It also lacks the ability to recognize the similarity or identity of an RDD.

*Spark-flow [24]* aims to mitigate the effect of JVM termination for checkpoint restoration. It makes use of Distributed Collection (DC), a library similar to the Dataset API. DC is able to analyze an RDD at the bytecode level with ASM. It can identify the location of checkpoint calls, inside an anonymous function. It also uses the MD5 hash function to help detect changes at the bytecode level. However, DC has some downside as the following. First, when calling checkpoint on a DC, the data is re-read again after checkpointing. Second, when restoring from checkpoint, the action *count* will be triggered, so the re-computation kicks in. Finally, computation is mainly done on the Driver machine, so the mechanism is actually not distributed. This often causes Out-of-Memory exception inside the Driver program and it stops working.

```
1  val data = sc.parallelize(Array(1,2,3,4,5))
2  val distData = data.map(x => (x,1))
3  distData.dtCheckpoint()
4  distData.count()
5  distData.collect()
```

**Figure 1** Example of a dtCheckpoint call on an RDD

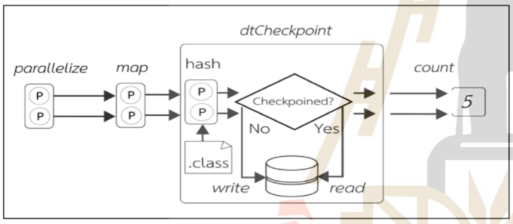


**Figure 2** The dtCheckpointing mechanism inside DTC

## III. DESIGN AND IMPLEMENTATION

Spark stores the RDD transformations in the form of a lineage graph a.k.a. the logical execution plan. When an action is triggered for a certain RDD, its job will be submitted to the DAG Scheduler to transform the RDD's lineage into a directed acyclic graph, whose a *vertex* is an RDD partition and *edge* is a transformation. After that the staging process will be kicked in. This staging process will be started from the final action going backwards to the beginning of the RDD. However, in the real execution, the process will be performed from the beginning of the RDD forwardly to the final action. After the staging, the system obtains a set of Stages and Tasks.

A checkpoint of an RDD however must be done before the first action is performed. From the source code in the Figure 1, when a program starts to process an array of integer 1 to 5, the array will be passed as a parameter of method *parallelize* of class SparkContext. This result in a ParallelCollectionRDD stored in variable *data.* At line 2, each element from the *data* RDD is mapped with 1 using the *map* method as a key/value pair. The result is a MapPartitionsRDD stored in variable *distData*. At line 3, method *dtCheckpoint* is invoked. Please note that the original Spark and DTC both use the lazy evaluation mechanism, this means that the checkpoint method only marks at a certain point over the DAG, where checkpoints will happen there. At line 4, command *distData.count()* is the first action. When this first action is triggered, the checkpoint is not yet created. The computation then is started from the beginning of the RDD to the mark point. After that, the checkpoint is stored at the 1st upper directory level as a hash value generated by the mechanism of DTC. At the line no 5, method *distData.collect()* is invoked as the second action. The system will then check backwards from the action to the beginning of the RDD. This time the system will find a checkpoint already existed because there is a directory whose name matches with the hash. When the DAG Scheduler starts to transform the lineage, it uses the data directly from the checkpoint without re-computation. Please also note that action *count()* and *collect()* belong to the different jobs. The result computed by *count()* will not be included as an input for *collect(),* despite their order of execution.

### *A. DtCheckpointing*

This mechanism works when the method *dtCheckpoint* of an RDD or a DataSet is called. This call marks an RDD and also starts the Hashing RDD mechanism to obtain a directory path from hash transformation. If there is no directory matched the hash value, it means that the system never created that checkpoint. After the creation of the directory content of the RDD will be stored inside of it. But if the directory exists, the system will read the content as the data of the RDD. In Figure 2, when an RDD is created using the *parallelize* method and is transformed with *map* followed by an invocation of *dtCheckpoint*. The sub-system DtCheckpointing kicks in to mark points in the RDD for later storing when action *count* is called.

We usually perform the test on a Spark Cluster with SBT, which is an interactive build tool to help develop software with Java or Scala. SBT allows us to write a build file using Scala-based Domain Specific Language. It manages a program dependency with Apache Ivy. With DTC, we modify test commands of the SBT namely *test*, *test-only*, and *test-quick* to support not only the local execution but also in the real working cluster. We solve the problem of *ClassNotFoundException* and *NoClassDefFoundError* by making a fat jar via custom SBT task. So, we introduce *testOnCluster* for testing every testcase, *testOnlyOnCluster* to test a specific testcase, and *testQuickOnCluster* to test a certain testcase which may be failed from last time, or never tested or need re-computation. Our modification to SBT allows the new mode of testing on the real cluster.

### *B. Hashing an RDD*

Hash function is a one-way function which can be used to check data modification. Eve one bit of data is changed this function notices that modification. In this paper, we will compare MD5, SHA-1 and SHA-256 because these algorithms have various speed of hash and resource usage.

**TABLE 1.** FEATURE COMPARISON BETWEEN CONFIGURATIONS

| Method | Failure tolerance | More abstraction layer | Prevent re-calculation from beginning | Suitable for Testing | Cluster |
|---|---|---|---|---|---|
| No-Checkpoint | No | No | No | No | Yes |
| Spark Original | Yes | No | Yes | Not Suitable | Yes |
| Spark-flow | Yes | Yes | Yes | Yes | No |
| DTC | Yes | No | Yes | Yes | Yes |

**TABLE 2.** THE COMBINATION OF ALL EXPERIMENTAL CONFIGURATIONS

| Configuration | Type | | | Checkpoint Data Format | | | | Hash Algorithm | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | RDD | DataSet | DC | Java | Kryo | Avro | Parquet | MD5 | SHA1 | SHA256 |
| No-checkpoint | √ | √ | - | - | - | - | - | - | - | - |
| Spark Original | √ | √ | - | √ | - | - | - | - | - | - |
| DTC | √ | √ | - | √ | √ | √ | √ | √ | √ | √ |
| Spark-flow | - | - | √ | - | - | - | √ | √ | - | - |

This technique of the DTC framework is able to track the change of an RDD because the generated transformations. So we can use this mechanism to detect modification of any transformation back to the original RDD. When an action is triggered, the DTC framework detects all RDD dependencies and prepares a clean bytecode available by the CleanF property of the RDD, following by preparing other Java bytecode's files which related to the dependencies. In preparation stage, DTC uses ASM, a tool to manage a Java bytecode [17], which Scala internally uses it for the compilation mechanism. With a ASM, the DTC's hashing an RDD mechanism can access Java class file at runtime and de-serialize them for reverse engineering propose. DTC needs to remove some brittle information such as LINENUMBER or serialVersionUID from a class file. With this information filtered out, we can detect changes of an RDD or DataSet even when the line numbers have been changed.

The result of class file analysis in preparation stage, after unnecessary dependencies was eliminated, these dependencies will compute hash number and input data, which the origin of an RDD will compute hash number also. The computation is distributed computing with Spark's accumulator in the first level hash number computation will compute hash number of input data for every partition, and then collect and reorder result because unpredictable computation time. After that, the DTC will compute hash number of sorted hash number again. Figure 3, illustrates the steps of hashing mechanism please note that the computation of input data is an option that can specify with *dtCheckpoint(true)*.

```
SET hash_array = empty array of string
IF (HASH_INPUT_DATA = true) THEN
  READ each data partition from (RDD or DataSet)
  COMPUTE hash of each data partition
  APPEND hashes to hash_array
ENDIF
COMPUTE hash_array ASC
CALL HASH_DEPENDENCY of (RDD or DataSet)
```

**Figure 3** Pseudo codes of the mechanism of Hashing an RDD

## IV. EXPERIMENTS

### *A. Cluster configuration*

The experiments presented in this paper have been conducted on a Spark cluster consisted of 10 nodes. Each node is an Intel Core i5-4570 Quad-core with 4 GB of RAM. The drive node is an Intel Xeon E5-2650V3 Deca-core with 8GB of RAM. We use Apache Spark 2.0 for the experiments along with Ceph as the distributed file system over these 10 nodes. The Ceph storage is 10 TB. The system architecture is illustrated in Figure 4.

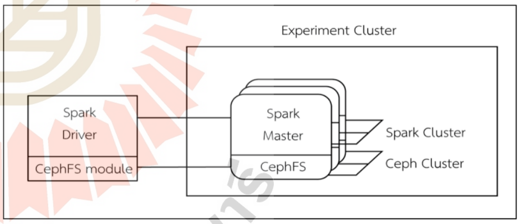


**Figure 4** The cluster architecture used by the experiments

### *B. Methodology*

Table 1 shows the comparison of checkpoint mechanism properties. If we do not use checkpoint, the system does not have the failure tolerance property. If we use the original Spark, it is not suitable for testing because its checkpoint mechanism does not work well in the test environment. In case of Spark-flow it does not work on the cluster environment out-of-the-box. DTC, on the other hand, is designed to address these problems in the testing environment. For the experiments, we use a MapReduce program to count words on a 31 GB Wikipedia dataset. The Wordcount program will split sentences into words and count them using an RDD with different checkpoint mechanisms. We tested 10 times continuously for each checkpoint mechanism.

Table 1 shows a brief differentiation of comparison method that we will experiment. That meant, if we have no checkpoint it will lack failure tolerance, the Spark original checkpoint insufficient to testing. The Spark-flow push developer in more abstraction layer by create a higher level of an DataSet and it not work on cluster naturally. In Table 2, we show the combination of all experimental configurations. Accordingly, the DTC introduce to rectify that plain. We will compare with MapReduce Wordcount algorithms on Wikipedia 31 GB with separate each word from each other with white space. And then, we filter only word occurred more than 10 million times, after that assert with the most word occurred. We consecutively repeat these steps 10 cases.

### *C. Experimental results*

From the experiments, we start discussing in the case of no hashing input data, denoted not-hashinput. In this case the input will not be verified by hashing functions before the program starts. We assume that development and during the tests. The experimental results are show in Figure 5. At the first run, DTC and the original checkpoint mechanism are all slow with insignificant difference. The DTC-java-sha1 is slowest. It uses 636 seconds slightly different from original-checkpoint. The no-checkpoint configuration does not have this startup overhead so it run at 136 seconds on average. For the first run, All DTC and the original checkpoint are 4.7 times or slower than the no-checkpoint mechanism. However, all DTC configurations are significantly faster in the subsequence runs.

Figure 6 shows the comparison between cases of applying hash functions over input data to allow the system to detect changes of the input. It shows that DTC mechanisms are slower than no-checkpoint and original-checkpoint only in the first run. In the subsequence runs, DTC mechanisms make the test s faster than those run by no-checkpoint and original-checkpoint. We found that DTC-kryo-sha1 is slowest in the first run. It uses 908 seconds on average, while no-checkpoint uses 136 seconds and original-checkpoint use 636 seconds. In the subsequence runs, DTC mechanism uses around 85 seconds on average. It is significantly faster that both no-checkpoint and original-checkpoint, which is 60%

In the first run with hash input, the fastest DTC mechanism is DTC-java-sha256 it is 480% slower than no-checkpoint and 24% slower than original-checkpoint. In the subsequence runs, this mechanism is 40% faster than no-checkpoint and 590% faster than original-checkpoint. Other cases are in similar trends.

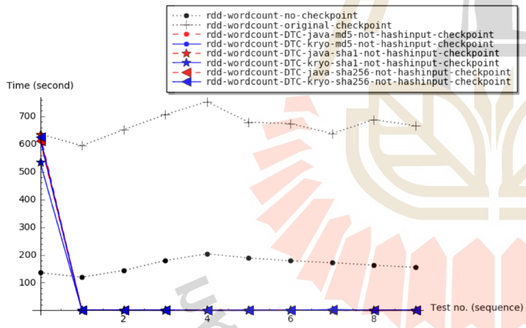


**Figure 5** Comparison of checkpoint time of RDDs without hashing inputs

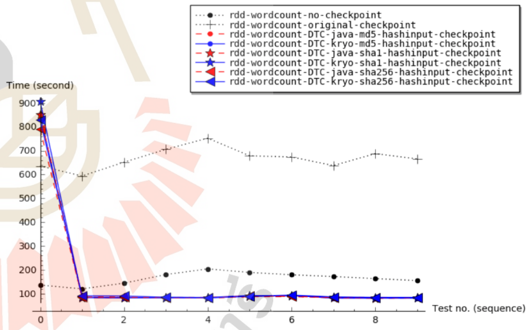


**Figure 6** Comparison of checkpoint time of RDDs with hashing inputs

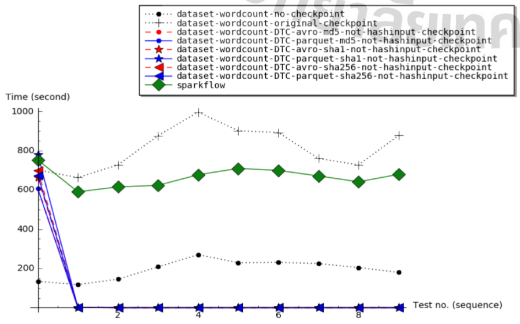


**Figure 7** Comparison of checkpoint time of DataSet, including Spark-flow without hashing inputs

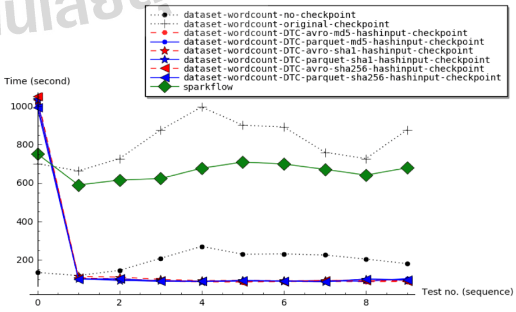


**Figure 8** Comparison of checkpoint time of DataSet, including Spark-flow with hashing inputs

In case of DataSet, we found the similar trends as the case of RDD. During the first run DTC mechanisms are slowest, and significantly faster in subsequence runs. Figure 7 and 8 show the comparison between checkpoint mechanisms for the DataSet without hashing input and with hashing input, respectively. We also include Spark-flow in these experiments. We found that Spark-flow uses 752 seconds at the first run, while DTC-parquet-md5 uses 606 seconds, so DTC is 24% faster than Spark-flow. In case of hash input data, DTC is 40% slower than Spark-flow for the first run. However, in the subsequence runs, DTC dramatically reduces time spending, according aforementioned trends.

The mechanism of checkpoint usually requires use of storage. The storage usage comparison is then presented in Table 3. According to the table, DTC with Java serializer uses the storage only one-tenth of those used by the original Spark checkpoint. In case of DTC with Kryo, it uses storage only 5% of the original checkpoint.

This storage usages are similar for DataSet. According to table 3, DTC with Avro format uses only 10% of the original storage. In case of DTC with Parquet format, it uses only 11% of the original storage. Comparison of these results with Spark-flow, we are roughly at the same ration.

DTC is designed to allow re-usability of RDDs and DataSets. It can traverse and detect change of the dependency of each RDD or a DataSet. From the experiments, we have found that DTC has a larger overhead than the mechanism of the Original Spark only when a testcases are in first run. When the testcases are in the later runs, DTC makes them 5-6 times faster than running by the Original Spark and Spark-flow. Moreover, DTC uses disk space 8-9 times less than both implementations as shown in Table 3 and Table 4.

**TABLE 3.** CHECKPOINT'S STORAGE USAGE OF AN RDD

| Storage usage | Size | Unit |
|---|---|---|
| No-checkpoint | 0 | MB |
| Spark original checkpoint | 9.870 | MB |
| DTC-java-with-hash | 0.987 | MB |
| DTC-java-without-hash | 0.987 | MB |
| *DTC-kryo-with-hash* | *0.501* | *MB* |
| *DTC-kryo-without-hash* | *0.501* | *MB* |

**TABLE 3.** CHECKPOINT'S STORAGE USAGE OF DATASET

| Storage usage | Size | Unit |
|---|---|---|
| No-checkpoint | 0 | MB |
| Spark original checkpoint | 9.860 | MB |
| *DTC-avro-with-hash* | *0.987* | *MB* |
| *DTC-avro-without-hash* | *0.987* | *MB* |
| DTC-parquet-with-hash | 0.993 | MB |
| DTC-parquet-without-hash | 0.993 | MB |
| Spark-flow | 9.930 | MB |

## V. CONCLUSIONS AND FUTURE WORK

There are still room of improvement and experiments. We would compare checkpoint mechanisms to observe the system behavior in case of VM termination, which is an important scenario to consider when choosing a testing tool. We still lack the experiments regarding high performance computing to compare cost and utilization. This subject is being studied.

## REFERENCES

[1] W. Fan and A. Bifet, "Mining big data: current status, and forecast to the future," in *ACM SIGKDD Explorations Newsletter*, 2012, vol. 14, pp. 1–5.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM - 50th anniversary issue: 1958 - 2008*, vol. 51, no. 1, pp. 107–113, 2008.

[3] B. Mark and B. Rajkumar, "Cluster Computing: The Commodity Supercomputer," in *Software-Practice and Experience*, 1999, vol. 29(6), pp. 551–576.

[4] "Welcome to Apache™ Hadoop®!" [Online]. Available: https://hadoop.apache.org/. [Accessed: 06-May-2017].

[5] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: Simplified Relational Data Processing on Large Clusters," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2007, pp. 1029–1040.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, 2010, pp. 10–10.

[7] M. A. Gulzar, M. Interlandi, T. Condie, and M. Kim, "BigDebug: Interactive Debugger for Big Data Analytics in Apache Spark," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2016, pp. 1033–1

[8] A. Spillner, T. Linz, and H. Schaefer, *Software testing foundations: a study guide for the certified tester exam*. Rocky Nook, Inc., 2014.

[9] "ScalaTest." [Online]. Available: http://www.scalatest.org/. [Accessed: 06-May-2017].

[10] "JUnit 5." [Online]. Available: http://junit.org/junit5/. [Accessed: 06-May-2017].

[11] "holdenk/spark-testing-base," *GitHub*. [Online]. Available: https://github.com/holdenk/spark-testing-base. [Accessed: 06-May-2017].

[12] "[SPARK-2243] Support multiple SparkContexts in the same JVM - ASF JIRA." [Online]. Available: https://issues.apache.org/jira/browse/SPARK-2243. [Accessed: 06-May-2017].

[13] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[14] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning spark: lightning-fast big data analysis*. O'Reilly Media, Inc., 2015.

[15] "JerryLead/SparkInternals," *GitHub*. [Online]. Available: https://github.com/JerryLead/SparkInternals. [Accessed: 07-May-2017].

[16] H. Karau and R. Warren, *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O'Reilly Media, Incorporated, 2017.

[17] E. Kuleshov, "Using the ASM framework to implement common Java bytecode transformation patterns," *Aspect-Oriented Software Development*, 2007.

[18] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 2–2.

[19] "A Universally Unique IDentifier (UUID) URN Namespace," *A Universally Unique IDentifier (UUID) URN Namespace*. [Online]. Available: https://www.ietf.org/rfc/rfc4122.txt. [Accessed: 07-May-2017].

[20] S. Saxena, *Getting Started with SBT for Scala*. Packt Publishing, 2013.

[21] "Home | Apache Ivy ™." [Online]. Available: https://ant.apache.org/ivy/. [Accessed: 07-May-2017].

[22] "sbt/sbt-assembly," *GitHub*. [Online]. Available: https://github.com/sbt/sbt-assembly. [Accessed: 07-May-2017].

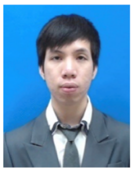
[23] "The Daily Build - write simple SBT task." [Online]. Available: http://blog.bstpierre.org/writing-simple-sbt-task. [Accessed: 07-May-2017].

[24] "bloomberg/spark-flow," *GitHub*. [Online]. Available: https://github.com/bloomberg/spark-flow. [Accessed: 08-May-2017].

[25] W. Zhu, H. Chen, and F. Hu, "ASC: Improving spark driver performance with automatic spark checkpoint," in *Advanced Communication Technology (ICACT), 2016 18th International Conference on*, 2016, pp. 607–611.

[26] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy, "Flint: batch-interactive data-intensive processing on transient servers," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, p. 6.

[27] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "TR-Spark: Transient Computing for Big Data Analytics," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, New York, NY, USA, 2016, pp. 484–496.



**Bhuridech Sudsee** received B.Eng. in Computer Engineering from Suranaree University of Technology and B.Sc. in Information Technology from Sukhothai Thammathirat Open University, both in Thailand. Currently, he is studying a Master degree in Computer Engineering. His fields of research interests are high-performance computing, distributed computing, data storage, Big Data processing and MapReduce frameworks.

**Chanwit Kaewkasi** received his PhD in Computer Science from the University of Manchester, United Kingdom in 2010. He is currently an Assistant Professor at School of Computer Engineering, Suranaree University of Technology, Thailand. Dr. Kaewkasi is actively researching in the areas of Low-Power Clusters, Cloud Computing, Big Data and Software Container Technologies.

Technically Co-sponsored

*The 20th International Conference on*
**Advanced Communications Technology**
**(http://www.icact.org )**

**Feb. 11 ~ 14, 2018, Elysian Gangchon, Republic of Korea**

# Certificate of Outstanding Paper Award

**1st Author:** Mr. Bhuridech Sudsee

**Affiliation:** Suranaree University of Technology

**Paper ID:** 20180097

**Title:** A Productivity Improvement of Distributed Software Testing using Checkpoint

**Author(s):** Bhuridech Sudsee, Chanwit Kaewkasi

**Affiliation(s)** Suranaree University of Technology, Thailand

**This award is presented to Mr. Bhuridech Sudsee for his/her excellent achievement in the ICACT2018 International Conference hosted by the Global IT Research Institute with IEEE Communication Society.**

**12-Feb-18**

**Prof. Dae-Young Kim**

**ICACT2018 TPC Chair**

# ประวัติผู้เขียน

นายภูริเดช สุดสี เกิดเมื่อวันที่ 29 มกราคม พ.ศ. 2535 ที่อำเภอสุวรรณคูหา จังหวัดหนองบัวลำภู สำเร็จการศึกษาระดับปริญญาตรี วิศวกรรมศาสตรบัณฑิต (วิศวกรรมคอมพิวเตอร์) จากมหาวิทยาลัยเทคโนโลยีสุรนารี จังหวัดนครราชสีมา และสำเร็จการศึกษาวิทยาศาสตร์บัณฑิต (การจัดการเทคโนโลยีสารสนเทศและการสื่อสาร) จากมหาวิทยาลัยสุโขทัยธรรมาธิราช จังหวัดนนทบุรี ภายหลังจบการศึกษาได้เข้าทำงานกับ บริษัท มิตซูบิชิ มอเตอร์ส (ประเทศไทย) จำกัด หลังจากสำเร็จการศึกษาเป็นระยะเวลา 1 ปี ได้เข้าศึกษาต่อในระดับปริญญาโทในสาขาวิศวกรรมคอมพิวเตอร์ สำนักวิศวกรรมศาสตร์ มหาวิทยาลัยเทคโนโลยีสุรนารี ในปีการศึกษา 2557

ในระหว่างศึกษาได้รับความไว้วางใจให้เป็นผู้ช่วยวิจัย ณ ห้องปฏิบัติการไอยราคลัสเตอร์ และเป็นผู้ช่วยสอนรายวิชา Computer Programming for Geo-technology, Embedded System, Event-driven Programming, Microcontroller, Object-oriented Technology, Software Analysis and Design และ Software Engineering ในระหว่างศึกษาได้ตีพิมพ์เผยแพร่บทความวิชาการโดยสามารถศึกษารายละเอียดได้ตามภาคผนวก ก