การทดลอง Sort ข้อมูลขนาดใหญ่ด้วยไลบรารี STXXL

SORTING EXPERIMENTS OF BIG DATA WITH STXXL LIBRARY

ณัฐกิจ ไพรสุวรรณา

ณัฐนนท์ รุ่งประเสริฐ

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2558

การทดลอง Sort ข้อมูลขนาดใหญ่ด้วยไลบรารี STXXL

SORTING EXPERIMENTS OF BIG DATA WITH STXXL LIBRARY

ณัฐกิจ ไพรสุวรรณา

ณัฐนนท์ รุ่งประเสริฐ

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2558

ปริญญานิพนธ์การศึกษา 2558

ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การทดลอง Sort ข้อมูลขนาดใหญ่ด้วยไลบรารี STXXL

SORTING EXPERIMENT OF BIG DATA WITH STXXL LIBRARY

ผู้จัดทำ

1.นายณัฐกิจ ไพรสุวรรณา รหัสนักศึกษา 55010350

2.นายณัฐนนท์ รุ่งประเสริฐ รหัสนักศึกษา 55010362

_____อาจารย์ที่ปรึกษา

(ผศ.ดร.สุรินทร์ กิตติธรกุล)

การทดลอง Sort ข้อมูลขนาดใหญ่ด้วยไลบรารี STXXL

นายณัฐกิจ ไพรสุวรรณา 55010350
นายณัฐนนท์ รุ่งประเสริฐ 55010362
ผศ.ดร.สุรินทร์ กิตติธรกุล อาจารย์ที่ปรึกษา
ปีการศึกษา 2558

บทคัดย่อ

การทำงานกับข้อมูลขนาดใหญ่มาก (Big Data) จำเป็นต้องใช้พื้นที่หน่วยความจำมากกว่าหน่วยความจำหลักในคอมพิวเตอร์ทั่วไป การใช้หน่วยความจำรองเช่น SSD เป็นหนึ่งในวิธีการรับมือกับปัญหานั้น ซึ่งการนำหน่วยความจำภายนอกมาใช้ร่วมกับการประมวลผลนั้นจำเป็นต้องมีอัลกอริทึมที่เหมาะสมเพื่อให้ค่าใช้จ่ายด้าน I/O มีผลน้อยที่สุด

โครงงานนี้เลือกใช้ STXXL ซึ่งเป็นไลบรารีมาตรฐานของภาษา C++ ให้ทำงานกับหน่วยความจำภายนอก โดย STXXL มีอัลกอริทึมที่สามารถประมวลผลข้อมูลขนาดใหญ่ที่มากกว่าขนาดของหน่วยความจำหลักได้ดี

ดังนั้นโครงงานนี้จึงนำเสนอการทดลองการจัดเรียงข้อมูลขนาดใหญ่ด้วยไลบรารี STXXL โดยมีการวัดประสิทธิภาพด้านต่างๆของฟังก์ชันการจัดเรียงข้อมูลด้วยเครื่องมือ iostat และ perf

การทดลองแสดงให้เห็นว่าอัลกอริทึมการจัดเรียงข้อมูลของไลบรารี STXXL ได้รับประโยชน์จากความเร็วที่เพิ่มขึ้นของหน่วยความจำรองมากที่สุด ตามด้วยจำนวนของข้อมูลขนาด 1600 ล้านตัว ใช้เวลามากกว่าข้อมูลจำนวน 800 ล้านตัวประมาณ 2.x เท่า และข้อมูลประเภท Double ที่มีขนาดใหญ่การ Uint32 2 เท่าใช้เวลามากกว่าประมาณ 1.7 เท่า โดยที่ตัวแปรอื่นๆส่งผลเพียงเล็กน้อยเท่านั้น

Sorting Experiments of Big Data with STXXL Library

Mr. Nathakit	Praisuwanna	55010350
Mr. Nattanon	Rungprasert	55010362
Asst.Prof.Dr. Surin	Kittitornkun	Advisor

Academic year2015

ABSTRACT

Dealing with Big Data requires more memory (RAM) than ordinary computer usually has. On the other hand, using secondary storage such as SSD is one of the solutions to this problem. In order to use external memory for Big Data while processing it at high performance, the algorithm is required to reduce I/O cost as much as possible.

This project chooses STXXL which is an implementation of C++ standard template library (STL) for external memory computation providing algorithms that can process data too large to fit in main memory.

Therefore, this project provides Sorting Experiment of Big Data with STXXL Library using performance evaluators include iostat and perf.

The STXXL sorting algorithm gains most benefit from secondary storage speed. In addition, sorting 1600M data takes longer time than 800M by 2.x times. The Double data type is twice bigger and 1.7 times longer than Uint32. Other factors have small/negligible effects to the runtime.

กิตติกรรมประกาศ

ปริญญานิพนธ์ฉบับนี้สำเร็จลุล่วงได้ด้วยดีด้วยความช่วยเหลือจากหลายฝ่ายทั้งในทางตรงและ ทางอ้อม โครงงานฉบับนี้จะสำเร็จลงไม่ได้หากปราศจากความช่วยเหลือของบุคคลเหล่านี้

อาจารย์ที่ปรึกษาคือ ผศ.ดร.สุรินทร์ กิตติธรกุล เป็นผู้ที่ให้คำแนะนำ ปรึกษา และให้ความช่วยเหลือตลอดการทำโครงงาน ซึ่งทำให้การทำงานต่าง ๆ เป็นไปได้อย่างราบรื่นและทำให้โครงงานฉบับ นี้สำเร็จไปได้ด้วยดี

อาจารย์และบุคลากรต่าง ๆ ในสาขาวิชาวิศวกรรมคอมพิวเตอร์ที่ได้ให้คำแนะนำและสั่งสอน ความรู้ต่าง ๆ มาตลอด รวมถึงอำนวยความสะดวกด้านสถานที่ในการทำวิจัยและพัฒนาโครงงาน

ในท้ายที่สุดนี้ขอขอบคุณบิดา มารดา และครอบครัวที่ได้เลี้ยงดูและสั่งสอน พร้อมทั้งให้โอกาสใน การศึกษา และให้กำลังใจเสมอมา

ณัฐกิจ ไพรสุวรรณา

ณัฐนนท์ รุ่งประเสริฐ

สารบัญ

หน้า

สารบัญ(ต่อ)

หน้า

สารบัญรูป

รูป หน้า

สารบัญรูป(ต่อ)

รูป หน้า

สารบัญรูป(ต่อ)

รูป หน้า

สารบัญตาราง

ตาราง หน้า

สารบัญตาราง(ต่อ)

ตาราง หน้า

สารบัญตาราง(ต่อ)

ตาราง หน้า

สารบัญตาราง(ต่อ)

ตาราง หน้า

บทที่ 1
บทนำ

1.1 ความสำคัญและที่มาของโครงงาน

ในปัจจุบันปริมาณข้อมูลที่ต้องใช้ในการคำนวณมีมากจนไม่สามารถนำข้อมูลทั้งหมดมาไว้บนหน่วยความจำหลักได้ ทำให้ระหว่างการทำงานตัวระบบปฏิบัติการจะต้องทำการ swap ข้อมูลบางส่วนที่อยู่บนหน่วยความจำหลักมาไว้ในหน่วนความจำรองแทน เนื่องจากหน่วยความจำรองมีความเร็วในการเขียนอ่านช้ากว่าหน่วยความจำหลักมาก และตัวระบบปฏิบัติการไม่สามารถเลือกได้อย่างแม่นยำว่าข้อมูลส่วนไหนที่ควรถูกเขียนลงบนหน่วยความจำสำรอง จึงส่งผลให้การทำงานของโปรแกรมช้าลงอย่างมาก

หนึ่งในแนวทางการแก้ไขคือการใช้หน่วยความจำรองที่มีความเร็วสูงขึ้น เช่น เปลี่ยนจาก HDD เป็น SSD ซึ่งจะทำให้ระบบปฏิบัติการสามารถ swap ข้อมูลระหว่างหน่วยความจำหลักและหน่วยความจำสำรองได้ดีขึ้น แต่เมื่อเทียบกันแล้วความเร็วของ SSD ก็ยังช้ากว่าความเร็วของหน่วยความจำหลักอยู่มาก

อีกหนึ่งแนวทางคือการความคุมการเขียนข้อมูลลงบนหน่วยความด้วยอัลกอริทึมที่กำหนดไว้ล่วงหน้า ซึ่งในโครงงานนี้เลือกใช้โมเดล External memory ที่ถูกพัฒนาด้วยไลบรารี STXXL โดยเราจะวิเคราะห์การทำงานของฟังก์ชันจัดเรียงข้อมูลเป็นหลัก

ฟังก์ชันในการจัดเรียงข้อมูลเป็นฟังก์ชันพื้นฐานที่สำคัญในการทำงานของคอมพิวเตอร์ ซึ่งในฟังก์ชันการจัดเรียงข้อมูลแบบดั้งเดิม เช่น quicksort จะต้องอ่านข้อมูลที่ต้องการจัดเรียงทั้งหมดขึ้นมาไว้บนหน่วยความจำหลัก ซึ่งจะเกิดปัญหาขึ้นเมื่อปริมาณข้อมูลมีขนาดใหญ่กว่าขนาดของหน่วยความจำหลัก แต่ฟังก์ชันการจัดเรียงข้อมูลของ STXXL จะแบ่งการทำงานของการจัดเรียงออกเป็นสองส่วน โดยช่วงแรกจะเป็นการอ่านข้อมูลและจัดเรียงข้อมูลทีละส่วนและในช่วงถัดมาจะเป็นการนำเสนอข้อมูลที่เรียงลำดับแล้วแต่ละส่วนมารวมกันด้วยอัลกอริทึม Multiway-merging ซึ่งทั้งสองช่วงจะมีการควบคุมข้อมูลที่ต้องอ่านหรือเขียนลงบนหน่วยความจำรองตลอดการทำงานเพื่อหลีกเลี่ยงการ swap ของระบบปฏิบัติการ

ในการทดลองเราใช้โปรแกรม iostat ในการวัดประสิทธิภาพของหน่วยความจำรองและ CPU โดยใช้โปรแกรม perf ในการวัดประสิทธิภาพในส่วนที่ iostat ไม่สามารถระบุได้ เช่น page-faults และ context-switches

1.2 วัตถุประสงค์ของโครงงาน

1) เพื่อศึกษาหลักการทำงานและรูปแบบการทำงานของไลบรารี STXXL
2) เพื่อศึกษารูปแบบของ External memory model
3) เพื่อศึกษาอัลกอริทึมจัดเรียงข้อมูลที่ใช้บน External memory
4) เพื่อวัดประสิทธิภาพของอัลกอริทึมจัดเรียงข้อมูลที่ใช้บน External memory

1.3 ขอบเขตของโครงงาน

1) วัดประสิทธิภาพด้านต่างๆของฟังก์ชันจัดเรียงข้อมูลของไลบรารี STXXL
2) ใช้ iostat และ perf ในการวัดประสิทธิภาพในการทำงาน
3) วิเคราะห์ข้อมูลที่ได้จาก iostat และ perf
4) การทดลองทั้งหมดทำงานบน single disk

1.4 วิธีการดำเนินการ

1) ศึกษาการทำงานของฟังก์ชันการจัดเรียงข้อมูลของไลบรารี STXXL
2) ศึกษาวิธีการวัดประสิทธิภาพด้วย iostat และ perf
3) ออกแบบการทดลองรวมถึงข้อมูลประเภทต่างๆที่ใช้ในการจัดเรียง
4) ทำการทดลองและบันทึกผลลัพธ์ที่ได้จาก iostat และ perf
5) วิเคราะห์และสรุปผลการทดลอง
6) จัดทำรูปเล่มรายงาน

1.5 ประโยชน์ที่คาดว่าจะได้รับ

1) สามารถใช้งานไลบรารี STXXL ได้ดี
2) เข้าใจการทำงานของฟังก์ชันการจัดเรียงข้อมูลของไลบรารี STXXL
3) สามารถใช้ iostat และ perf เพื่อวัดประสิทธิผลของการจัดเรียงข้อมูลโดยใช้ไลบรารี STXXL ได้

1.6 ส่วนประกอบปริญญานิพนธ์

ปริญญานิพนธ์ฉบับนี้ได้แบ่งเนื้อหาออกเป็น 5 บทด้วยกัน คือ

บทที่ 1 บทนำ กล่าวถึงความสำคัญและที่มาโครงงาน วัตถุประสงค์ของโครงงาน ขอบเขตของโครงงาน วิธีดำเนินการ ประโยชน์ที่คาดว่าจะได้รับ และส่วนประกอบของปริญญานิพนธ์

บทที่ 2 ทฤษฎีที่เกี่ยวข้อง ในบทนี้จะกล่าวถึงทฤษฎีพื้นฐานของ STXXL เป็นทฤษฎีที่เป็นหัวใจหลักของปริญญานิพนธ์ฉบับนี้

บทที่ 3 การออกแบบการทดลอง กล่าวถึงการออกแบบการทดลองรวมถึงตัวแปรต่างๆที่ใช้ประกอบการทดลอง

บทที่ 4 การทดลองและผลการทดลองกล่าวถึงพารามิเตอร์ต่างๆที่ใช้ในการทดลองอัลกอริทึม การวัดประสิทธิภาพของอัลกอริทึมและผลทดลองของอัลกอริทึม

บทที่ 5 บทสรุป เป็นการสรุปผลการทดลองของปริญญานิพนธ์ จะกล่าวถึงการสรุปผล และประโยชน์ที่ได้รับ

1.7 ดัชนีตัวแปร

N = ข้อมูลจำนวน N ในหน่วยตัว

B = ขนาดของ Blockในหน่วย Byte

M = ขนาดของ CPU cacheในหน่วย Byte

D = จำนวนหน่วยความจำรองในหน่วยลูก

K = จำนวนบัฟเฟอร์สำหรับการผสาน(Merge)ในหน่วยตัว

L = เวลาในการทำ I/O แต่ละครั้งในหน่วยวินาที

σ = ลำดับของบล็อคที่จะอ่าน

บทที่ 2
ทฤษฎีที่เกี่ยวข้อง

2.1 STXXL

STXXL (Standard Template Library for Extra Large Data Sets) เป็นไลบรารีที่พัฒนาต่อยอดจาก standard template library ในภาษา C++ เพื่อทำงานกับหน่วยความจำภายนอก ไลบรารีมี object container และ algorithm ที่สามารถประมวลผลข้อมูลขนาดใหญ่ที่ต้องใช้พื้นที่หน่วยความจำมากกว่าหน่วยความจำหลักได้ นอกจากนั้น STXXL ยังมีความเข้ากันได้ (compatibility) กับSTL ซึ่งทำให้โปรแกรมประยุกต์ที่ใช้ STL อยู่แล้วสามารถเปลี่ยนมาใช้ STXXL ได้โดยง่าย วัตถุประสงค์อีกด้านหนึ่งของ STXXL คือด้านประสิทธิภาพซึ่งมีดังนี้

- สนับสนุนการทำงานของหน่วยความจำรองหลายลูก
- กำหนดขนาดของบล็อคข้อมูลได้ตามที่ต้องการ
- การทับซ้อนกันของ I/O กับการคำนวณ
- การป้องกัน File buffering overhead ของระบบปฏิบัติการ
- การทำงานแบบไปป์ไลน์
- สามารถใช้จำนวนหลายคอร์ในการคำนวณ

2.1.1 External memory model

โมเดลนี้ถูกนำเสนอโดย Aggarwal และ Vitter ในปี1998 ในตอนนั้นใช้ชื่อว่า "I/O Model" หรือ "Disk Access Model" (DAM) ในการนำเสนอนั้นแบ่งลำดับชั้นของ memory เป็น 2 ส่วนคือ CPU ติดต่อกับ Cache ที่มีขนาด M และ Cache เชื่อมต่อกับหน่วยความจำรอง ที่มีขนาดไม่จำกัด โดยทั้งแคชและหน่วยความจำรองจะถูกแบ่งออกเป็นบล็อคที่มีขนาด B หน่วย ดังนั้นจะมีบล็อคในแคช M/B บล็อค การย้ายบล็อคที่อยู่ในแคชลงไปในหน่วยความจำรองจะใช้เวลา 1 หน่วย แต่การกระทำต่างๆที่อยู่บนแคชจะใช้เวลาน้อยมาก ดังนั้น เป้าหมายหลักคือการทำให้จำนวนครั้งของการส่งข้อมูลระหว่างหน่วยความจำรองและแคชมีจำนวนน้อยที่สุด

2.1.1.1 Scanning

การค้นหาข้อมูล N ตัว จะใช้เวลาส่งข้อมูลจากหน่วยความจำ $O(\lceil \frac{N}{B} \rceil)$ ที่ต้องมีขอบเขตบน เพราะถ้า N = O(B) เราจะได้ข้อมูลเกินกว่าที่ต้องการ

2.1.1.2 Searching

การค้นหาที่รวดเร็วจะใช้ B-Tree มี branching factor เป็น $\Theta(B)$ ในทางปฏิบัติ เราต้องการให้มีขนาด B + 1 จริง ๆ ดังนั้นจะมี 1โนดพอดีใน1บล็อคของหน่วยความจำ และมี

branching factor ≥ 2 สำหรับ การแทรก ลบ และ predecessor/successor search จะใช้เวลาส่งข้อมูลจากหน่วยความจำเป็น $O(\log_{B+1} N)$ ซึ่งต้องการเวลาเปรียบเทียบโมเดลเป็น $O(\log N)$

ขอบเขตของ $O(\log_{B+1} N)$ ตามความเป็นจริงแล้วดีที่สุด เราจะเห็นได้จากข้อมูลโต้แย้งทางทฤษฎี

สมมุติว่าเราต้องการจัดการกับข้อมูลที่เหมาะสมกับข้อมูล N ตัว มี N+1 ตำแหน่งที่เราสามารถให้ได้พอดีและเราต้องการ $\theta(\log N + 1)$ บิตเพื่อระบุตำแหน่งของข้อมูลเหล่านั้น ในการอ่านจากแคชแต่ละครั้ง(หนึ่งบล็อค) สามารถบอกได้ว่า มีข้อมูลที่ตรง B ตัว ทำให้ได้ค่าเป็น $O(log(B+1))$ บิตของข้อมูล ดังนั้นเราต้องใช้เวลาอย่างน้อย $\theta(\frac{\log N}{\log(B+1)})$ หรือ $\Omega(\log_{B+1} N)$ ในการส่งข้อมูลเพื่อดูทุกๆ $\theta(\log N + 1)$ บิตสำหรับการลบ หรือแทรกข้อมูล วิธีการนี้ไม่ใช่วิธีการที่ดีที่สุด

2.1.1.3 Sorting

ในแรมนั้นการใช้โครงสร้างข้อมูล B-tree ถือเป็นวิธีที่ดีที่สุด โดยต้องใส่ทุก element จากนั้นทำการแวะผ่านแบบมีลำดับ (in-order traverse) จากเทคนิคนี้ทำให้ได้ค่าเป็น $O(\log_{B+1} N)$ในการส่งข้อมูลใน external memory model ซึ่งไม่ใช่วิธีการที่ดีที่สุด

วิธีการดีที่สุด คือ $\frac{M}{B}$ – way ในรูปแบบของmerge sort เพราะเป็นการแก้ปัญหาโดยแบ่งขนาดให้พอดีกับแคช ทำให้ใช้เวลาในการส่งข้อมูลเป็น $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ ซึ่งเป็นเวลาที่ดีที่สุดในการเปรียบเทียบโมเดล

2.1.2 Object ชนิด sorter

มี external memory container เรียกว่า sorter โดยใช้ stream package object ในการเก็บข้อมูลที่เรียงลำดับเรียบร้อยแล้ว

Sorter object เป็นการรวมการทำงานของฟังก์ชัน runs_creator และruns_merger จาก stream package ให้กลายเป็น two-phase container

ในขั้นตอน(phase)แรกจะเป็นการใส่ข้อมูลไม่เรียงลำดับ (unordered)ลงใน container ซึ่งระหว่างนี้ข้อมูลจะถูกเรียงลำดับภายในชุดข้อมูลตามขนาดที่กำหนด เมื่อหน่วยความจำภายในเกิด overflow ชุดข้อมูลเหล่านั้นก็จะถูกเขียนลงหน่วยความจำภายนอกเป็นบล็อคตามที่กำหนดไว้

ขั้นตอนที่สองจะเริ่มหลังจากข้อมูลทั้งหมดถูกใส่ลงใน container เรียบร้อยแล้ว โดยในขั้นตอนนี้ sorter จะทำการนำชุดข้อมูลที่อยู่ในหน่วยความจำภายนอกมาทำ external sorting

เมื่อ External sorting เสร็จแล้วเราสามารถอ่านข้อมูลที่เรียงลำดับแล้วโดยการใช้ operator*() ในการอ่านข้อมูลตัวแรกสุด และอ่านข้อมูลตัวต่อไปด้วย operator++() เช่นเดียวกับ stream interface

2.1.3 External sorting

External sorting เป็นอัลกอริทึมสำหรับการเรียงลำดับข้อมูลขนาดใหญ่ที่อยู่บนหน่วยความจำภายนอก ซึ่งเหมาะกับการเรียงลำดับข้อมูลที่มีขนาดใหญ่กว่าหน่วยความจำหลัก โดยทั่วไปแล้วการทำ External sorting จะใช้การทำงานร่วมกันระหว่างการเรียงลำดับ (sort) และการผสาน (merge)

การทำงานของ External sorting แบ่งเป็น 2 ขั้นตอน ในขั้นตอนการเรียงลำดับ(sorting phase)จะมีการอ่านชุดข้อมูลขนาดเล็กที่สามารถทำงานบนหน่วยความจำหลักได้ แล้วนำข้อมูลนั้นมาเรียงลำดับหลังจากนั้นจึงเขียนลงบนไฟล์ชั่วคราว(temporary file) ในขั้นตอนการผสาน(merge phase)จะเป็นการนำไฟล์ชั่วคราวเหล่านั้นมารวมกันเป็นไฟล์เดียว

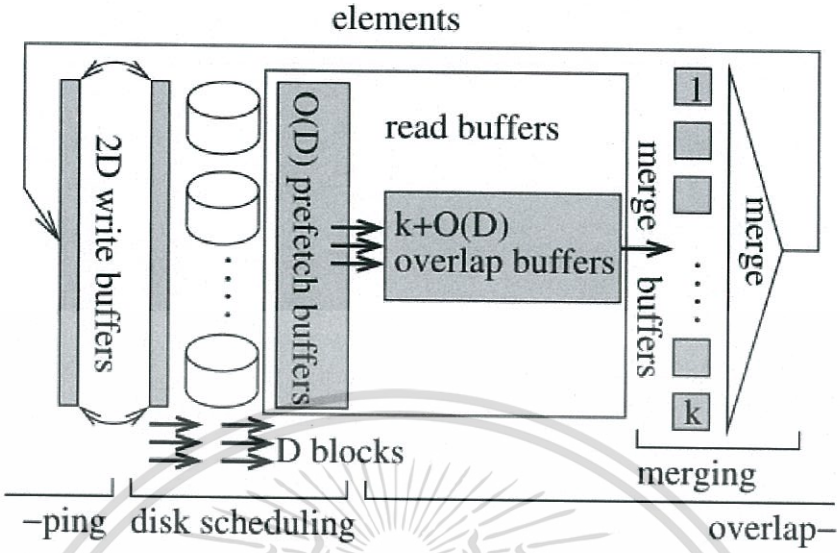
ตัวอย่างการทำงานของ External merge sort ในการเรียงข้อมูลขนาด 900 MB โดยใช้หน่วยความจำหลักขนาด 100MB

1) อ่านข้อมูลขนาด 100 MB ลงบนหน่วยความจำและจัดเรียงข้อมูลด้วยวิธีการเรียงข้อมูลทั่วไปเช่น quicksort
2) เขียนข้อมูลที่เรียงลำดับแล้วลงบนหน่วยความจำภายในทำขั้นตอนที่ 1 และ 2 ซ้ำจนได้ข้อมูลที่เรียงลำดับแล้วจำนวน 9 ชุด แต่ละชุดขนาด 100 MB
3) ทำการรวมข้อมูลที่เรียงลำดับแล้วหลายๆชุดให้เป็นชุดเดียวโดยการ อ่านข้อมูลจำนวน 10 MB แรกของแต่ละชุด (รวม 90MB)และจองพื้นที่สำหรับทำหน้าที่เป็น output buffer จำนวน 10 MB (ในทางปฏิบัติอาจต้องลดขนาด input buffer และเพิ่มขนาด output buffer เพื่อให้ประสิทธิภาพดีขึ้น)
4) ทำการผสานข้อมูล 9 ชุด(9-way merge)และเก็บผลลัพธ์ลงใน output buffer เมื่อใดก็ตามที่ output buffer เต็มให้ทำการเขียนลงใน output file แล้วลบข้อมูลใน output buffer และเมื่อใดก็ตามที่ข้อมูลใน input buffer ใดจากทั้ง 9 input buffer หมดให้ทำการอ่านข้อมูลขนาด 10 MB ถัดไปเข้ามาแทนที่จนกระทั่งไม่มีชุดข้อมูลเหลืออีก

2.2 Asynchronous disk sorting

การจัดเรียงข้อมูลใน STXXL มีลักษณะการทำงานแบบ Asynchronous ซึ่งประกอบด้วย การควบคุมกระแสข้อมูล (Data flow), การ Overlap ระหว่าง I/O และการคำนวณ และการทำ Multiway merging

2.2.1 Data flow



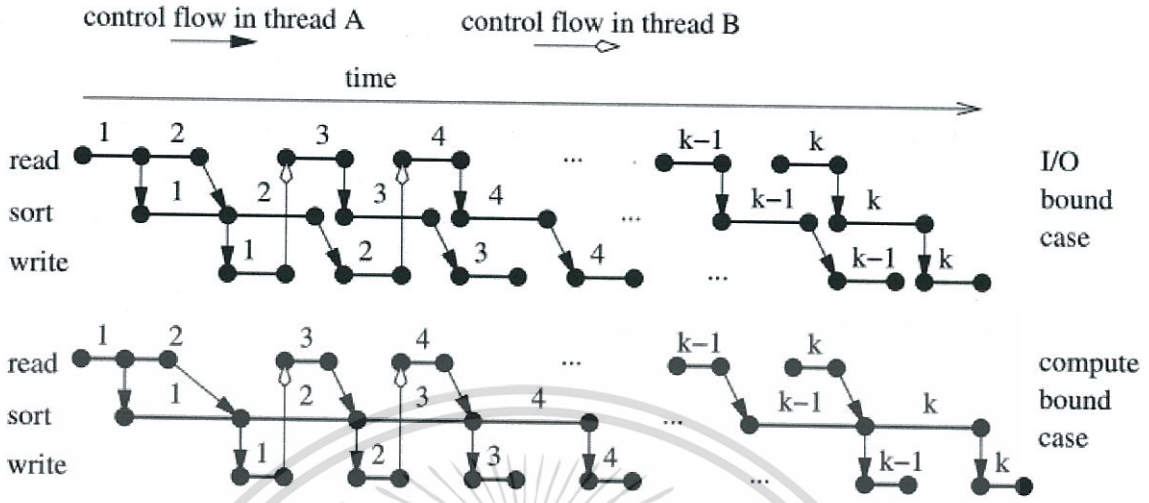
รูป 2.1 กระแสข้อมูล (Data Flow)

เพื่อที่จะทำให้เกิดการ overlap ระหว่าง I/O กับการคำนวณให้มากที่สุด จำเป็นจะต้องมีการควบคุมกระแสข้อมูลอย่างเหมาะสมและมี Buffer จำนวนหนึ่งในการทำ Multi-way merging

การจัดเรียงข้อมูลด้วยหน่วยความจำภายนอกของไลบรารี STXXL มีการไหลของกระแสข้อมูลในขั้นตอนการผสานดังนี้

1) อ่านข้อมูลที่ต้องการจัดเรียงจากหน่วยความจำรองลงใน Prefetch buffersซึ่งสามารถอ่านข้อมูลจากหน่วยความจำรองพร้อมกันหลายลูกได้แบบขนาน
2) เนื่องจากการอ่านข้อมูลแบบขนานเป็นไปอย่างไม่คงที่ทำให้ไม่สามารถการันตีการเกิด overlap ได้ จึงต้องส่งต่อข้อมูลลงใน Overlap buffers ที่มีลักษณะการทำงานแบบ FIFO
3) Merge buffers รับข้อมูลแต่ละบล็อคจาก Overlap buffers แล้วทำการ Merge ข้อมูลด้วยอัลกอริทึม Multi-way merging
4) Write buffers เขียนข้อมูลที่เรียงลำดับแล้วจาก Merger ลงหน่วยความจำรองโดยใน STXXL จะใช้ 2Dwrite buffers เพื่อให้การเขียนข้อมูลเป็นไปอย่างต่อเนื่อง

2.2.2 การ Overlap กันระหว่าง I/O และการคำนวณ



รูป 2.2 การ Overlap ระหว่าง I/O และการคำนวณ

ถึงแม้ว่าเราจะสามารถทำนายลำดับของบล็อคที่ต้องอ่านได้ แต่ไม่สามารถทำนายปริมาณงานที่ต้องทำในระหว่างการอ่านแต่ละครั้งได้ ตัวอย่างเช่นชุดข้อมูลที่เรียงลำดับแล้ว [1,2][3,4][5,6].... เมื่อทำการ initializing merge buffers แล้ว merge thread จะใช้ค่า '1' ก่อนที่จะอ่านข้อมูลลำดับถัดไปในบล็อค '2' ซึ่งก่อนจะได้ข้อมูล '2' นั้นจะเกิดการชะงักขึ้น

เพื่อที่จะแสดงการ overlap ระหว่าง I/O และการคำนวณ เราให้ I/O แต่ละครั้งใช้เวลา L และ I/O สามารถเกิดขึ้นได้พร้อมกันกับการคำนวณ เราให้ Overlap buffer เก็บบล็อคได้จำนวน k+3D บล็อคและมีการทำงานแบบ FIFO เมื่อ Overlap buffer มีข้อมูลอยู่เราสามารถอ่านข้อมูลได้โดยไม่ต้องบล็อค (Asynchronous I/0) ในส่วนของการเขียนนั้น implement ด้วย write buffer FIFO ที่มีขนาด 2 DB

I/O thread จะทำ input หรือ output ด้วยวิธีดังนี้

- เมื่อใดก็ตามที่ไม่มีการทำงานของI/O และมี element อยู่ใน write buffer อย่างน้อย DB ตัว ให้เริ่มขั้นตอน Output
- เมื่อใดก็ตามที่ไม่มีการทำงานของI/O,จำนวน output บล็อคเหลือน้อยกว่า D และมี overlap buffers ว่างอย่างน้อย D ให้ fetch ข้อมูลจาก σ มาไว้ใน overlap buffer จำนวน D บล็อค

ซึ่งวิธีการดังกล่าวสามารถทำให้เกิด overlap ได้เกือบสมบูรณ์

2.2.3 Multi-way merging

ในการ merge ชุดข้อมูลที่เรียงลำดับแล้วนั้น merging thread จะเลือก element ที่มีขนาดเล็กที่สุดจาก k แล้วส่งให้ I/O thread โดยการเลือกอ่านข้อมูลนั้นอาศัยหลักการที่ว่า merging thread ไม่จำเป็นต้องเข้าถึงข้อมูลภายในบล็อคนั้นจนกระทั่ง element ที่เล็กที่สุดของบล็อคนั้นกลายเป็นข้อมูลที่ไม่เคยถูกอ่านมาก่อน ดังนั้นเราจึงต้องมีการบันทึก element ที่เล็กที่สุดของแต่ละบล็อคในขณะที่กำลังอยู่ในขั้นตอนการเรียงลำดับการ merge ลำดับ k โดยเริ่มจาก element ที่มีขนาดเล็กที่สุด แสดงว่าเราสามารถสร้างลำดับของบล็อค(σ)ที่จะถูกอ่านโดย merging thread ได้เช่นกัน

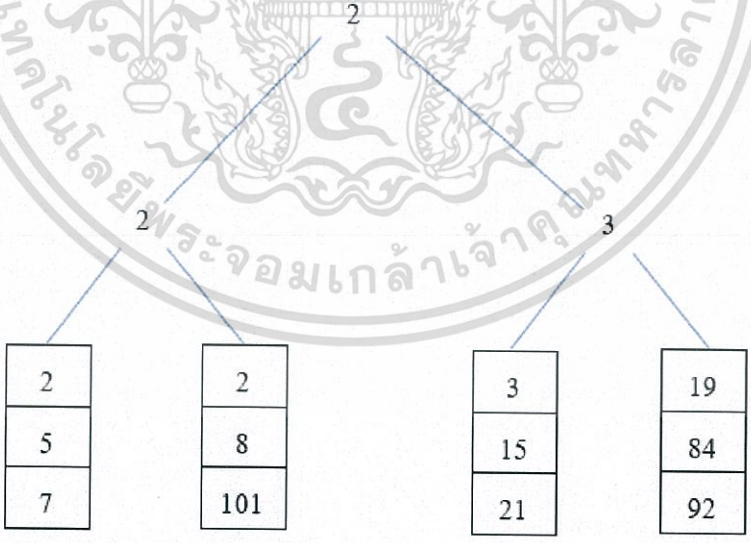
การใช้ลำดับ σ สามารถทำได้ดังนี้

1) Merging thread อ่านบล็อคจำนวน k โดยลำดับบล็อคที่อ่านเลือกจาก σ เก็บไว้ใน merge buffers

2) เมื่อ element สุดท้ายใน merge buffer ใดถูกเลือก ให้อ่านบล็อคถัดไปโดยอิงจาก σ

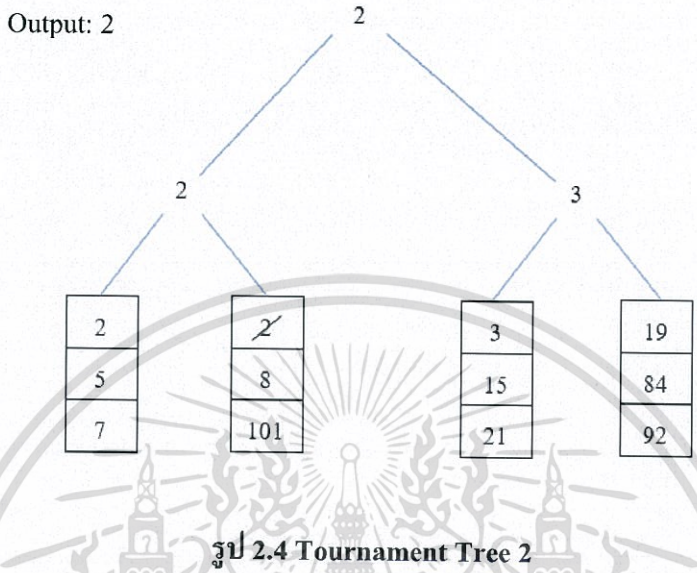
Tournament tree คือ Key ของ element ที่เล็กที่สุดของแต่ละบล็อคจะถูกเก็บไว้ในโครงสร้างข้อมูล (Data structure) เรียกว่า Tournament tree ซึ่งสามารถหา element ที่มีขนาดเล็กที่สุดภายในเวลา O(log k) โดยการทำงานของ Tournament tree ในการหาลำดับ σ มีขั้นตอนดังนี้ สมมุติข้อมูลมีทั้งหมด 3 element ต่อ หนึ่งบล็อคและมีบล็อคทั้งหมด 4 บล็อค

1) Tournament tree จะเก็บเฉพาะค่าที่เล็กที่สุดของแต่ละบล็อคแล้วหา element ที่เล็กที่สุดของทุกบล็อค



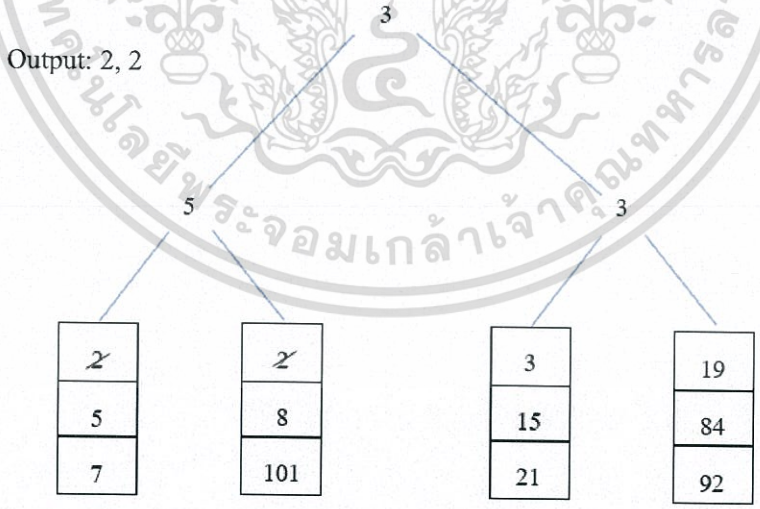
รูป 2.3 Tournament Tree 1

2) เมื่อได้ค่าที่เล็กที่สุดแล้วเขียนลง output file แล้วนำค่านั้นออกจากบล็อคหลังจากนั้นอ่านข้อมูลที่อยู่ถัดไปภายในบล็อคนั้น (8)



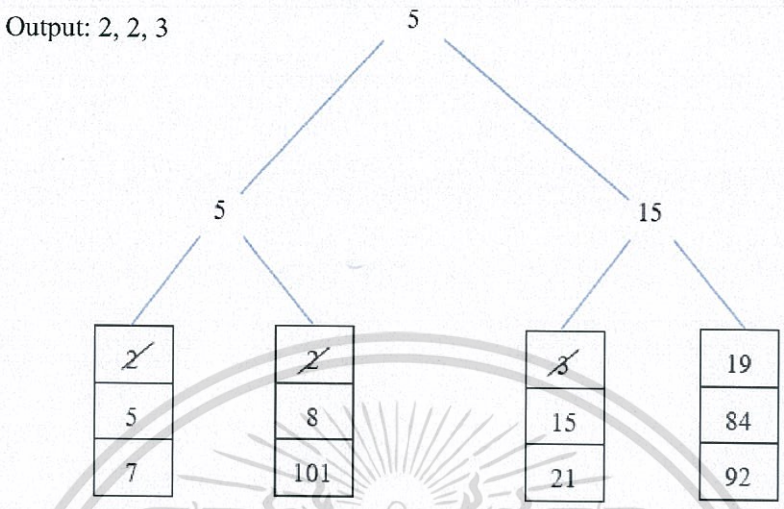
รูป 2.4 Tournament Tree 2

3) ทำการเปรียบเทียบซ้ำเช่นเดียวกับขั้นตอนที่ 1-2



รูป 2.5 Tournament Tree 3

4) การเปรียบเทียบจะจบลงเมื่อบล็อคที่ยังมีข้อมูลเหลืออยู่เพียงบล็อคเดียว



รูป 2.6 Tournament Tree 4

จากอัลกอริทึม Tournament tree ทำให้เราไม่จำเป็นต้องอ่านข้อมูลทั้งบล็อคในการหา σ และสามารถอ่านข้อมูลเฉพาะบล็อคที่มีการเปลี่ยน element ที่มีค่าต่ำที่สุดเท่านั้นได้ ส่งผลให้ overhead ที่เกิดจากการสร้างและเก็บข้อมูลลงในโครงสร้างข้อมูลจะมีขนาดเล็กกว่า input อย่างน้อย B ส่วนเสมอเมื่อ B คือขนาดของบล็อก

2.3 Performance Evaluation

เครื่องมือที่ใช้ในการวัดประสิทธิภาพในการทดลองนี้ประกอบด้วย iostat และ perf โดย iostat จะเน้นในด้านการวัดผลการทำงานของ CPU ที่สัมพันธ์กับ I/O ส่วน perf จะใช้สำหรับการวัดประสิทธิภาพเชิงลึกของ CPU เช่น จำนวน instructions

2.3.1 iostat

iostat เป็นโปรแกรมสำหรับรายงานสถิติการทำงานของ CPU และสถิติของ I/O ของอุปกรณ์และพาร์ติชันใช้สำหรับดูการทำงานของอุปกรณ์ I/O โดยการเก็บข้อมูลจากความสัมพันธ์ระหว่างเวลาที่อุปกรณ์ทำงานและอัตราการโอนย้ายข้อมูล ผลลัพธ์ที่ได้จาก iostat สามารถนำไปใช้ในการปรับปรุงการตั้งค่าเพื่อให้ได้ input/output load ที่เหมาะสมของหน่วยความจำรองแต่ละตัวได้

คำสั่งของ iostat

iostat [-c] [-d] [-N] [-n] [-h] [-k | -m] [-t] [-V] [-x] [-z] [device [...] | ALL] [-p [device [,...] | ALL]] [interval [count]]

ผลลัพธ์ที่ได้จากการรัน iostat ครั้งแรกสุดเป็นสถิติที่นับตั้งแต่ระบบบูทเสร็จ และในการรันครั้งต่อมาจะเป็นผลที่นับจากการรันครั้งล่าสุดผลลัพธ์ที่ได้ประกอบด้วย สถิติของ CPU ซึ่งในกรณีที่เครื่องที่รันเป็น multiprocessor สถิติที่ได้จะเกิดจากค่าเฉลี่ยของหน่วยประมวลผลทั้งหมดและสถิติของแต่ละอุปกรณ์

พารามิเตอร์ interval เป็นตัวระบุระยะเวลาระหว่างแต่ละ report ในหน่วยเป็นวินาทีโดย report แรกจะเป็นสถิตินับตั้งแต่ระบบบูทเสร็จ ส่วนในการรันครั้งถัดไปจะเป็นสถิติที่เริ่มนับจากการรันครั้งล่าสุด พารามิเตอร์ count เป็นพารามิเตอร์ที่ใช้ร่วมกับ interval โดยเมื่อระบุ count แล้ว iostat จะรันเป็นจำนวน count ครั้งแต่ละครั้งห่างกัน interval วินาที แต่ถ้าไม่ระบุค่า count ผลลัพธ์ที่ได้จะออกมาอย่างต่อเนื่อง

คำสั่ง iostat จะสร้าง report ออกมา 2 อย่างได้แก่ CPU Utilization report และ Device Utilization report

ส่วนแรก CPU Utilization Report เป็นการแสดงปริมาณงานของ CPU ในกรณีที่เครื่องเป็น multiprocessor ค่าที่ได้จะเกิดจากค่าเฉลี่ยจากทุก processor โดย report จะมีข้อมูลดังนี้

%user: แสดงเปอร์เซ็นต์ของ CPU utilization ที่เกิดขึ้นในการรันระดับ user level (application)

%nice:แสดงเปอร์เซ็นต์ของ CPU utilization ที่เกิดขึ้นในการรันระดับ user level และค่า nice

%system: แสดงเปอร์เซ็นต์ของ CPU utilization ที่เกิดขึ้นในการรันระดับ system level (kernel)

%iowait: แสดงเปอร์เซ็นต์ของเวลาที่ CPU อยู่ในสภาวะ idle และระบบมี I/O เกิดขึ้นแบบมีนัยยะสำคัญ

%steal: แสดงเปอร์เซ็นต์ของเวลาที่ CPU ถูกบังคับให้อยู่ในสภาวะ wait โดย virtual CPU ในระหว่างที่ hypervisor กำลัง service processor อื่น

%idle: แสดงเปอร์เซ็นต์ของเวลาที่ CPU อยู่ในสภาวะ idle และระบบไม่มี I/O เกิดขึ้นแบบมีนัยยะสำคัญ

Report ส่วนที่สองที่ได้จาก iostat คือ Device Utilization Report ซึ่งแสดงสถิติการใช้งานของแต่ละอุปกรณ์หรือพาร์ติชันโดยสามารถเลือกแสดงเฉพาะอุปกรณ์หรือพาร์ติชันที่สนใจได้ แต่ถ้าไม่ระบุไว้ข้อมูลที่แสดงจะมาจากอุปกรณ์และพาร์ติชันทั้งหมดที่ kernel เก็บสถิติไว้ แต่ถ้ามีการระบุ “ALL” จะเป็นการแสดงสถิติของทุกอุปกรณ์ที่มีอยู่ในระบบรวมถึงอุปกรณ์ที่ไม่เคยถูกใช้งานด้วย โดยข้อมูลที่แสดงสามารถมีได้ดังนี้

Device: แสดงชื่อของอุปกรณ์หรือพาร์ติชัน

Tps: แสดงจำนวน transfer per second ของ device โดย logical I/O request หลายๆครั้งอาจถูกรวมเป็น I/O request ครั้งเดียวได้

Blk_read/s: ความเร็วในการอ่านจากอุปกรณ์โดยใน kernel 2.4 เป็นต้นไปขนาดของบล็อคจะเป็น 512 bytes

Blk_wrtn/s: ความเร็วในการเขียนในหน่วยบล็อคต่อวินาที

Blk_read: จำนวนบล็อคที่ถูกอ่านทั้งหมด

Blk_wrtn: จำนวนบล็อคที่ถูกเขียนทั้งหมด

Rrqm/s: จำนวน read request merged ต่อวินาที

Wrqm/s: จำนวน write request merged ต่อวินาที

r/s: จำนวน read request ต่อวินาที

w/r: จำนวน write request ต่อวินาที

rsec/s: จำนวน sector ที่ถูกอ่านต่อวินาที

wsec/s: จำนวน sector ที่ถูกเขียนต่อวินาที

avgrq-sz: ค่าเฉลี่ยของขนาดของ request ในหน่วย sectors

await: เวลาเฉลี่ยก่อนที่ request จะถูก service ในหน่วยมิลลิวินาที

%util: เปอร์เซ็นต์ของ CPU time ระหว่างที่มี I/O request

Options

-c: แสดง CPU utilization report

-d: แสดง Device utilization report

-k: ให้สถิติที่แสดงอยู่ในหน่วย kbps แทน บล็อคต่อวินาที (blocks per second) (สำหรับ kernel 2.4 ขึ้นไปเท่านั้น)

-m: ให้สถิติที่แสดงอยู่ในหน่วย mbps แทน บล็อคต่อวินาที (blocks per second) (สำหรับ kernel 2.4 ขึ้นไปเท่านั้น)

-N: แสดง registered device mapper names สำหรับทุก ๆ device mapper มีประโยชน์สำหรับการดูสถิติ LVM2

-n: แสดง NFS report สำหรับ kernel 2.6.17 ขึ้นไปเท่านั้น

-p [{device [...] | ALL}]: แสดงสถิติของแต่ละอุปกรณ์แล้วพาร์ติชันที่ถูกใช้โดยระบบ ถ้ามีการใส่ชื่ออุปกรณ์ผลลัพธ์จะแสดงเฉพาะของอุปกรณ์นั้น ถ้าใส่ “ALL” จะแสดงทุกอุปกรณ์และทุกพาร์ติชัน (สำหรับ kernel 2.5 ขึ้นไปเท่านั้น)

-t: แสดง timestamp ของแต่ละ report

-V: แสดงหมายเลข version

-x: แสดงข้อมูลเพิ่มเติม (สำหรับ kernel 2.5 ขึ้นไปเท่านั้น)

-z: ไม่แสดงอุปกรณ์ที่ไม่มีกิจกรรมใด ๆในระหว่างที่ iostat ถูกเรียก

2.3.2 Perf

เป็นโปรแกรมรวบรวมข้อมูลสำหรับ ระบบปฏิบัติการ Linux kernel เวอร์ชั่น 2.6 ขึ้นไป มีลักษณะการทำงานแบบcommand line interface การทำงานของ Perf มีพื้นฐานมาจากอินเตอร์เฟซ perf_eventsของ Linux kernel

คำสั่ง

Perf มีคำสั่งต่าง ๆมากมายในการเก็บรวบรวมและวิเคราะห์ประสิทธิภาพรวมถึงติดตามข้อมูลชุดคำสั่งที่ใช้ในการคอมไพล์

-e, -event=: PMU event โดยอาจเลือก symbolic event name (รายชื่อใน perf list all event) หรือข้อมูลดิบ (eventsel + usmask)โดยแสดงในรูปแบบของ rNNN ซึ่ง NNN เป็น event descriptor ในรูปแบบเลขฐานสิบหก

-i, –no-inherit: counter ของ child tasksจะไม่นับต่อจาก parent

-p, --pid = <pid>: สถิติของ events ของ process id ที่ระบุ

-t, --tid =<tid>: สถิติของ events ของthread id ที่ระบุ

-a –all-cpus: รวบรวมข้อมูลจากทุก CPU

-c, –scale: scale/normalize ค่าของ counter

-r, --repeat = <n>: ทำคำสั่งนั้นซ้ำและแสดงค่าเฉลี่ยกับส่วนเบี่ยงเบนมาตรฐาน(สูงสุดที่ 100)

-B –big-num: พิมพ์เลขที่มีขนาดใหญ่มากด้วย พร้อมทั้งแสดงเครื่องหมายคั่นหลักพันตามแต่ละ locale

-C --cpu: นับเฉพาะรายชื่อของ CPU ที่กำหนดไว้ โดยสามารถระบุ CPU หลายตัวได้โดยการใช้จุลภาคโดยไม่ต้องใส่ช่องว่าง เช่น 0,1 ส่วนการกำหนดเป็นช่วงจะใช้เครื่องหมาย – เช่น 0-2 และในการทำงานแบบ per-thread-mode จะไม่สนใจคำสั่งนี้ ถ้าไม่มีการระบุค่าใด ๆ perf จะเก็บข้อมูลจาก cpu ทั้งหมด

-A, --no-aggr:ไม่รวมค่าที่นับได้จากทุก CPU (ใช้ได้ใน system-wide-mode เท่านั้น)

-n, --null: ไม่นับค่าใด ๆ

-v, –verbose: แสดงข้อมูลอย่างละเอียดมากขึ้น (แสดงข้อผิดพลาดของการเปิดตัวนับฯลฯ)

-x SEP, --field-separator SEP: แสดงจำนวนมีผลลัพธ์ในแบบCSVเพื่อทำให้ง่ายต่อการนำเข้าใน spreadsheets โดยแต่ละแถวจะถูกแบ่งโดยตัวอักษรที่ระบุโดยพารามิเตอร์SEP

-G name, --cgroup name: วัดเฉพาะค่าที่อยู่ใน cgroup ที่ระบุโดยพารามิเตอร์ "name" ความสามารถนี้สามารถใช้ได้ใน per-cpu mode เท่านั้นโดยที่ cgroup filesystem จะต้องถูก mount อยู่ เราสามารถระบุ cgroup หลายๆค่าได้ ซึ่งจะทำให้ Perf วัดค่าเฉพาะเหตุการณ์ที่เราต้องการเท่านั้นได้

-o file, --output file: แสดงผลลัพธ์ในไฟล์ที่ระบุ

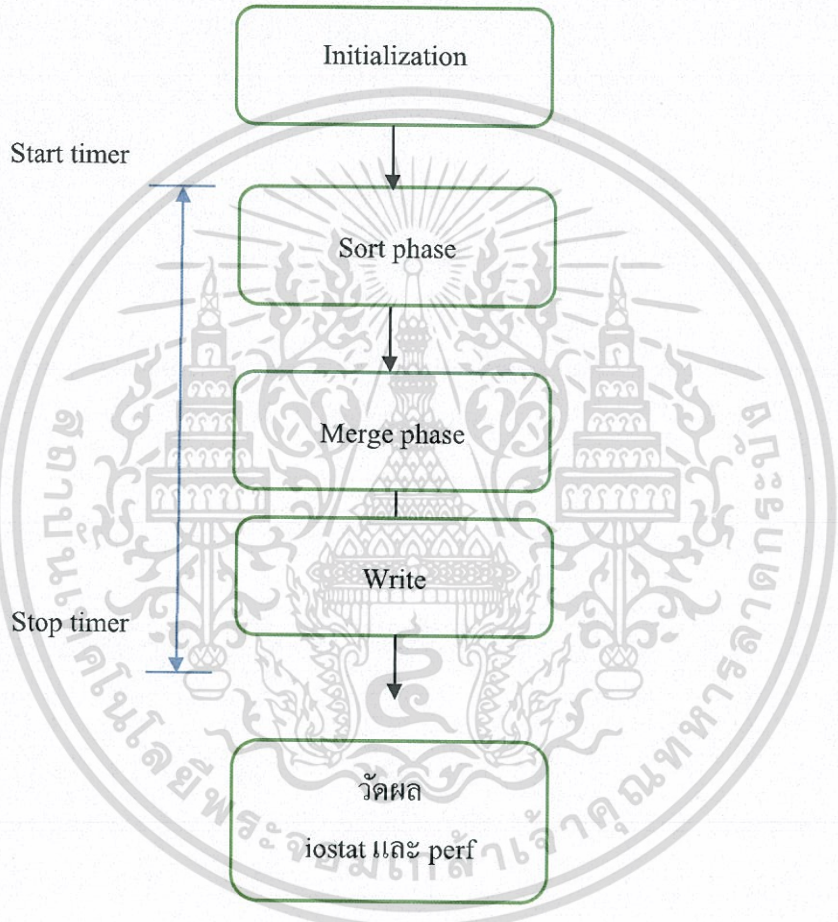
-append: ต่อท้ายผลลัพธ์ของไฟล์ที่ระบุที่ใช้ option–o (ใช้ไม่ได้ถ้า ไม่มี –o)

--log-fd: บันทึก ผลลัพธ์ ในfd แทนที่stderr ไม่สามารถใช้ร่วมกับ –oได้ แต่สามารถใช้ร่วมกับ–append ได้ เช่น 3> result perf stst –log-fd 3 -$cmd 3>>results perf stat –log-fd 3 –append - $cmd

บทที่3
วิธีการทดลอง

3.1 การทำงานของโปรแกรม

ขั้นตอนการทำงานของโปรแกรมสามารถแบ่งได้ดังนี้



รูป 3.1 ขั้นตอนการทำงานของโปรแกรม

3.1.1 Initialization

สร้าง sorter object สำหรับประเภทข้อมูลที่ใช้ทดลองประกอบด้วย uint32 และ double มีการกำหนดขนาดของ บล็อค และ หน่วยความจำหลักที่ใช้ซึ่งมีการใช้งานดังนี้ stxxl: : sorter<unsigned,my_comparator,1* 1024* 1024>unsigned_sorter(my_comparator( ) , 7*1024*1024);

คือการสร้าง sorter สำหรับข้อมูลชนิด unsigned โดยกำหนดขนาดบล็อคเป็น 1*1024*1024 Bytes และให้ขนาดของหน่วยความจำหลักที่จะใช้เป็น 7*1024*1024 โดย CompareType (my_comparator) ต้องเป็นคลาสที่ประกอบด้วยเมธอด operator() , min_value( ) และ max_value()

3.1.2 ขั้นตอนการจัดเรียงข้อมูล (Sort phase)

การทำงานของขั้นตอนการจัดเรียงข้อมูลประกอบด้วยการอ่านไฟล์ข้อมูลที่สร้างไว้ล่วงหน้าส่วนหนึ่งขึ้นมาไว้บนหน่วยความจำหลัก แล้วใช้อัลกอริทึมการจัดเรียงแบบปกติ เช่น quicksort จากนั้นจึงเขียนข้อมูลลงบน external memory แล้วจึงอ่านข้อมูลที่ยังไม่ได้จัดลำดับส่วนต่อไปจนข้อมูลถูกจัดเป็นกลุ่มทั้งหมด

มีการเริ่มจับเวลาตั้งแต่การเริ่มอ่านไฟล์เพื่อนำข้อมูลใส่ใน sorter object และดำเนินงานตามช่วงการเรียงลำดับของอัลกอริทึม external sorting โดยการใส่ข้อมูลลงใน sorter ทำได้โดย unsigned_sorter.push(data)

3.1.3 ขั้นตอนการผสาน (Merge phase)

การทำงานของขั้นตอนการผสานประกอบด้วยการอ่านบล็อคข้อมูลที่เรียงลำดับแล้วจาก external memory มาทำการ merge ด้วยอัลกอริทึม Multiway merging โดยทำนายข้อมูลที่จะต้องอ่านถัดไปของแต่ละบล็อคด้วย Tournament tree

ขั้นตอนการผสานจะเริ่มหลังจากขั้นตอนการจัดเรียงทำงานเสร็จและจะหยุดจับเวลาเมื่อฟังก์ชันการผสานทำงานเสร็จ หลังจากจบขั้นตอนนี้เราจะได้ผลลัพธ์ที่เรียงลำดับเก็บไว้ใน data structure การเปลี่ยนจากช่วงการจัดเรียงข้อมูลไปเป็นช่วงการผสานสามารถทำได้โดย unsigned_sorter.sort() และทำการแสดงเวลาหลังจากที่ฟังก์ชัน sort ทำงานเสร็จแล้ว

โดยการทำงานของฟังก์ชัน sort() จะเป็นการอ่านข้อมูลจาก external memory มา merge เฉพาะเท่าที่พารามิเตอร์ Buffer จะอนุญาตเท่านั้น

3.1.4 การเขียนข้อมูล

การเขียนข้อมูลคือการอ่านข้อมูลที่เรียงลำดับแล้วทั้งหมดจาก external memory แล้วเขียนลงบนหน่วยความจำรอง โดยลักษณะการอ่านจะเหมือนกับ stream interface โดยจะอ่านข้อมูลลำดับถัดไปด้วย operator++()

ระหว่างการอ่านข้อมูลลำดับถัดไปจะมีการนับจำนวนข้อมูลที่คงเหลืออยู่ใน buffer โดยเมื่อข้อมูลใน buffer ถูกอ่านจนใกล้หมดจะมีการอ่านข้อมูลชุดถัดไปจาก external memory เพื่อทำการ merge ครั้งถัดๆไป

3.1.5 การวัดผลโดยใช้ iostat และ perf

เราใช้ iostat ในการวัดประสิทธิภาพการทำงานของหน่วยความจำรองและการทำงานของ CPU ในระหว่างการทำงานของโปรแกรม และใช้ perf สำหรับแสดงรายละเอียดอื่นๆเช่น context-switch เป็นต้น

3.1.5.1 iostat

ตัวอย่างการใช้งาน iostat

mate-terminal -e "./test1 dat/10000000_double.dat" & iostat -c -d 2

คำสั่ง -c แสดงค่า CPU utilization ทุก ๆ 2 วินาที

คำสั่ง -d แสดงค่า Device utilization ทุก ๆ 2 วินาที

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           5.91    0.00    0.50   21.76    0.00   71.82

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda              55.50     14336.00     14080.00      28672      28160
scd0              0.00         0.00         0.00          0          0

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           6.03    0.00    0.63   22.24    0.00   71.11

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda              61.50     15872.00     15616.00      31744      31232
scd0              0.00         0.00         0.00          0          0

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           7.35    0.00    3.17   18.88    0.00   70.60

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda              73.00     19456.00     17424.00      38912      34848
scd0              0.00         0.00         0.00          0          0

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           9.76    0.00   15.85   18.56    0.00   55.83

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda              59.00     30208.00         0.00      60416          0
scd0              0.00         0.00         0.00          0          0
```

รูป 3.2 ผลลัพธ์ของ iostat

จากรูปผลลัพธ์ของ iostat จะแสดงค่าต่างๆดังนี้

%user: แสดงเปอร์เซ็นต์ของ CPU utilization ที่เกิดขึ้นในการรันระดับ user level (application)

%nice:แสดงเปอร์เซ็นต์ของ CPU utilization ที่เกิดขึ้นในการรันระดับ user level และค่า nice

%system: แสดงเปอร์เซ็นต์ของ CPU utilization ที่เกิดขึ้นในการรันระดับ system level(kernel)

%iowait: แสดงเปอร์เซ็นต์ของเวลาที่ CPU อยู่ในสภาวะ idle และระบบมี I/O เกิดขึ้นแบบมีนัยยะสำคัญ

%steal: แสดงเปอร์เซ็นต์ของเวลาที่ CPU ถูกบังคับให้อยู่ในสภาวะ wait โดย virtual CPUในระหว่างที่ hypervisor ก าลัง service processor อื่น

%idle: แสดงเปอร์เซ็นต์ของเวลาที่ CPU อยู่ในสภาวะ idle และระบบไม่มี I/O เกิดขึ้นแบบมีนัยยะสำคัญ

tps: แสดงจำนวน transfer per second ของ device โดย logical I/O request หลายๆ ครั้งอาจถูกรวมเป็น I/O request ครั้งเดียวได้

kB_read/s: แสดงประมาณข้อมูลที่อ่านได้ในหนึ่งวินาทีโดยมีหน่วยเป็นกิโลไบต์

kB_read: แสดงประมาณข้อมูลที่อ่านได้ทั้งหมดนับตั้งแต่รายงานครั้งก่อนโดยมีหน่วยเป็นกิโลไบต์

kB_wrtn/s: แสดงประมาณข้อมูลที่อ่านได้ในหนึ่งวินาทีโดยมีหน่วยเป็นกิโลไบต์

kB_wrtn: แสดงประมาณข้อมูลที่อ่านได้ทั้งหมดนับตั้งแต่รายงานครั้งก่อนโดยมีหน่วยเป็นกิโลไบต์

3.1.5.2 Perf

ตัวอย่างการใช้งาน perf โดยการใช้คำสั่ง

perf stat -r 10 ./test1 dat/10000000_double.dat

คำสั่ง –r คือการให้ทำซ้ำตามจำนวน 10 รอบ

```
[STXXL-MSG] Finished writing file after 2.46621 seconds

 Performance counter stats for './test1 dat/10000000_double.dat' (10 runs):

      14648.042305      task-clock (msec)         #    0.617 CPUs utilized
      ( +-  0.62% )
            1,843       context-switches          #    0.126 K/sec
      ( +-  0.76% )
               73       cpu-migrations            #    0.005 K/sec
      ( +-  6.01% )
            2,779       page-faults               #    0.190 K/sec
      ( +-  2.50% )
   <not supported>      cycles
                0       stalled-cycles-frontend   #    0.00% frontend cycles idl
e
                0       stalled-cycles-backend    #    0.00% backend  cycles idl
e
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

      23.721654386 seconds time elapsed
( +-  2.68% )
```

รูป 3.3 ผลลัพธ์ของ Perf

จากรูปผลลัพธ์ของ iostat จะแสดงค่าต่างๆดังนี้

task-clock: เวลาที่ใช้ในการทำงานของ task นี้มีหน่วยเป็นมิลลิวินาที

context-switches: จำนวน context switch ตลอดการทำงานของโปรแกรม

cpu-migrations: จำนวนครั้งที่งานถูกย้ายการทำงานจากคอร์หนึ่งไปสู้อีกคอร์หนึ่ง

page-faults: จำนวน page fault ตลอดการทำงานของโปรแกรม

stalled-cycles-frontend: จำนวน cycle ที่ stalled ในระหว่าง fetch และ decode phase

stalled-cycles-backend: จำนวน cycle ที่ stalled ระหว่างการ execute instruction

instructions: จำนวน instructions ของโปรแกรม

branches: จำนวนครั้งของการ branch ตลอดการทำงานของโปรแกรม

branches-miss: จำนวนครั้งที่ branch prediction ทำงานพลาด

3.2 รูปแบบการทดลอง

3.2.1 ประเภทข้อมูลที่ใช้ทดลอง

ทุกเครื่องในการทดลองใช้ระบบปฏิบัติการ Ubuntu 14.04 LTS โดยแต่ละเครื่องมี HDD เป็นของตัวเองแต่ใช้ SSD ลูกเดียวกัน

ตาราง 3.1 พารามิเตอร์ที่ใช้ในการทดลอง

พารามิเตอร์	ค่า
CPU	Intel I3-2100, Intel I7-2600, AMD A6-3650, AMD FX-8320
Media	HDD , SSD
RAM	8GB , 16GB
Cache	YES , NO
Type	Uint32 , Double
Distribution	Random , Reversed
Data Size	1600M , 800M
Buffer Size	256MB, 1024MB
Optimization flag	-O3

แคช(Cache) คือขนาดของ Page Cache ลบด้วย Swap Cached ระบบปฏิบัติการจะทำ file i/o ทั้งหมดผ่าน page cache โดยในส่วนของการ write จะทำโดยการทำให้ mark page ที่ต้องการเขียนเป็น dirty หลังจากนั้น flusher threads จะเขียน dirty page กลับไปยัง disk เป็นระยะ ๆ ในส่วนการอ่านข้อมูลทำโดยการอ่านค่าจาก page cache แต่ถ้าข้อมูลนั้น ๆยังไม่ได้อยู่ใน cache (Page cache miss) ข้อมูลจะต้องถูกโหลดลงในแคชก่อน

ขนาดของแคชอาจมีหลายกิกะไบต์ได้แต่เมื่อระบบต้องการหน่วยความจำเพิ่มเพื่อจะนำไปใช้กับงานอื่น ๆ ระบบปฏิบัติการจะลบ page cache หรือย้ายข้อมูลลงไปไว้ใน disk เพื่อเพิ่ม available memory

3.2.2 คุณลักษณะเฉพาะของเครื่องที่ใช้ทดสอบ

ตาราง 3.2 เปรียบเทียบคุณลักษณะเฉพาะของเครื่องที่ใช้ทดสอบ

Core architecture (code name)	Intel i3-2100 (Sandybridge)	Intel i7-2600 (Sandybridge)	AMD A6-3650 (Llano)	AMD FX-8320 (Vishera)
Socket/Core	1/4	1/4	1/4	1/8
HyperThread	Yes	Yes	No	No
Clock(GHz)	3.1 GHz	3.4 GHz	2.6 GHz	3.5 GHz
L1/L2 (per core)	32KB/256KB	32KB/256KB	64KB/1MB	32i+16dKB/1MB
L3	3MB	8MB	-	8MB
RAM_capacity	8GB	16GB	16GB	16GB
RAM_Techno	DDR3-1066/1333	DDR3-1066/1333	DDR3-1866	DDR3-1866
Bandwidth	21 GB/s	21 GB/s	29.9 GB/s	29.9 GB/s
Mem Channels	2	2	2	2
Others	shared Video Smart Cache 3MB	Smart Cache 8MB DMI 5GT/s	PCI express 2.0 16-way L2	HyperTransport technology

บทที่ 4

ผลการทดลอง

4.1 ผลการทดลองที่ใช้เวลาเร็วและช้าที่สุด

จากผลการทดลองสามารถแบ่งออกได้เป็น ผลที่ใช้เวลาเร็วและช้าที่สุด, ผลการเปรียบเทียบ HDD กับ SSD , ผลการเปรียบเทียบแบบมีแคชและไม่มีแคช , ผลการเปรียบเทียบข้อมูลชนิด Uint32 กับ Double , ผลการเปรียบเทียบข้อมูลจำนวน 800 ล้านตัว กับ 1600 ล้านตัว, ผลการเปรียบเทียบการเรียงข้อมูลแบบ Reversed กับ Random, ผลการเปรียบเทียบบัฟเฟอร์ขนาด 256 MB กับ 1024 MB และ ผลการเปรียบเทียบแรม 8 GB กับ 16 GB โดยได้ผลการทดลองเร็วที่สุดและช้าที่สุดดังนี้

ตาราง 4.1 เวลาในการรันที่น้อยที่สุดและมากที่สุด

CPU	Time (sec)			Time(sec)		
i3-2100	Storage: SSD Ram: 16 GB Cache: YES Type: Uint32 Data size: 800M : Reversed Buffer: 1024 MB	Read (Sort)	28.87	Storage: HDD Ram: 8 GB Cache: YES Type: Double Data size: 1600m : Random Buffer: 1024 MB	Read (Sort)	409.11
		Merge	2.98		Merge	9.34
		Write	34.30		Write	400.47
		Total	66.16		Total	819.06
i7-2600		Read (Sort)	25.80	Storage: HDD Ram: 8 GB Cache: YES Type: Double Data size: 1600m : Reversed Buffer: 256 MB	Read (Sort)	597.09
		Merge	3.40		Merge	3.16
		Write	26.80		Write	356.7
		Total	56.01		Total	957.36
A6-3650		Read (Sort)	38.68	Storage: HDD Ram: 16 GB Cache: NO Type: Double Data size: 1600m : Random Buffer: 1024 MB	Read (Sort)	804.08
		Merge	0.82		Merge	18.37
		Write	59.79		Write	634.95
		Total	99.30		Total	1458.67

ตาราง 4.1 เวลาในการรันที่น้อยที่สุดและมากที่สุด (ต่อ)

CPU	Time (sec)			Time(sec)		
FX-8320	Storage: SSD Ram: 16 GB Cache: YES Type: Uint32 Data size: 800M : Reversed Buffer: 1024 MB	Read (Sort)	47.50	Storage: HDD Ram: 8 GB Cache: NO Type: Double Data size: 1600m : Random Buffer: 256 MB	Read (Sort)	885.98
		Merge	5.74		Merge	7.41
		Write	121.28		Write	1625.37
		Total	174.52		Total	2518.98

4.2 เปรียบเทียบ SSD กับ HDD

โดยทั่วไปแล้ว SSD สามารถเข้าถึงข้อมูลเร็วกว่า HDD ทำให้ในขั้นตอนการ Read , ขั้นตอนการ Merge และ ขั้นตอนการ Write เวลาของนั้น SSD ใช้เวลาน้อยกว่า HDD

ตาราง 4.2 เวลาในการรันโดยใช้ข้อมูลชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบ SSD กับ HDD

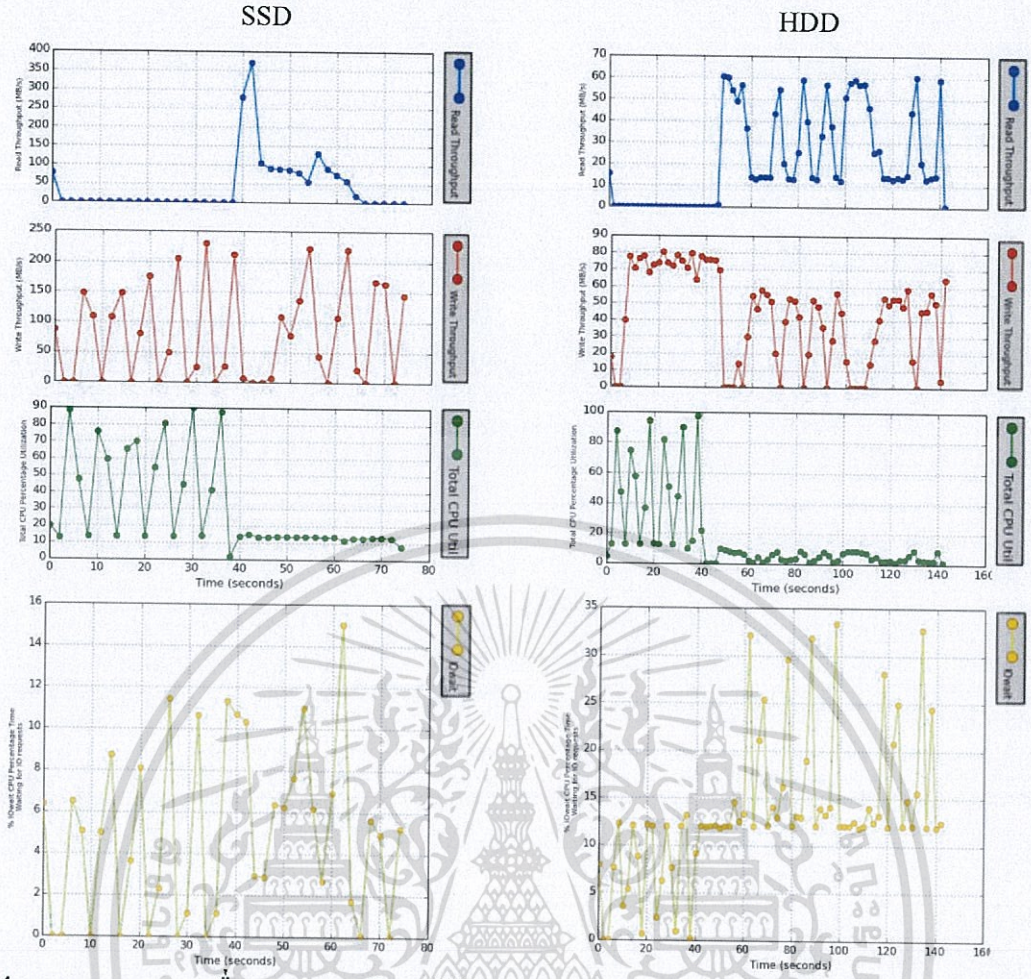
CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	SSD	HDD	SSD	HDD	SSD	HDD	SSD	HDD
Read (Sort)	48.72	46.93	33.33	35.94	51.79	72.47	72.72	73.17
Merge	6.67	8.72	4.85	10.24	1.47	28.51	10.29	18.72
Write	44.44	65.42	36.26	96.51	71.09	185.52	141.16	189.23
Total	99.83	121.06	74.44	142.69	124.35	286.49	224.17	281.12

จากผลการทดลองพบว่าเวลาในขั้นตอน Read นั้น SSD และ HDD ใช้เวลาใกล้เคียงกันเพราะการอ่านข้อมูลเป็นการอ่านจากแรมเนื่องจากเลือกเก็บข้อมูลแบบมีแคชสำหรับเวลาในขั้นตอน Merge และ Write พบว่า SSD สามารถทำได้เร็วกว่า

ตาราง 4.3 อัตราส่วนความเร็วโดยใช้ข้อมูลชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบ SSD กับ HDD

Speedup	Ratio HDD:SSD speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	0.96	1.08	1.40	1.01
Merge	1.31	2.11	19.35	1.82
Write	1.47	2.66	2.61	1.34
Total	1.21	1.92	2.30	1.25

จากอัตราส่วนความเร็วพบว่าขั้นตอนการ Read พบว่า SSD มีเวลาไม่ต่างกับ HDD มากนัก เพราะเป็นการอ่านข้อมูลบนแรมแทน แต่ขั้นตอนการ Merge และการ Write นั้น SSD ใช้เวลาน้อยกว่า HDD



รูป 4.1 iostat ของเครื่อง i7-2600 โดย ใช้ข้อมูลชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบ SSD กับ HDD

จากกราฟเครื่อง i7-2100 พบว่าขั้นตอน read มี CPU utilization ของ SSD สูงถึง 90% และ HDD สูงถึง 94% โดยจะสลับขึ้นลงกับ Write Throughput และ Read Throughput แต่เนื่องจากเป็นแบบมี cache ทำให้การอ่านข้อมูลไปอยู่ที่แรมแทนซึ่งส่งผลให้เกิดการอ่านและเขียนข้อมูลพร้อมกันได้ซึ่ง HDD สามารถเขียนข้อมูลได้อย่างต่อเนื่องแต่เนื่องจาก SSD มีการเขียนข้อมูลที่ไวทำให้ต้องรอข้อมูลจากการอ่านและจัดเรียงจนกว่าจะเต็มบัฟเฟอร์จึงจะเริ่มเขียนข้อมูล

ในขั้นตอนการ Merge CPU utilization ในวินาทีที่ 33 ถึงวินาทีที่ 38 ของ SSD จาก 30% ขึ้นไปถึง 90% และลดลงมาเหลือ 0% ในขั้นตอน Merge ซึ่งตอนที่ CPU utilization เพิ่มขึ้น SSD ได้ทำการ Merge ข้อมูลบนแรมซึ่งตอนนี้จะไม่มีการอ่านเขียนไฟล์จนกว่าจะคำนวณเสร็จและเมื่อคำนวณเสร็จจะนำข้อมูลบนแรมเขียนกลับไปในหน่วยความจำรองทำการให้ Write Throughput สูงขึ้นพร้อมกับ iowait ที่สูงขึ้นระหว่างนี้ CPU utilization จะเหลือ 0% แต่ของ HDD อยู่ที่วินาทีที่ 36 ถึงวินาทีที่ 46 จะพบว่า CPU utilization เพิ่มขึ้นจาก 15% ถึง 100% และลดลงมาเหลือ 0% และอยู่นาน

ถึง 4 วินาทีเนื่องจากยังเขียนข้อมูลที่ได้จากการ merge ไม่เสร็จโดยสังเกตุจาก Write Throughput ที่ 80 MB/s

ขั้นตอน Write ของ SSD พบว่า Read Throughput ขึ้นสูงถึง 370 MB/s ก่อนจะลงมาอยู่ในช่วง 100 MB/s และคงที่ ซึ่งขั้นตอนนี้คือการพยายามอ่านข้อมูลที่เรียงแล้วใส่ในบัฟเฟอร์เมื่อบัฟเฟอร์เต็มก็จะทำการเขียนข้อมูลซึ่ง Write Throughput จะสูงขึ้นและ Read Throughput จะลดลง โดยขั้นตอนนี้จะใช้ CPU คงที่และ iowait จะขึ้นลงตามความสามารถในการอ่านเขียน และเมื่อบัฟเฟอร์หมดจะทำดึงข้อมูลส่วนถัดไปมา merge ใหม่สำหรับ HDD เมื่อเริ่มเขียน iowait จะเพิ่มมาอยู่ที่ 12% และจะเพิ่มสูงขึ้นถึง 34% และ CPU utilization ลดลงซึ่งแสดงให้เห็นถึงการเขียนข้อมูลที่ช้าจน CPU ไม่สามารถทำงานได้และต้องรอการเขียนข้อมูลจนกว่าจะเสร็จ

ตาราง 4.4 เวลาในการรันโดยใช้ข้อมูลชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบ SSD กับ HDD

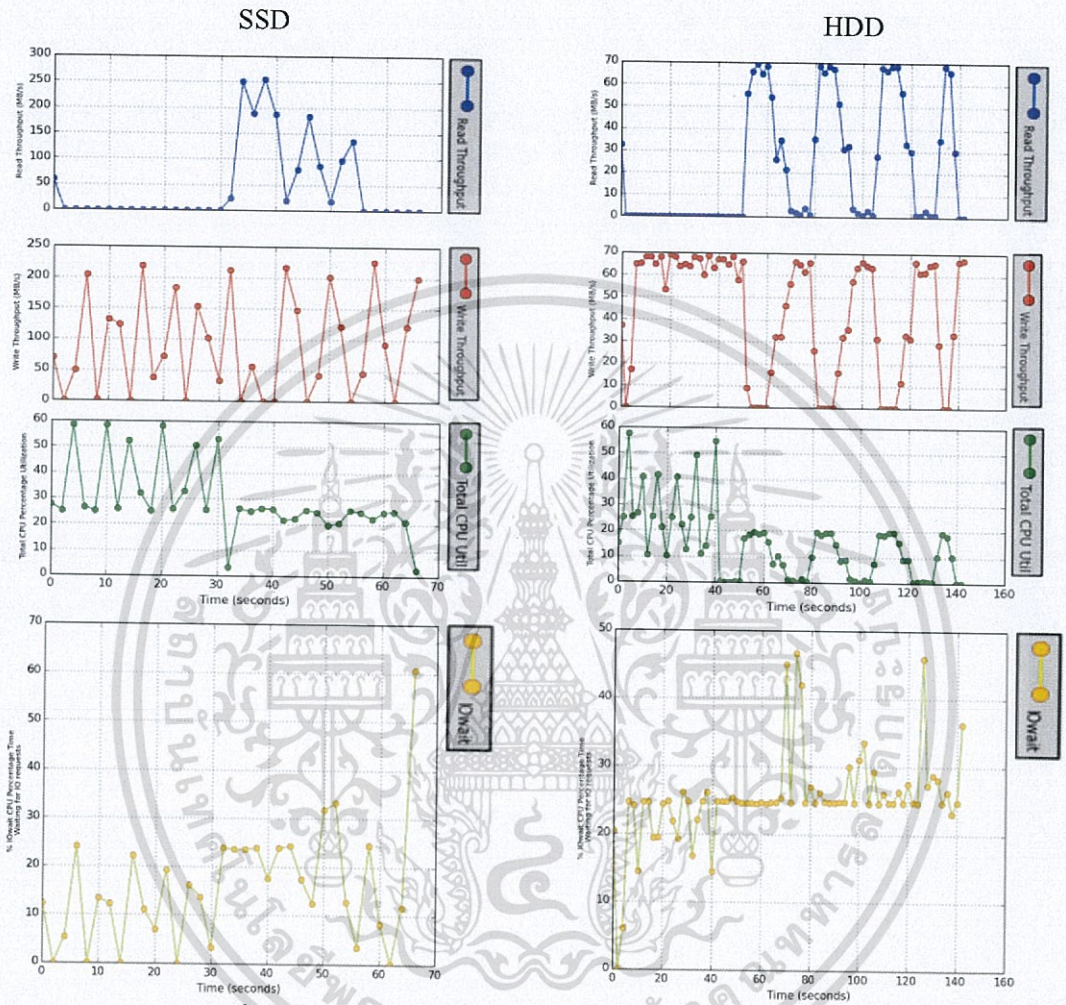
CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	SSD	HDD	SSD	HDD	SSD	HDD	SSD	HDD
Read (Sort)	28.87	39.07	25.80	44.56	38.68	97.36	47.50	72.99
Merge	2.98	11.50	3.40	15.58	0.82	23.72	5.74	25.56
Write	34.30	91.59	26.80	119.45	59.79	173.53	121.27	169.51
Total	66.16	142.16	56.01	179.59	99.30	294.62	174.52	268.05

จากผลการทดลอง พบว่า SSD จะใช้เวลาน้อยกว่า HDD ทั้งในขั้นตอน Read , Merge , Write ซึ่งเราสามารถเทียบโดยสามารถเทียบอัตราส่วนความเร็วได้ดังนี้

ตาราง 4.5 อัตราส่วนความเร็วโดยใช้ข้อมูลชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบ SSD กับ HDD

Speedup	Ratio HDD:SSD speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	1.35	1.72	2.51	1.54
Merge	3.86	4.58	28.81	4.45
Write	2.67	4.46	2.90	1.40
Total	2.15	3.21	2.97	1.54

จากอัตราส่วนความเร็วพบว่า SSD เร็วกว่าในขั้นตอน Merge และการ Write ที่สูงมากโดยมากกว่าในระดับที่เกิน 2.5 เท่าของการทดลองเมื่อเทียบผล SSD กับ HDD ซึ่งเวลาที่ใช้ส่วนใหญ่จะอยู่ในขั้นตอน Read และ Write ดังนั้น SSD ทำความเร็วได้ดีกว่าเมื่อเทียบกับ HDD



รูป 4.2 iostat ของเครื่อง i3-2100 โดย ใช้ข้อมูลชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบ SSD กับ HDD

จากกราฟ iostat ของเครื่อง i3-2100 ในช่วงแรกจะไม่มีการอ่านข้อมูลเนื่องจากมีแคชทำให้กราฟของ Read Throughput เป็น 0 MB/s ในขั้นตอน Read และข้อมูลในส่วนที่ถูกอ่านจะถูกเขียนกลับไปยังหน่วยความจำรองทำให้มีค่า Write Throughput ขึ้นลง

ในขั้นตอน Merge จะพบว่า CPU utilization เพิ่มขึ้นสูงและจะเหลือ 0 MB /s เมื่อจบขั้นตอนการของ Merge ในขั้นตอนนี้ กราฟ SSD มี CPU utilization จาก 28% ขึ้นไปถึง 52% ในช่วง 28-30 วินาที และในช่วง HDD มี CPU utilization 55% ที่ 40 วินาทีในช่วงเริ่มต้นของขั้นตอน merge และ

ลดลงเหลือ 0% หลังจากนั้นเพราะเนื่องจากต้องเขียนข้อมูลที่ได้จากการผสานข้อมูล จนกว่าจะเสร็จ ซึ่ง HDD ทำงานได้

ในขั้นตอน Write จะเป็นการอ่านและเขียนข้อมูลสลับกันโดยอ่านข้อมูลที่เรียงลำดับแล้วลงในบัฟเฟอร์ขนาด 1024 MB ทำให้ Read Throughput เพิ่มขึ้นเมื่อบัฟเฟอร์เต็มจะทำการเขียนข้อมูลกลับลงในหน่วยความจำรอง ทำให้ มีค่า Write Throughput เพิ่มขึ้นและ Read Throughput ลดลง และบัฟเฟอร์หมดมันจะดึงข้อมูลส่วนถัดไปมาผสานใหม่ทำให้ค่า Read Throughput สูงขึ้นและ Write Throughput ลดลง

ในส่วนของ iowait จะเพิ่มขึ้นและลดลงเป็นช่วงๆโดยกราฟจะไปในทิศทางเดียวกับ Write throughput นั่นคือเมื่อในขณะที่เครื่องกำลังเขียนข้อมูล ก็จะมี iowait ที่เพิ่มขึ้นและ iowait จะลดลงเมื่อค่า Read Throughput สูงขึ้น โดย SSD มีค่า iowait น้อยกว่า SSD

4.3 เปรียบเทียบ แรม 8 GB กับ 16 GB

ขนาดของแรมมีผลโดยตรงกับการแคชไฟล์โดยระบบปฏิบัติการ ในกรณีที่ข้อมูลที่จะใช้จัดเรียงมีขนาดน้อยกว่าขนาดของหน่วยความจำหลักที่เหลืออยู่ (free memory) ระบบปฏิบัติการจะสามารถเก็บข้อมูลของไฟล์นั้นไว้บนแคชในหน่วยความจำหลักได้ทั้งหมด ซึ่งส่งผลให้การอ่านข้อมูลเดิมในครั้งถัดไปได้ความเร็วที่ดีขึ้นมาก

ตาราง 4.6 เวลาในการรันโดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัว เรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แบบมีแคช โดยเปรียบเทียบ แรม 8GB กับ 16GB

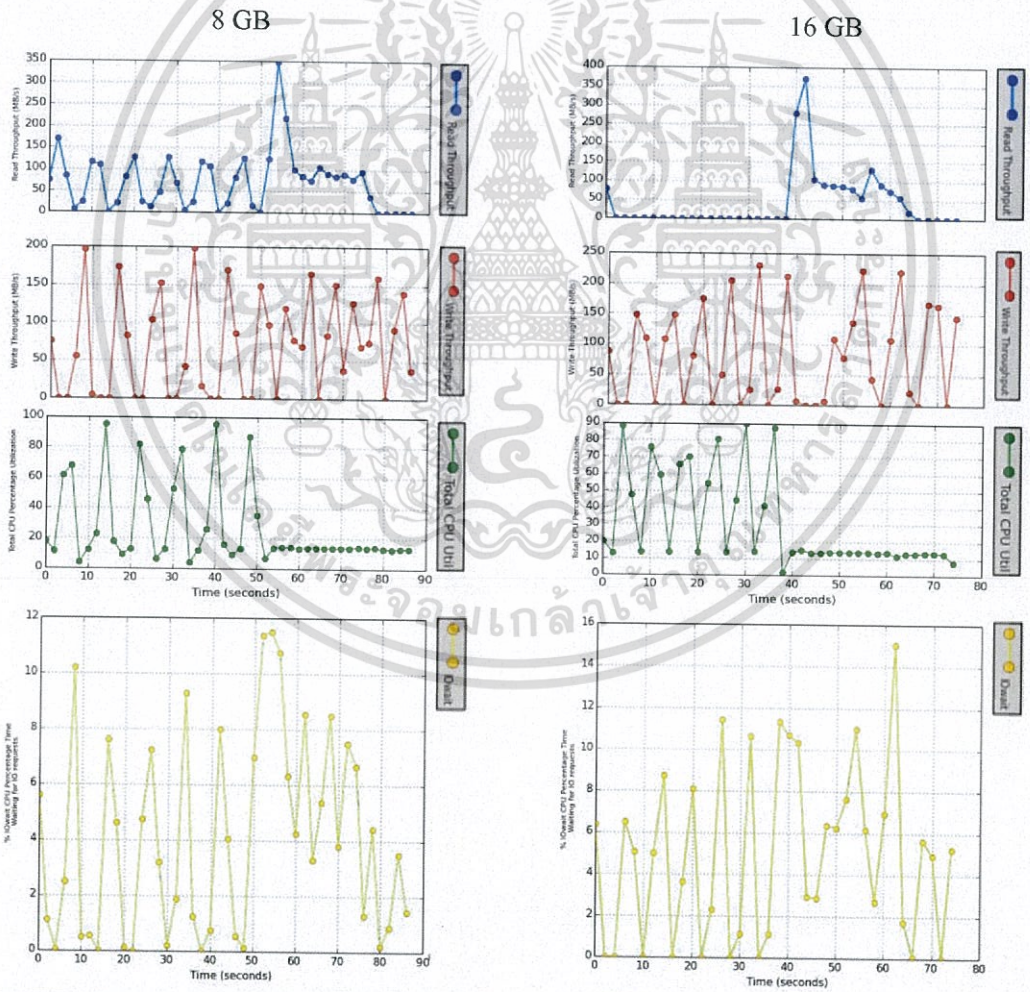
เครื่อง	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	8GB	16GB	8GB	16GB	8GB	16GB	8GB	16GB
Read (Sort)	58.80	48.72	46.07	33.33	53.44	51.79	83.60	72.72
Merge	7.30	6.67	4.89	4.85	1.47	1.47	10.44	10.29
Write	43.70	44.44	35.71	36.26	69.35	71.09	139.64	141.16
Total	109.80	99.83	86.66	74.44	124.26	124.35	233.67	224.17

จากผลการทดลองพบว่าเวลาที่ใช้ในส่วนของ Merge และ Write ไม่ต่างกันมากแต่เวลาของ Read เมื่อใช้แรม 16 GB เร็วกว่าแรม 8 GB ประมาณ 10 วินาที

ตาราง 4.7 อัตราส่วนความเร็ว โดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัว เรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แบบมีแคช โดยเปรียบเทียบ แรม 8GB กับ 16GB

Media/Time(sec)	Ratio 8GB:16GB speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	1.21	1.38	1.03	1.15
Merge	1.10	1.01	1.00	1.01
Write	0.98	0.98	0.98	0.99
Total	1.10	1.16	1.00	1.04

จากอัตราส่วนความเร็วในขั้นตอน Read แรม 16 GB เร็วกว่า แรม 8 GB ประมาณ 1.2 เท่า



รูป 4.3 iostat ของเครื่อง i7-2600 โดย ใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed แรม 16GB และมีแคช โดยเปรียบเทียบแรม 8GB กับ 16GB

จากกราฟพบว่ามีแรม 8 GB มี Read Throughput เกิดขึ้นในขั้นตอน Read เนื่องจาก ไม่สามารถเก็บข้อมูลทั้งหมดลงในแรมที่มีได้ ทำให้ต้องเข้าถึงข้อมูลบางส่วนจากหน่วยความจำรองส่งผลให้ใช้เวลาช้ากว่า แรม 16 GB

ตาราง 4.8 เวลาในการรันโดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัว เรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แบบมีแคช โดยเปรียบเทียบ แรม 8GB กับ 16GB

เครื่อง	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	8GB	16GB	8GB	16GB	8GB	16GB	8GB	16GB
Read (Sort)	41.49	28.87	38.61	25.80	41.46	38.68	57.98	47.50
Merge	3.31	2.98	3.15	3.40	0.97	0.82	5.84	5.74
Write	32.59	34.30	26.89	26.80	62.26	59.79	121.12	121.27
Total	77.47	66.16	68.71	56.01	104.69	99.30	185.34	174.52

จากผลการทดลอง เวลาในขั้นตอน Read ของแรม 16 GB เร็วกว่าแรม 8 GB ค่อนข้างมาก แต่เวลาของขั้นตอน Merge และ Write มีเวลาที่ไม่ต่างกันมากนักโดยสามารถเทียบอัตราส่วนความเร็วได้ดังนี้

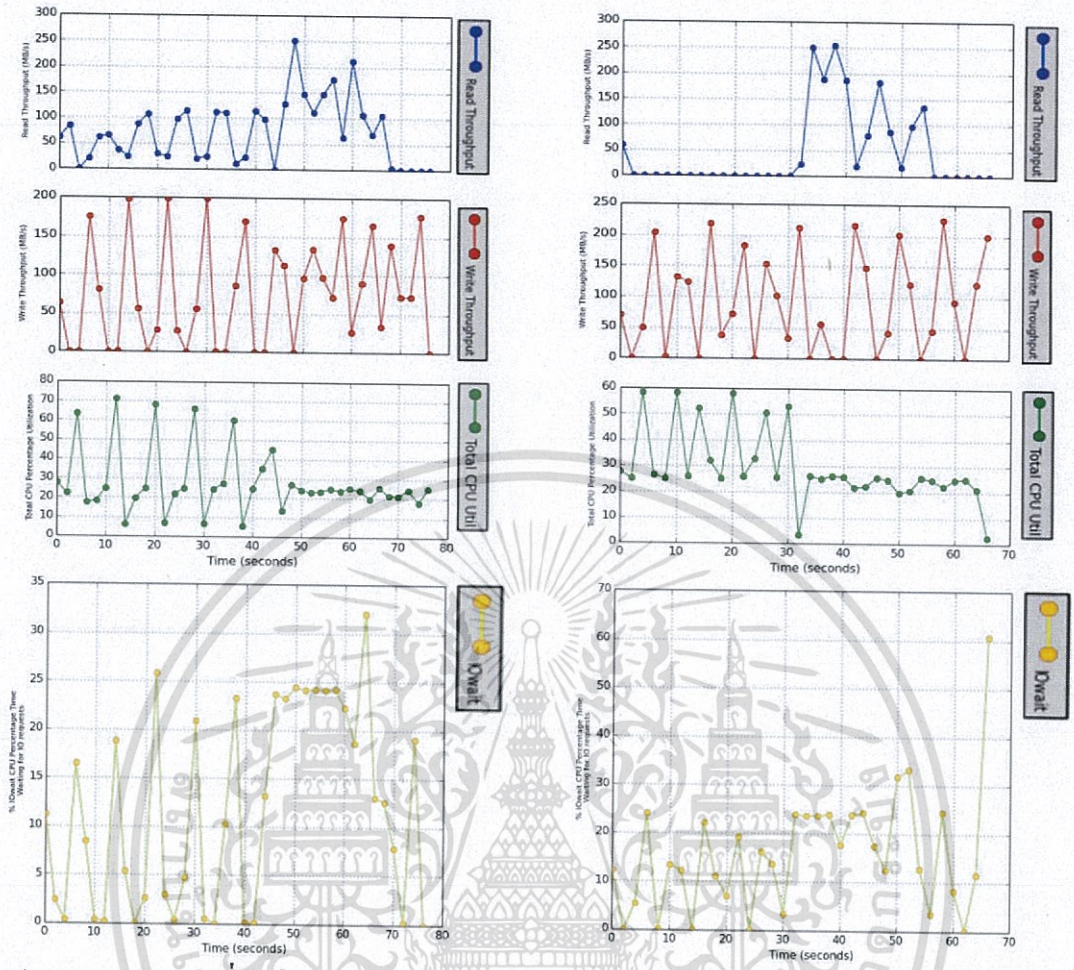
ตาราง 4.9 อัตราส่วนความเร็ว โดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัว เรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แบบมีแคช โดยเปรียบเทียบ แรม 8GB กับ 16GB

Media/Time(sec)	Ratio 8GB:16GB speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	1.44	1.45	1.07	1.22
Merge	1.11	0.93	1.18	1.01
Write	0.95	1.00	1.04	1.00
Total	1.17	1.23	1.05	1.06

จากผลอัตราส่วนความเร็วในขั้นตอน Read พบว่าแรม 16 GB เร็วกว่า 8 GB ประมาณ 1.3 เท่า แต่ในขั้นตอน Merge และ Write พบว่าไม่แตกต่างกันมากนัก

8 GB

16 GB



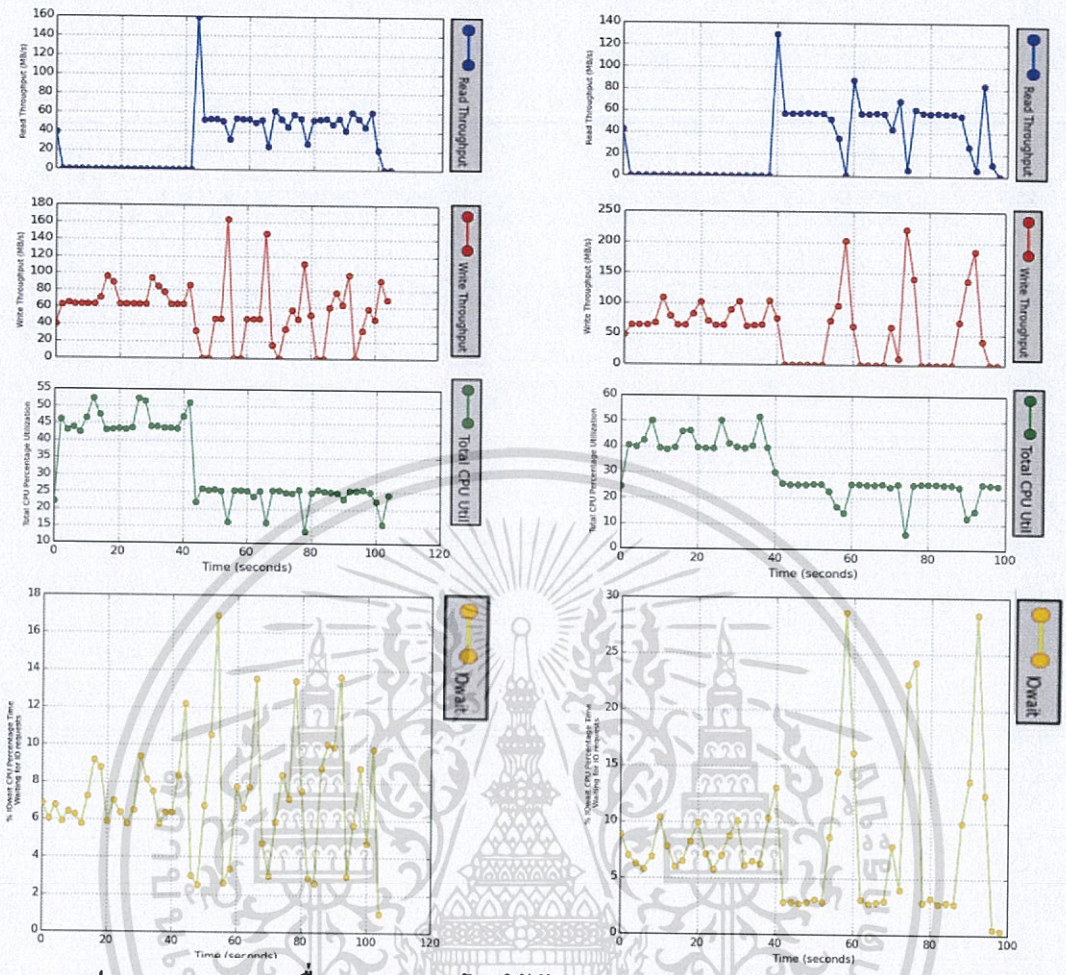
รูป 4.4 iostat ของเครื่อง i3-2100 โดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัว เรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แบบมีแคช โดยเปรียบเทียบ แรม 8GB กับ 16GB

จากผลกราฟ จะพบว่าแรมขนาด 8GB ยังต้องอ่านข้อมูลจากหน่วยความจำรอง แม้จะมีแคช เพราะเนื่องจากไม่สามารถเก็บข้อมูลทั้งหมดไว้บนแรมได้แบบ 16GB จึงทำให้ มี Read Throughput ขึ้นมาตอนต้นของแรม 8GB

แต่เนื่องจากผลการทดลองของ A6-3650 ไม่ค่อยมีความแตกต่างกันมากนักจึงได้นำผลการทดลอง iostat มาเปรียบเทียบ

8 GB

16 GB



รูป 4.5 iostat ของเครื่อง A6-3650 โดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัว เรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แบบมีแคช โดยเปรียบเทียบ แรม 8GB กับ 16GB

จากกกราฟพบว่า แรม 8 GB สามารถเก็บข้อมูลลงในแรมได้ทั้งหมดทำให้ไม่ต้องอ่านข้อมูลจากหน่วยความจำรอง ทำให้ขั้นตอน Read มีเวลาเท่ากับแรม 16 GB

4.4 เปรียบเทียบแคชกับไม่แคช

ในการทดลองจะทำการทดลองทั้งในกรณีที่ไฟล์ถูกแคชและ ไม่ถูกแคชการ clear cache ทำด้วยคำสั่ง "sync && echo 3 | sudo tee /proc/sys/vm/drop_caches"

ตาราง 4.10 เวลาในการรันโดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แรม 16GB โดยเปรียบเทียบ แบบมีแคช กับ ไม่มีแคช

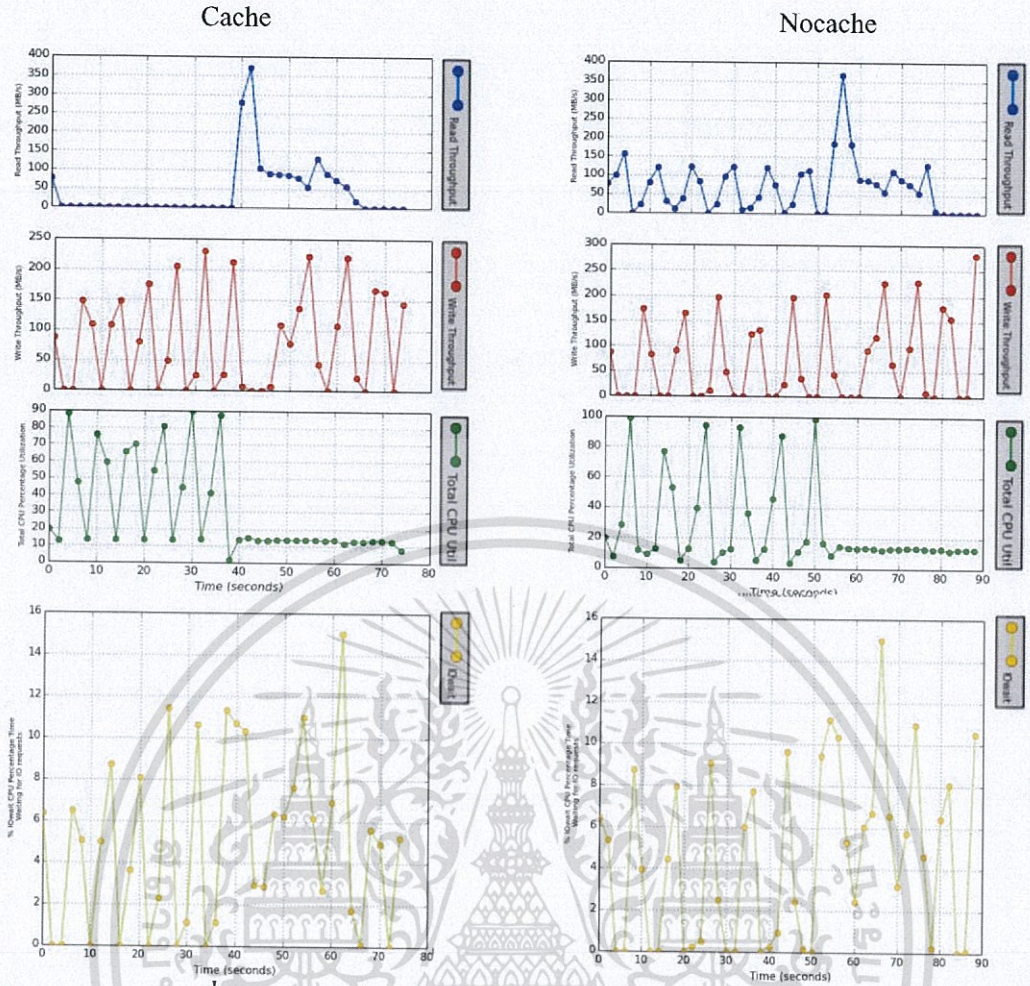
CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	Cache	Nocache	Cache	Nocache	Cache	Nocache	Cache	Nocache
Read (Sort)	48.72	59.15	33.33	46.94	51.79	63.06	72.72	83.52
Merge	6.67	6.36	4.85	4.85	1.47	1.46	10.29	10.30
Write	44.44	43.68	36.26	35.57	71.09	74.69	141.16	140.16
Total	99.83	109.19	74.44	87.36	124.35	139.21	224.17	233.98

จากผลการทดลองพบว่าแบบมีแคชสามารถช่วยอ่านข้อมูลได้เร็วกว่าแบบไม่มีแคชประมาณ 10 วินาทีแต่ขั้นตอนการ Merge และ Write พบว่าแบบมีแคชและไม่มีแคชใช้เวลาเท่ากัน

ตาราง 4.11 อัตราส่วนความเร็วโดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบ แบบมีแคช กับไม่มีแคช

Speedup	Ratio Nocache:Cache speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	1.21	1.41	1.22	1.15
Merge	0.95	1.00	0.99	1.00
Write	0.98	0.98	1.05	0.99
Total	1.09	1.17	1.12	1.04

จากผลการเปรียบเทียบอัตราส่วนความเร็วพบว่าเครื่อง i7-2600 ในขั้นตอน Read มีอัตราส่วนความเร็วที่ดีกว่าเครื่องเล็กน้อยเมื่อเทียบกับเครื่องอื่น



รูป 4.6 iostat ของเครื่อง i7-2600 โดย ใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แรม 16GB โดยเปรียบเทียบ แบบมีแคช กับ ไม่มีแคช

จากกราฟจะเห็นความแตกต่างในขั้นตอน Read คือแบบมีแคชจะไม่มี Read Throughput เกิดขึ้น เพราะมีการเก็บข้อมูลไว้บนแรมทำให้อ่านข้อมูลบนแรมแทนหน่วยความจำรองแต่การอ่านข้อมูลแบบไม่มีแคชจะเกิด Read Throughput เมื่อเทียบ CPU utilization พบว่าแบบมีแคชจะสูงสุดที่ 90% และแบบไม่มีแคชจะสูงสุดที่ 100%

ในขั้นตอน Merge แบบมีแคชพบว่า CPU utilization เพิ่มขึ้นจาก 40% เป็น 90% แต่แบบไม่มีแคชมี CPU utilization เพิ่มขึ้นจาก 12% เป็น 100% โดยในช่วงนี้จะไม่มีการอ่านเขียนข้อมูลจนกว่าจะทำการเรียงข้อมูลเสร็จทำการเขียนข้อมูลที่ merge ได้จากแรมลงหน่วยความจำรอง ทำให้ Write Throughput และ iowait สูงขึ้นพร้อมกับ CPU utilization ลดลง

ในขั้นตอน Write การมีแคชหรือไม่มีแคชจะไม่เกิดผลกระทบกับการเขียนข้อมูล

ตาราง 4.12 เวลาในการรันโดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB โดยเปรียบเทียบ แบบมีแคช กับ ไม่มีแคช

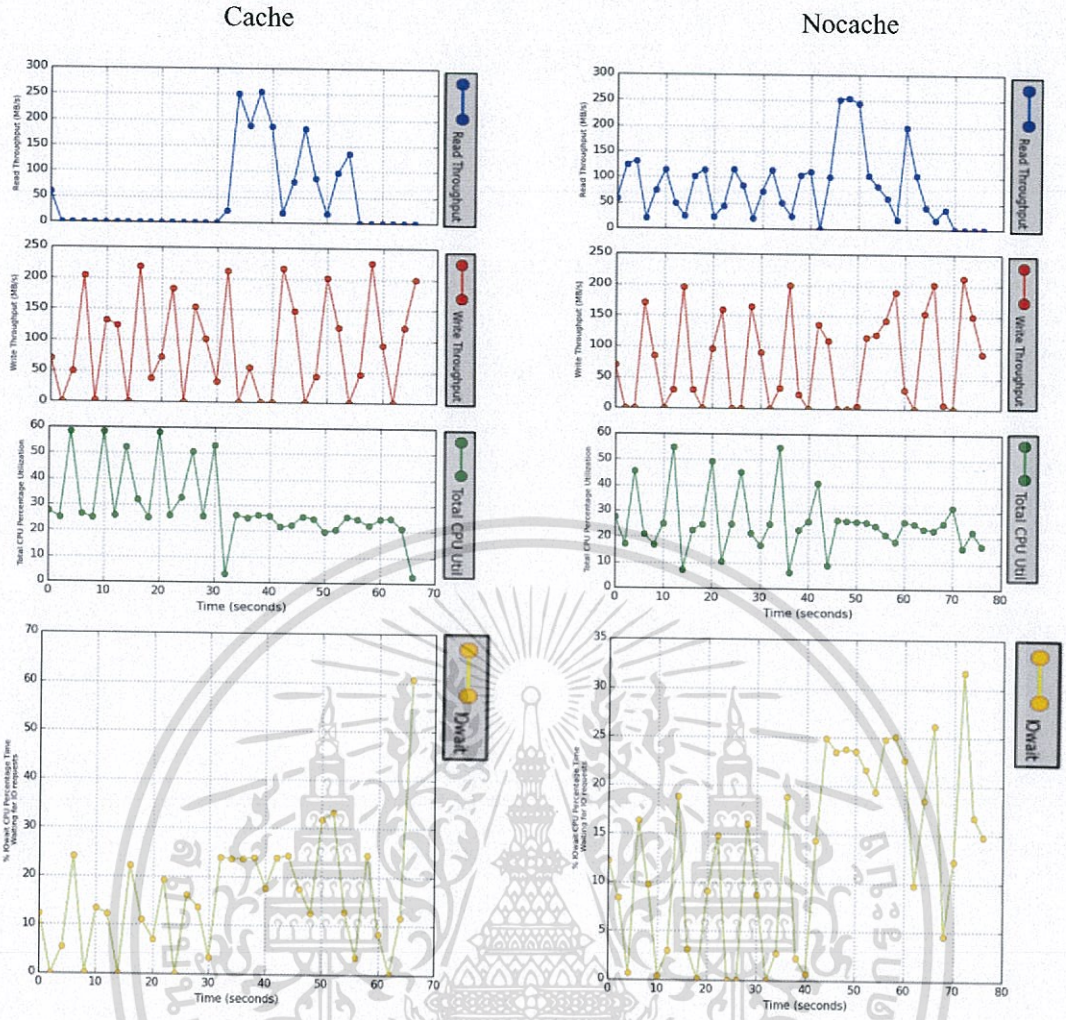
CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	Cache	Nocache	Cache	Nocache	Cache	Nocache	Cache	Nocache
Read (Sort)	28.87	39.33	25.80	38.33	38.68	50.02	47.50	58.25
Merge	2.98	3.26	3.40	3.73	0.82	0.82	5.74	5.37
Write	34.30	33.38	26.80	27.84	59.79	59.85	121.27	122.45
Total	66.16	76.11	56.01	70.16	99.30	110.74	174.52	186.25

จากผลการทดลอง พบว่าแคชช่วยเพิ่มความเร็วในขั้นตอน Read เป็นหลักแต่เวลาในขั้นตอน Merge และ Write นั้นพบว่าแบบมีแคชและไม่มีแคชใช้เวลาไม่ต่างกันมากโดยสามารถเทียบอัตราส่วนความเร็วได้ดังนี้

ตาราง 4.13 อัตราส่วนความเร็วโดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบ แบบมีแคช กับ ไม่มีแคช

Speedup	Ratio Nocache:Cache speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	1.36	1.49	1.3	1.23
Merge	1.09	1.1	0.99	0.94
Write	0.97	1.04	1.00	1.01
Total	1.15	1.25	1.12	1.07

จากผลการเปรียบเทียบอัตราส่วนความเร็วพบว่าความเร็วในขั้นตอน Read เพิ่มขึ้นแต่ขั้นตอน Merge และ Write ไม่ได้เปลี่ยนแปลงมากนักจึงสามารถบอกได้ว่า การมีแคชหรือไม่มีแคชไม่ได้มีผลกับเวลาในขั้นตอน Merge และ Write



รูป 4.7 iostat ของเครื่อง i3-2100 โดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB โดยเปรียบเทียบ แบบมีแคช กับ ไม่มีแคช

จากผลของ iostat จะเห็นความแตกต่างได้ชัดเจนในขั้นตอน Read เพราะแบบมีแคช Read Throughput จะเป็น 0 MB/s แต่แบบไม่มีแคช จะไม่ใช่ 0 MB/s เพราะการอ่านข้อมูลแบบมีแคชจะไปอยู่บนแรมทำให้ไม่ต้องไปอ่านบนหน่วยความจำรอง และกราฟ CPU utilization ของแบบมีแคชจะสูงกว่าแบบไม่มีแคชเพราะข้อมูลสามารถเข้าถึงได้เร็วกว่าทำให้การสามารถทำการเรียงข้อมูลได้เร็วกว่า

สำหรับขั้นตอน Merge แบบมีแคช มี CPU utilization จาก 28% ขึ้นไปถึง 52% ในช่วง 28-30 วินาทีและแบบไม่มีแคช CPU utilization จาก 35% ขึ้นไป 40% ในช่วง 39 วินาทีถึง 42 วินาที

ในขั้นตอน Write จะเริ่มอ่านข้อมูลด้วย Read throughput และสลับกับ Write Throughput โดย CPU utilization จะค่อนข้างคงที่ แบบมีแคชอยู่ที่ 20-27% Write Throughput และแบบไม่มีแคช 19-31%

iowait ตอนอ่านข้อมูลแบบมีแคชจะสลับขึ้นลงในช่วง 0-24% และ แบบไม่มีแคชอยู่ในช่วง0-18% แต่เนื่องจากแบบไม่มีแคชใช้เวลานานกว่าถึง 10 วินาทีในขั้นตอน Read

4.5 เปรียบเทียบประเภทข้อมูล Uint32 กับ Double

ข้อมูลประเภท Uint32 มีขนาด 4 bytes ต่อหนึ่งตัว ส่วน Double มีขนาด 8 bytes ต่อหนึ่งตัว การเปลี่ยนประเภทตัวแปรจึงเป็นการทดสอบประสิทธิภาพการทำงานเมื่อขนาดของข้อมูลในแต่ละ element เปลี่ยนไป โดยที่จำนวนของข้อมูลยังคงเดิม

กราฟแสดงการรันข้อมูลบน SSD ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB แบบมีแคช โดยเปรียบเทียบ ข้อมูลชนิด Uint32 กับ Double

ตาราง 4.14 เวลาในการรันโดยใช้ข้อมูลบน SSD ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แรม 16GB แบบมีแคช โดยเปรียบเทียบ ข้อมูลชนิด Uint32 กับ Double

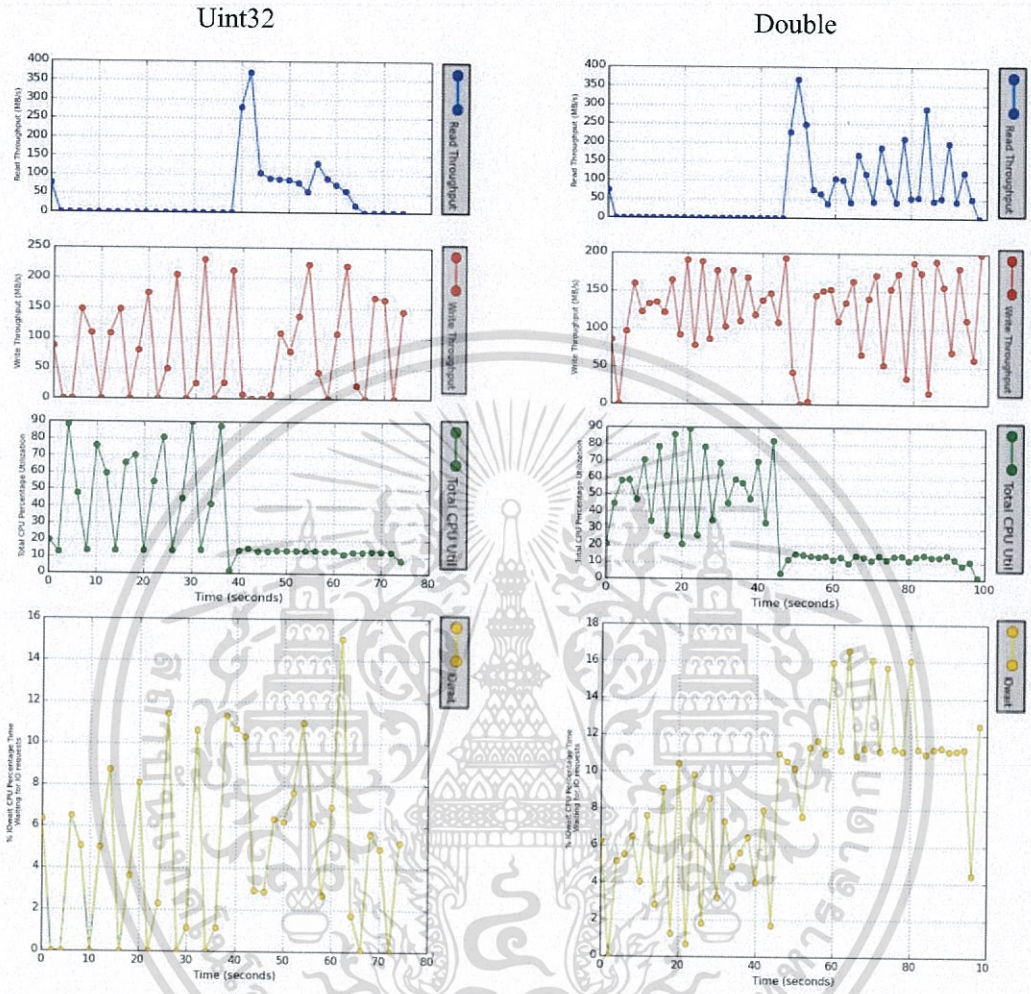
CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	Uint32	Double	Uint32	Double	Uint32	Double	Uint32	Double
Read (Sort)	48.72	54.41	33.33	42.28	51.79	59.39	72.72	86.65
Merge	6.67	4.97	4.85	4.21	1.47	1.19	10.29	7.22
Write	44.44	61.31	36.26	51.75	71.09	103.85	141.16	195.37
Total	99.83	120.69	74.44	98.24	124.35	164.44	224.17	289.24

จากผลการทดลองพบว่าข้อมูลประเภท Uint32 ใช้เวลาในการอ่านและการเขียนน้อยกว่าข้อมูลประเภท Double แต่ Double ใช้เวลาในขั้นตอน Merge เร็วกว่า Uint32 เล็กน้อย

ตาราง 4.15 อัตราส่วนความเร็ว โดยใช้ข้อมูลบน SSD ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แรม 16GB แบบมีแคช โดยเปรียบเทียบ ข้อมูลชนิด Uint32 กับ Double

Speedup	Ratio Double:Uint32 speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	1.12	1.27	1.15	1.19
Merge	0.75	0.87	0.81	0.70
Write	1.38	1.43	1.46	1.38
Total	1.21	1.32	1.32	1.29

จากผลการเปรียบเทียบอัตราส่วนความเร็วพบว่าข้อมูล Uint32 เร็วกว่า Double โดยเฉพาะในขั้นตอน Write และขั้นตอน Merge Double เร็วกว่า Uint32



รูป 4.8 iostat ของเครื่อง i7-2600 โดย ใช้ข้อมูลบน SSD ชนิด ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random บัฟเฟอร์และมีแคช มีขนาด 1024 MB แรม 16GB โดยเปรียบเทียบ Uint32 กับ Double

จากกราฟพบว่าในขั้นตอน Read ข้อมูลประเภท Double มี Write Throughput ที่สูงกว่า Uint32 เพราะข้อมูลประเภท Double มีขนาดข้อมูลมากกว่า Uint32 ถึง 2 เท่า ทำให้ iowait ของ Double สูงกว่า Uint32

ในขั้นตอน Merge พบว่าแบบมีแคชใช้ CPU utilization ขึ้นสูงถึง 90% ในวินาทีที่ 36 และแบบไม่มีแคช CPU utilization อยู่ที่ 80% ในวินาทีที่ 44 และเมื่อ CPU utilization ลดลง Write Throughput ก็ขึ้นสูง เพื่อเขียนข้อมูลที่ได้จากการผสานเสร็จแล้ว

ในขั้นตอน Write Uint32 และ Double มี Read Throughput ที่สูงถึง 370 MB/s เพื่ออ่านข้อมูลลงบัฟเฟอร์จนเต็ม เมื่อบัฟเฟอร์เต็มจะทำการเขียนข้อมูลลงในหน่วยความจำรองทำให้ Write Throughput ต่ำลงและ Read Throughput สูงขึ้น ซึ่ง ข้อมูล Double มีขนาดข้อมูลที่มากกว่า Uint32 ทำให้ต้องเขียนอ่านข้อมูลหลายรอบ CPU utilization สามารถทำได้เท่ากันแต่ iowait ของ Double คงที่อยู่ 11% และขึ้นสูงไปที่ 16% ซึ่งมีค่าสูงกว่า Uint32

ตาราง 4.16 เวลาในการรันโดยใช้ข้อมูลบน SSD ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB แบบมีแคช โดยเปรียบเทียบ ข้อมูลชนิด Uint32 กับ Double

CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	Uint32	Double	Uint32	Double	Uint32	Double	Uint32	Double
Read (Sort)	28.87	35.74	25.80	35.99	38.68	42.66	47.50	53.01
Merge	2.98	5.52	3.40	4.84	0.82	0.92	5.74	4.39
Write	34.30	55.08	26.80	48.93	59.79	94.30	121.27	154.83
Total	66.16	96.34	56.01	89.76	99.30	137.88	174.52	212.23

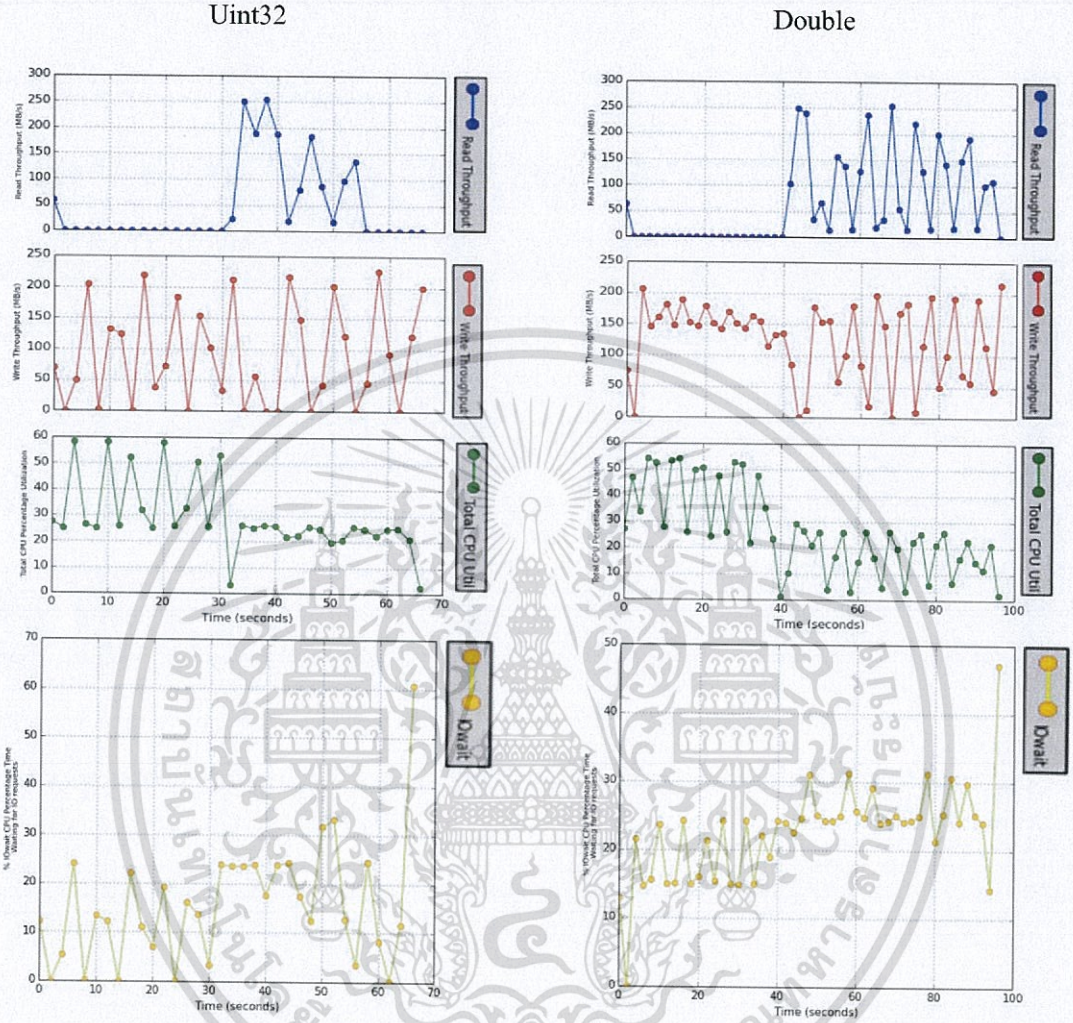
จากผลการทดลอง พบว่าเวลาของขั้นตอน Read และ Write ในส่วนของ Uint32 ใช้เวลาน้อยกว่า Double เนื่องจาก Uint32 มีขนาด 32 bit แต่ Double มีขนาด 64 bit ส่งผลให้ขนาดข้อมูล Double มากกว่า Uint32 ถึง 2 เท่า จึงมีผลกระทบกับการเขียนอ่านโดยตรง

เวลาขั้นตอน Merge พบว่า Uint32 ใช้เวลาน้อยกว่า Double เช่นกันแต่เครื่อง FX-8320 ได้ผลการทดลองตรงกันข้ามโดยสามารถเทียบอัตราส่วนความเร็วได้ดังนี้

ตาราง 4.17 อัตราส่วนความเร็ว โดยใช้ข้อมูลบน SSD ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB แบบมีแคช โดยเปรียบเทียบ ข้อมูลชนิด Uint32 กับ Double

Speedup	Ratio Double:Uint32 speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	1.24	1.39	1.10	1.12
Merge	1.85	1.42	1.12	0.76
Write	1.61	1.83	1.58	1.28
Total	1.46	1.61	1.39	1.22

จากผลการเปรียบเทียบอัตราส่วนพบว่า Uint32 เร็วกว่า Double ทั้งขั้นตอน Read , Merge และ Write โดยผลการทดลองส่วนใหญ่ ขั้นตอน Write ดีขึ้นมากที่สุด



รูป 4.9 iostat ของเครื่อง i3-2600 โดย ใช้ข้อมูลบน SSD ชนิด ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed บัฟเฟอร์และมีแคช มีขนาด 1024 MB แรม 16GB โดยเปรียบเทียบ Uint32 กับ Double

จากกราฟ ขั้นตอน Write พบว่า Double มี Write Throughput ที่สูงกว่า Uint32 เพราะข้อมูล Double มีขนาดใหญ่กว่า Uint32 นอกจากนี้ iowait ของ Uint32 ยังน้อยกว่า Double อีกด้วย

ในขั้นตอนการ Merge มี CPU utilization สูงขึ้นโดย Double ใน35-40 วินาที CPU utilization จาก 35% ไป 40% และหลังจากนั้น CPU จะตกลงเพราะนำข้อมูลที่ได้จากการผสานเขียนลงในหน่วยความจำรอง

ในขั้นตอน Write Uint32 และ Double มี Read Throughput และ Write Throughput ที่ขึ้นลงสลับกัน แต่ต่างตรงที่ Uint32 ใช้เวลาน้อยกว่า Double

4.6 เปรียบเทียบข้อมูลจำนวน 800ล้านตัว กับ 1600ล้านตัว

การทดลองใช้ข้อมูล 800 ล้านตัว และ 1600ล้านตัวเพื่อทดสอบทั้งกรณีที่ขนาดของข้อมูลทั้งหมดสามารถบรรจุอยู่ในหน่วยความจำหลักได้ เช่น กรณี Uint32 จำนวน 800 ล้านตัว จะมีขนาดข้อมูลทั้งหมดเป็น 3.2 กิกะไบต์ซึ่งสามารถบรรจุลงในหน่วยความจำหลักขนาด 8 กิกะไบต์ในการทดลองได้ และกรณีที่ขนาดของข้อมูลทั้งหมดใหญ่กว่าขนาดของหน่วยความจำหลัก เช่น กรณี Double จำนวน 1600 ล้านตัว จะมีขนาดข้อมูลทั้งหมดเป็น 12.8 กิกะไบต์ซึ่งไม่สามารถบรรจุลงในหน่วยความจำหลักขนาด 8 กิกะไบต์ในการทดลองได้

ตาราง 4.18 เวลาในการรันโดยใช้ข้อมูลบน SSD ชนิด Uint32 เรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แรม 16GB แบบมีแคช โดยเปรียบเทียบ ขนาดข้อมูล 800 ล้านตัว กับ 1600 ล้านตัว

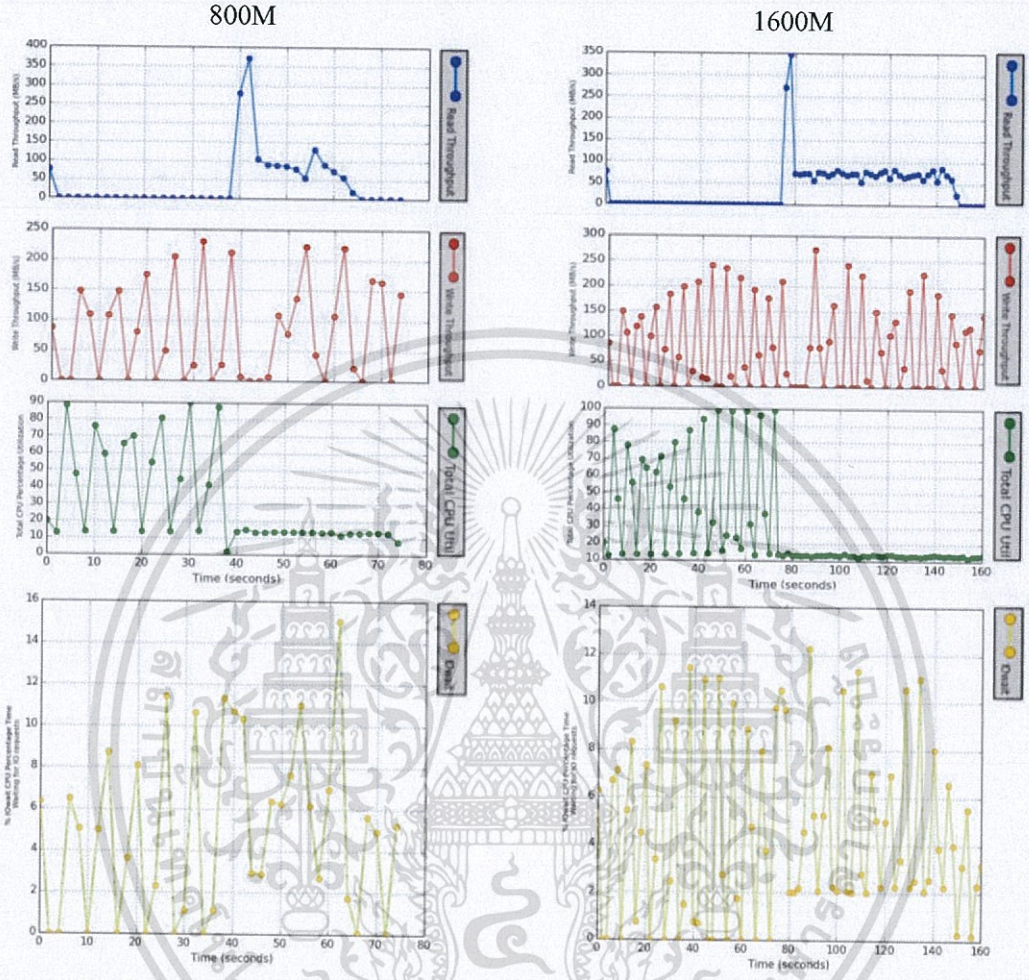
CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	800M	1600M	800M	1600M	800M	1600M	800M	1600M
Read (Sort)	48.72	103.84	33.33	69.97	51.79	103.32	72.72	155.72
Merge	6.67	6.45	4.85	4.35	1.47	1.41	10.29	10.19
Write	44.44	101.30	36.26	86.58	71.09	158.77	141.16	332.89
Total	99.83	211.59	74.44	160.89	124.35	263.50	224.17	498.80

ในส่วนของผลการทดลองพบว่าข้อมูล 1600 ล้านตัวใช้เวลามากกว่าข้อมูล 800 ล้านตัวในขั้นตอนการอ่านและการเขียนแต่เวลาในขั้นตอน Merge ใช้เวลาไม่แตกต่างกัน

ตาราง 4.19 อัตราส่วนความเร็วโดยใช้ข้อมูลบน SSD ชนิด Uint32 เรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แรม 16GB แบบมีแคช โดยเปรียบเทียบ ขนาดข้อมูล 800 ล้านตัว กับ 1600 ล้านตัว

Speedup	Ratio 1600M:800M speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	2.13	2.10	2.00	2.14
Merge	0.97	0.90	0.96	0.99
Write	2.28	2.39	2.23	2.36
Total	2.12	2.16	2.12	2.23

จากผลการเปรียบเทียบสัดส่วนพบว่าสัดส่วนของขั้นตอน Read และ Write ข้อมูล 1600 ล้าน เร็วกว่า 800 ล้านประมาณ 2 เท่า แต่ในขั้นตอน Merge ไม่แตกต่างกันมาก



รูป 4.10 iostat ของเครื่อง i7-2600 โดย ใช้ข้อมูลบน SSD ชนิด Uint32 เรียงข้อมูลแบบ Random บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบข้อมูล 800 ล้านตัว และ 1600 ล้านตัว

จากกราฟข้อมูลพบว่าในขั้นตอน Read พบว่า ข้อมูลขนาด 800 ล้านตัวมี CPU utilization สูงสุดที่ 90% แต่ข้อมูลขนาด 1600 ล้านตัวมี CPU utilization สูงสุดที่ 100% และ Write Throughput สูงในข้อมูล 1600 ล้านตัว เพราะข้อมูลมีจำนวนมาก

ในขั้นตอนการ Merge พบว่า CPU utilization ของ ข้อมูล 1600 ล้านขึ้นสูงถึง 100% ในวินาทีที่ 72 เพื่อทำการผสานข้อมูล และลดลงเหลือ 0 พร้อมกับ Write Throughput ที่สูงขึ้นเพื่อเขียนข้อมูลที่ได้ลงหน่วยความจำรอง

ในขั้นตอนการเขียนข้อมูลจะใช้ CPU utilization ระดับ 10% แต่จะมี iowait ที่สลับขึ้นลงซึ่งอยู่ในระดับ 0-15% เพราะต้องสลับกันอ่านเขียนข้อมูล

ตาราง 4.20 เวลาในการรันโดยใช้ข้อมูลบน SSD ชนิด Uint32 เรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB แบบมีแคช โดยเปรียบเทียบ ขนาดข้อมูล 800 ล้านตัว กับ 1600 ล้านตัว

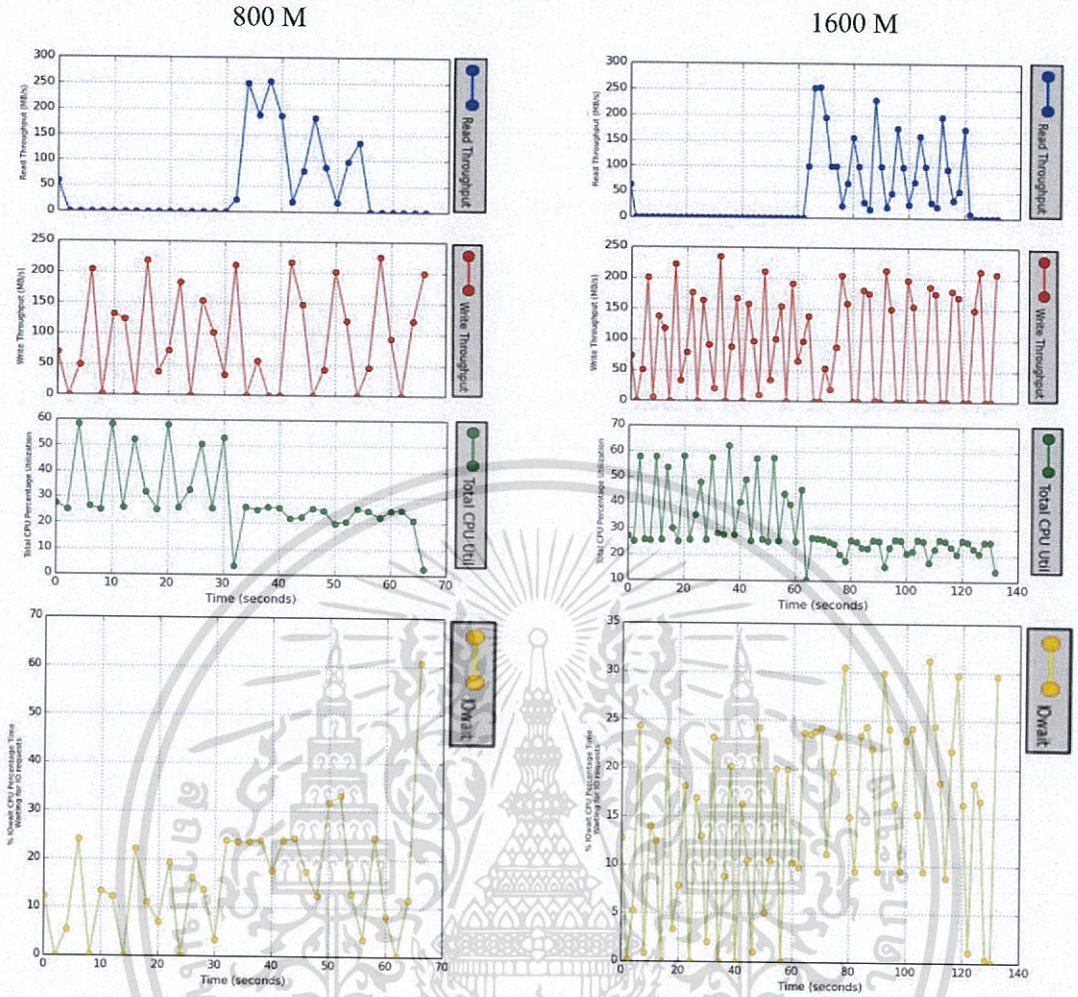
CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	800M	1600M	800M	1600M	800M	1600M	800M	1600M
Read (Sort)	28.87	60.38	25.80	55.05	38.68	77.80	47.50	99.14
Merge	2.98	2.90	3.40	3.05	0.82	0.88	5.74	5.55
Write	34.30	70.09	26.80	57.39	59.79	176.96	121.27	292.16
Total	66.16	133.37	56.01	115.49	99.30	255.63	174.52	396.86

จากผลการทดลองพบว่า เวลาในขั้นตอน Read และWrite ข้อมูลขนาด 800 ล้านตัว เร็วกว่า 1600 ล้านตัว ประมาณ 2 เท่า แต่เวลาในขั้นตอน Merge ไม่ได้ต่างกันมากนักซึ่งส่งผลให้เวลารวม 800 ล้านตัว ใช้เวลาน้อยกว่า 1600 ล้านตัวประมาณ 2 เท่า

ตาราง 4.21 อัตราส่วนความเร็วโดยใช้ข้อมูลบน SSD ชนิด Uint32 เรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB แบบมีแคช โดยเปรียบเทียบ ขนาดข้อมูล 800 ล้านตัว กับ 1600 ล้านตัว

Speedup	Ratio 1600M:800M speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	2.09	2.13	2.01	2.09
Merge	0.97	0.90	1.07	0.97
Write	2.04	2.14	3.0	2.41
Total	2.02	2.06	2.57	2.27

จากผลการปรียบเทียบอัตราส่วนจะเห็นได้ชัดว่า เวลาในส่วนของ Read และ Write ของข้อมูล 800 ล้านตัวเร็วกว่าข้อมูล 1600 ล้านตัวอยู่ประมาณ 2 เท่า แต่อัตราส่วนความเร็วของขั้นตอน Merge พบว่าข้อมูล 1600 ล้านตัวเร็วกว่าข้อมูล 1800 ล้านตัวเล็กน้อย



รูป 4.11 iostat ของเครื่อง i3-2100 โดย ใช้ข้อมูลบน SSD ชนิด Uint32 เรียงข้อมูลแบบ Reversed บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบข้อมูล 800 ล้านตัว และ 1600 ล้านตัว

จากกราฟจะเห็นได้ว่า ในขั้นตอน Read พบว่ากราฟ Read Throughput , Write Throughput , CPU utilization มีลักษณะเดียวกันคือขึ้นลงสลับไปมา แต่ต่างตรงที่ข้อมูล 1600 ล้านตัวใช้เวลานานกว่า ข้อมูล 800 ล้านตัวอยู่ประมาณ 30 วินาที หรือ ใช้เวลาอ่านเป็น 2 เท่าของข้อมูล 800 ล้านตัว

ในขั้นตอน Merge โดยใช้ข้อมูล 800 ล้านตัวมี CPU utilization จาก 28% ขึ้นไปถึง 52% ในช่วง 28-30 วินาทีและ1600 ล้านตัวมี CPU utilization จาก 27% ถึง 46%

ในขั้นตอน Write กราฟ Read Throughput จะขึ้นลงสลับกับ Write Throughput เพราะต้องอ่านข้อมูลไปใส่บัฟเฟอร์ส่งผลให้ Read Throughput สูง Write Throughput ลดลงขึ้นเมื่อบัฟเฟอร์เต็มจะเขียนข้อมูลที่เรียงลำดับแล้วลงในหน่วยความจำรองช่วงนี้ Write Throughput สูงขึ้น และ Read

Throughput ลดลง ซึ่งข้อมูล 1600 ล้านตัวใช้เขียนนานกว่า 800 ล้านตัวประมาณ 40 วินาที หรือใช้เวลาเขียนเป็น 2 เท่าของข้อมูล 800 ล้านตัว

สำหรับ iowait เมื่อ Write Throughput สูงขึ้น iowait ก็จะสูงขึ้นตามและจะลดลงเมื่อ Read Throuhput สูงขึ้น

4.7 เปรียบเทียบการเรียงข้อมูลแบบ Reversed กับ Random

ในการทดลองจะทำการทดลองกับข้อมูลที่เรียงลำดับจากน้อยไปหามาก (Reversed) และข้อมูลที่เรียงลำดับแบบสุ่ม (Random)

โดยทั่วไปข้อมูลที่เรียงลำดับจากน้อยไปหามากจะสามารถจัดเรียงลำดับได้เร็วกว่าชุดข้อมูลที่เรียงลำดับแบบสุ่ม

ตาราง 4.22 เวลาในการรันโดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวบัฟเฟอร์มีขนาด 1024 MB แรม 16GB แบบมีแคช โดยเปรียบเทียบการเรียงข้อมูลแบบ Reversed กับ Random

CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	Reversed	Random	Reversed	Random	Reversed	Random	Reversed	Random
Read(Sort)	28.87	48.72	25.80	33.33	38.68	51.79	47.50	72.72
Merge	2.98	6.67	3.40	4.85	0.82	1.47	5.74	10.29
Write	34.30	44.44	26.80	36.26	59.79	71.09	121.27	141.16
Total	66.16	99.84	56.01	74.45	99.30	124.36	174.52	224.18

จากผลการทดลองพบว่า การเรียงข้อมูลแบบ Random จะใช้เวลาในการอ่านและการผสานนานกว่าการเรียงข้อมูลแบบ Reversed ค่อนข้างมาก และ ใช้เวลาในขั้นตอนWrite นานกว่าเล็กน้อย โดยสามารถเทียบอัตราส่วนได้ดังนี้

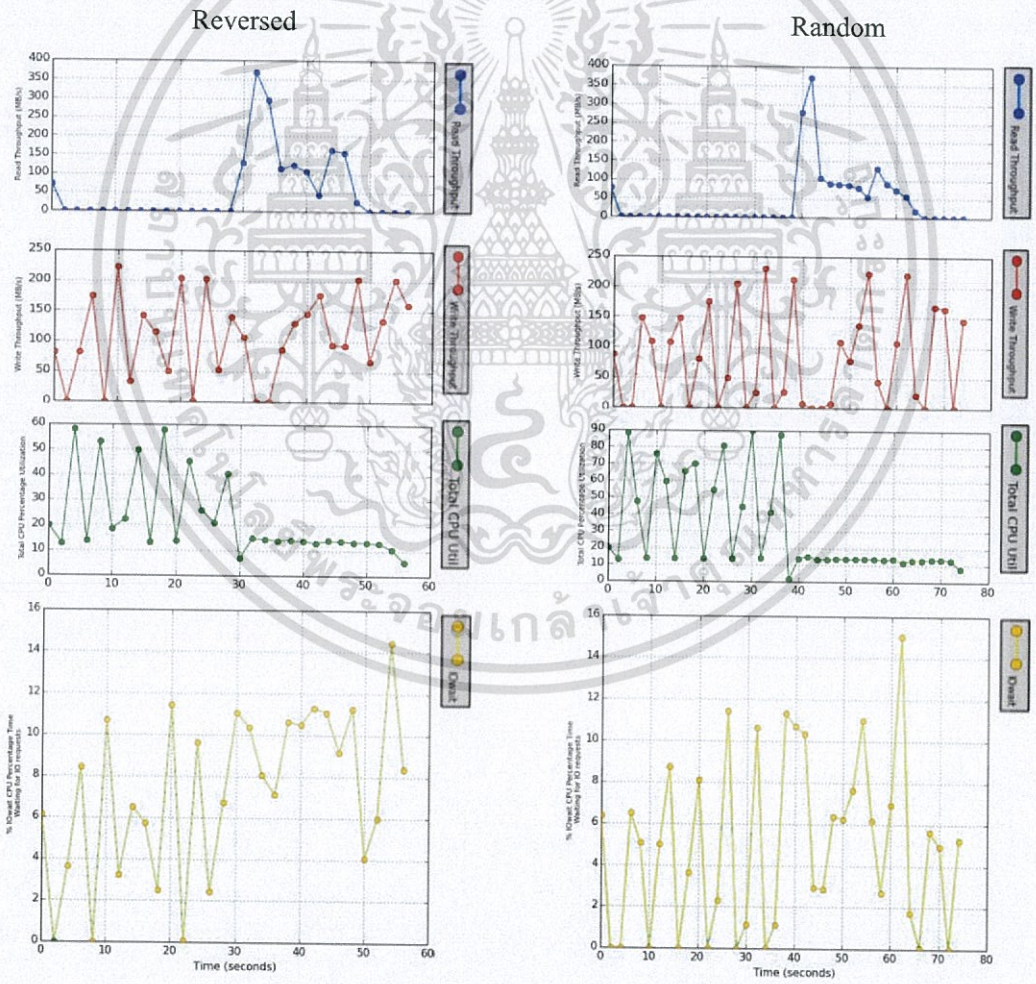
ตาราง 4.23 อัตราส่วนความเร็วโดยใช้ข้อมูลชนิด Uint32 ขนาด 800 ล้านตัว บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบ เรียงข้อมูลแบบ Random กับ Reversed

Speedup	Ratio Random:Reversed speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	1.69	1.29	1.34	1.53
Merge	2.24	1.43	1.79	1.79

ตาราง 4.23 อัตราส่วนความเร็วโดยใช้ข้อมูลชนิด Uint32 ขนาด 800 ล้านตัว บัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบ เรียงข้อมูลแบบ Random กับ Reversed (ต่อ)

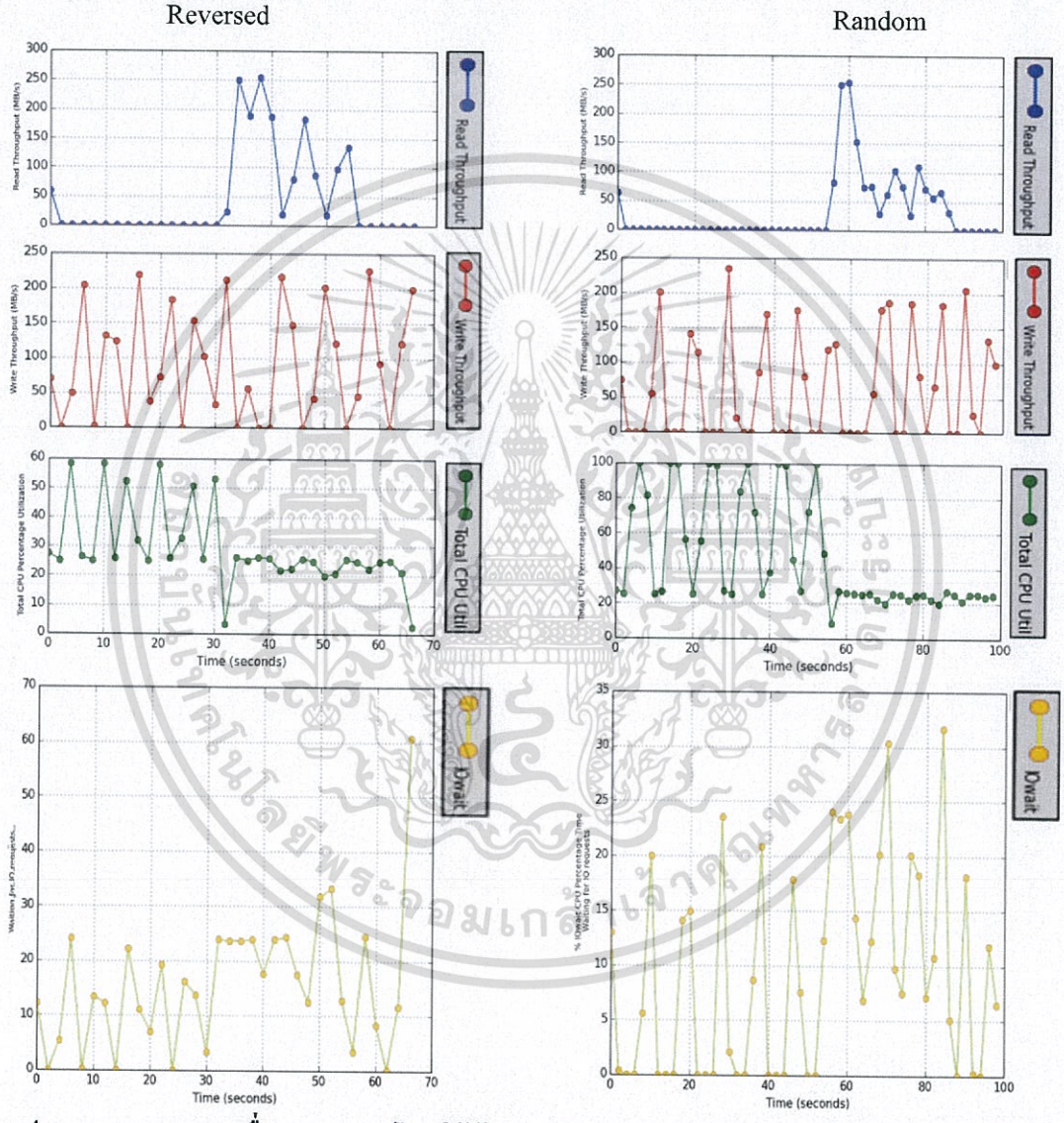
Speedup	Ratio Random:Reversed speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Write	1.30	1.35	1.19	1.16
Total	1.51	1.33	1.25	1.28

จากผลการเปรียบเทียบพบว่า การเรียงข้อมูลแบบ Random ใช้เวลาในขั้นตอน Write นานกว่าการเรียงข้อมูลแบบ Reversed ประมาณ 1.4 เท่า และใช้เวลาในขั้นตอน Merge นานกว่า ประมาณ 1.8 เท่า และใช้เวลาในขั้นตอน Write นานกว่า 1.25 เท่า



รูป 4.12 iostat ของเครื่อง i7-2600 โดย ใช้ข้อมูลชนิด Uint32 ขนาด 800 ล้านตัวบัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบการเรียงข้อมูลแบบ Reversed กับ Random

จากกราฟพบว่าในขั้นตอน Merge ข้อมูลแบบ Reversed ใช้ CPU utilization น้อยกว่าแบบ Random เพราะในการเรียงข้อมูลจะใช้ branch prediction ในการทำนายข้อมูลถัดไปซึ่งข้อมูลแบบ Reversed จะทำให้การทำนายข้อมูลถูกต้องมากว่าการเรียงข้อมูลแบบ Random ซึ่ง branch prediction นี้ส่งผลกับขั้นตอน Merge และขั้นตอน Write เช่นกันเพราะเป็นการเขียนเมื่อบัฟเฟอร์เต็ม และเมื่อบัฟเฟอร์หมดจะอ่านข้อมูลชุดถัดไปแล้วทำการผสานต่อ



รูป 4.13 iostat ของเครื่อง i3-2100 โดย ใช้ข้อมูลชนิด Uint32 ขนาด 800 ล้านตัวบัฟเฟอร์มีขนาด 1024 MB แรม 16GB และมีแคช โดยเปรียบเทียบการเรียงข้อมูลแบบ Reversed กับ Random

จากกราฟ พบว่าการใช้ CPU utilization ของ Random สูงกว่า แบบ Reversed เพราะการจัดเรียงข้อมูลจะมี branch prediction ในการทำนายข้อมูลถัดไปซึ่งการเรียงข้อมูลแบบ Reversed จะทำให้

branch prediction มีโอกาสทำนายถูก มากกว่า และทำให้ใช้เวลาน้อยกว่า และ ในขั้นตอนการผสาน เนื่องจากโครงสร้างของ tree ของข้อมูลแบบ Reversed จะทำให้ ข้อมูลใน branch ที่ถูกเรียงก่อนมีค่าน้อยกว่าข้อมูลใน branch ที่ถูกเรียงที่หลังแน่นอน จึงทำให้ branch prediction สามารถทำนายได้ว่าข้อมูลใน branch ที่อ่านอยู่จะน้อยกว่า branch อื่นเสมอ ซึ่งจะช่วยลดระยะเวลาการเปรียบเทียบข้อมูลระหว่างได้ ซึ่งขั้นตอน Write ก็จะมีการผสาน เมื่อบัฟเฟอร์หมดด้วยเช่นกันทำให้ข้อมูลแบบ Reversed เร็วกว่าข้อมูลแบบ Random ทั้ง 3 ขั้นตอน

4.8 เปรียบเทียบบัฟเฟอร์ขนาด 256 MB กับ 1024 MB

บัฟเฟอร์ที่กำหนดในการทดลองคือปริมาณบัฟเฟอร์ภายในโปรแกรม สำหรับการทำงานบนหน่วยความจำหลักในแต่ละช่วงของอัลกอริทึม ได้แก่ การจัดเรียงข้อมูลบนหน่วยความจำหลักก่อนที่จะเขียนลงบนหน่วยความจำภายนอกในขั้นตอนการจัดเรียง (Sort phase), ขนาดของ write buffer และโครงสร้างข้อมูลแบบ Tournament tree สำหรับติดตามบล็อคข้อมูลในขั้นตอนการผสาน (Merge phase)

ตาราง 4.24 เวลาในการรันโดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random แรม 16GB แบบมีแคช โดยเปรียบเทียบบัฟเฟอร์ขนาด 256 MB กับ 1024 MB

CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	256 MB	1024 MB	256 MB	1024 MB	256 MB	1024 MB	256 MB	1024 MB
Read (Sort)	49.19	48.72	35.32	33.33	51.07	51.79	77.59	72.72
Merge	1.57	6.67	1.12	4.85	1.47	1.47	2.37	10.29
Write	57.33	44.44	46.33	36.26	71.03	71.09	195.40	141.16
Total	108.09	99.83	82.77	74.44	123.56	124.35	275.36	224.17

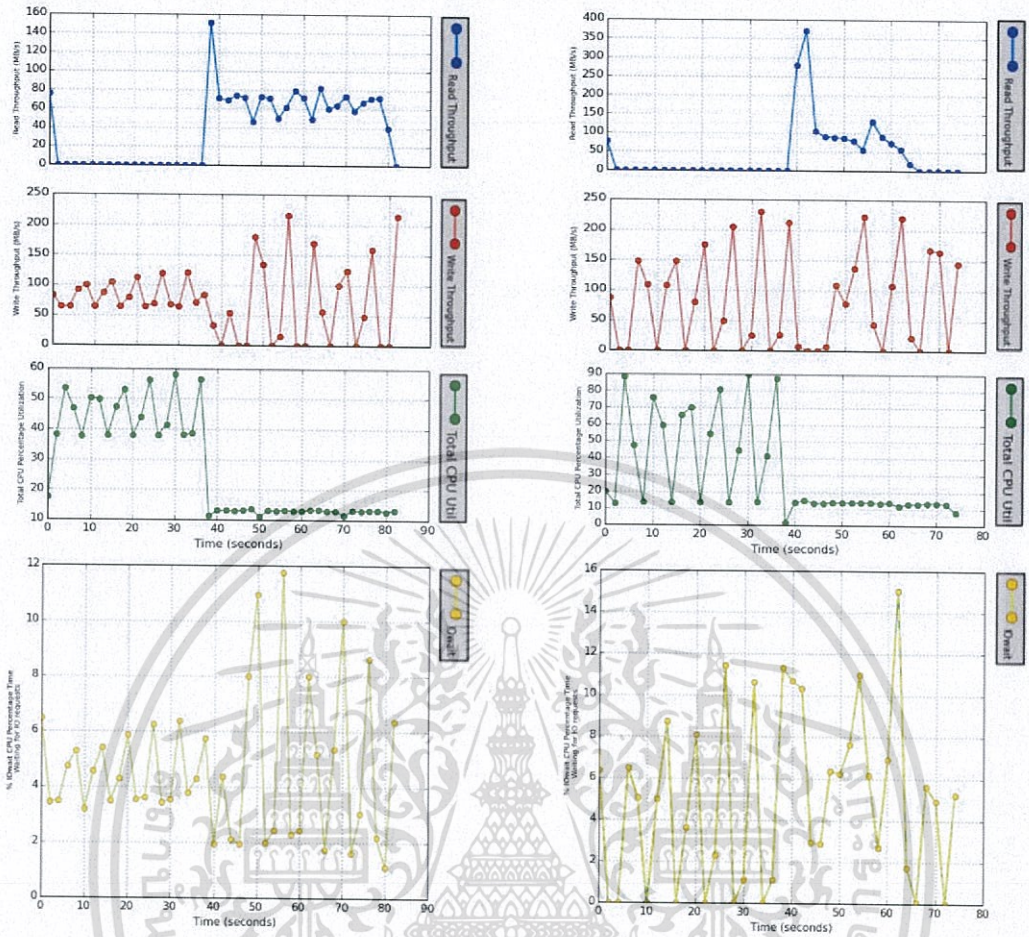
จากผลการทดลองพบในขั้นตอน Read ใช้เวลาไม่ต่างกันมากนัก แต่ในขั้นตอน Merge บัฟเฟอร์ขนาดเล็กใช้เวลาเร็วกว่าบัฟเฟอร์เร็วกว่าขนาดใหญ่เนื่องจากข้อมูลเต็มเร็วกว่าแต่เวลาที่ใช้ในขั้นตอน Write กลับช้ากว่าเนื่องจากการบัฟเฟอร์ขนาดเล็กทำให้ต้องผสาน ด้วยจำนวนรอบที่มากกว่าบัฟเฟอร์ขนาดใหญ่

ตาราง 4.25 อัตราส่วนความเร็วโดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random แรม 16GB และมีแคช โดยเปรียบเทียบบัฟเฟอร์ขนาด 256 MB กับ 1024 MB

Speedup	Ratio 256MB:1024MB speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	1.01	1.06	0.99	1.07
Merge	0.24	0.23	1.00	0.23
Write	1.29	1.28	1.00	1.38
Total	1.08	1.11	0.99	1.23

จากผลอัตราส่วนความเร็วเห็นได้ว่าในขั้นตอน Merge นั้นบัฟเฟอร์ 1024 MB ใช้เวลาผสานช้ากว่า บัฟเฟอร์ 256 MB อยู่ประมาณ 4 เท่าแต่ในขั้นตอน Write บัฟเฟอร์ 1024 MB ดีกว่า 256 MB อยู่ประมาณ 1.2 เท่า

256 MB 1024 MB



รูป4.14 iostat ของเครื่อง i7-2600 โดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random แรม 16GB และมีแคช โดยเปรียบเทียบ บัฟเฟอร์ขนาด 256 MB กับ 1024 MB

จากกราฟในขั้นตอน Read ข้อมูลพบว่า บัฟเฟอร์ขนาด 256 MB กราฟ Read Throughput , Write Throughput และ iowait จะขึ้นลงไม่มากเท่าขนาด 1024 MB เพราะบัฟเฟอร์ขนาดเล็กจะเต็มไวกว่าบัฟเฟอร์ 1024 MB สำหรับขั้นตอน Merge บัฟเฟอร์ 256 MB มีขนาดเล็กจึงสามารถเรียงข้อมูลได้ไวกว่าบัฟเฟอร์ 1024 MB แต่ในขั้นตอน Write พบว่า Read Throughput ของบัฟเฟอร์ 256 อยู่ประมาณ 40-80 MB/s แต่บัฟเฟอร์ขนาด 1024 MB มีอยู่ที่ประมาณ 100 MB/s จากกราฟ Write Throughput จะเห็นว่า ไม่ได้เขียนข้อมูลอยู่ตลอดเพราะต้องรอข้อมูลบัฟเฟอร์เต็มถึงจะได้เขียนทำให้บางช่วง Write Throughput เป็น 0 MB/s ซึ่ง บัฟเฟอร์ขนาด 256 MB จะต้องอ่านเขียนข้อมูลบ่อยกว่าทำให้เกิดช่วง Write Throughput เป็น 0 MB/s แต่ Read Throughput ก็ต่ำน้อยกว่า บัฟเฟอร์ 1024 MB/s เช่นกันจึงทำให้ บัฟเฟอร์ขนาด 256 MB ช้ากว่า บัฟเฟอร์ขนาด 1024 MB

ตาราง 4.26 ผล perf โดย ใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random แรม 16GB และมีแคช โดยเปรียบเทียบ บัฟเฟอร์ขนาด 256 MB กับ 1024 MB

CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Buffer size	256 MB	1024MB	256 MB	1024MB	256 MB	1024MB	256 MB	1024MB
cpu-cycles	5.46E+11	5.52E+11	6.27E+11	6.21E+11	4.73E+11	4.75E+11	2.20E+12	2.06E+12
instructions	4.80E+11	4.62E+11	5.07E+11	4.88E+11	4.79E+11	4.79E+11	2.25E+12	2.04E+12
cache-ref	2.70E+8	**3.21E+8**	4.01E+8	4.09E+8	2.02E+11	2.03E+11	7.25E+11	6.70E+11
cache-misses	1.47E+8	**1.90E+8**	1.88E+8	1.79E+8	**2.00E+8**	1.85E+8	3.34E+8	3.46E+8
LLC-loads	2.37E+8	2.56E+8	4.07E+8	4.83E+8	1.56E+9	1.61E+9	1.99E+9	2.12E+9
LLC-load-misses	-	-	-	-	1.28E+8	1.44E+8	2.16E+8	2.55E+8
LLC-stores	9.31E+7	**1.30E+8**	**1.59E+8**	1.27E+8	2.20E+9	2.21E+9	8.77E+8	1.05E+9
dTLB-loads	1.30E+11	1.23E+11	1.36E+11	1.29E+11	2.17E+11	2.17E+11	1.55E+12	1.42E+12
dTLB-load-misses	1.49E+8	**1.16E+9**	4.76E+8	5.02E+8	**9.52E+5**	2.22E+5	**7.99E+5**	2.28E+5
dTLB-store	5.97E+10	5.93E+10	6.65E+10	6.58E+10	-	-	-	-
dTLB-store-miss	3.57E+7	4.47E+7	5.38E+6	9.69E+6	-	-	-	-
iTLB-loads	4.67E+6	**7.29E+6**	7.98E+6	**1.52E+7**	2.02E+11	2.02E+11	7.27E+11	6.72E+11
iTLB-load-misses	2.08E+5	**1.77E+6**	2.66E+5	**6.45E+5**	1.93E+4	1.36E+4	**4.88E+4**	2.33E+4
branch-loads	1.07E+11	1.06E+11	1.09E+11	1.07E+11	1.07E+11	1.07E+11	3.61E+11	3.35E+11
branch-load-misses	1.03E+10	1.02E+10	1.05E+10	1.03E+10	1.11E+10	1.11E+10	1.08E+10	1.05E+10
page-faults	**14,893**	4,722	**81,246**	6,774	**14,332**	13,822	**78,942**	4,861
context-switches	20,809	21,069	21,180	23,210	4,355	4,355	56,171	55,892

ตาราง 4.26 ผล perf โดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Random แรม 16GB และมีแคช โดยเปรียบเทียบ บัฟเฟอร์ขนาด 256 MB กับ 1024 MB(ต่อ)

CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Buffer size	256 MB	1024MB	256 MB	1024MB	256 MB	1024MB	256 MB	1024MB
cpu-migrations	75	75	104	78	263	309	111	135

จากรายงานของ perf แสดงให้เห็นว่าบัฟเฟอร์ขนาด 256 MB มีจำนวน page-faults เกิดขึ้นมากกว่าบัฟเฟอร์ขนาด 1024 MB สูงสุดถึง 16 เท่า ในกรณีของเครื่อง FX-8320 และกรณีจำนวณ page-faults เปลี่ยนแปลงน้อยที่สุดคือกรณีของเครื่อง A6-3650 ซึ่งบัฟเฟอร์ขนาด 256 MB มีจำนวน page-faults มากกว่าเพียง 1.03 เท่า

ตาราง 4.27 เวลาในการรันโดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed แรม 16GB แบบมีแคช โดยเปรียบเทียบบัฟเฟอร์ขนาด 256 MB กับ 1024 MB

CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Time (Sec)	256 MB	1024 MB	256 MB	1024 MB	256 MB	1024 MB	256 MB	1024 MB
Read (Sort)	30.27	28.87	27.19	25.80	39.25	38.68	49.61	47.50
Merge	0.81	2.98	0.84	3.40	0.81	0.82	1.35	5.74
Write	37.36	34.30	30.46	26.80	62.71	59.79	163.91	121.27
Total	68.44	66.16	58.49	56.01	102.78	99.30	214.89	174.52

จากผลการทดลอง พบว่า บัฟเฟอร์มีขนาดเล็กจะช่วยให้เวลาในขั้นตอน Merge ดีขึ้นเพราะบัฟเฟอร์เต็มเร็วทำให้เขียนได้ไวกว่า แต่จะเขียนด้วยจำนวนรอบที่มากกว่าโดยผลของเวลาอ่านบัฟเฟอร์ 256 MB ช้ากว่า บัฟเฟอร์ 1024 MB อยู่ประมาณ 2 วินาที และ เวลาเขียนข้อมูล บัฟเฟอร์ 256 MB ช้ากว่า บัฟเฟอร์ 1024 MB อยู่ ประมาณ 3 วินาที

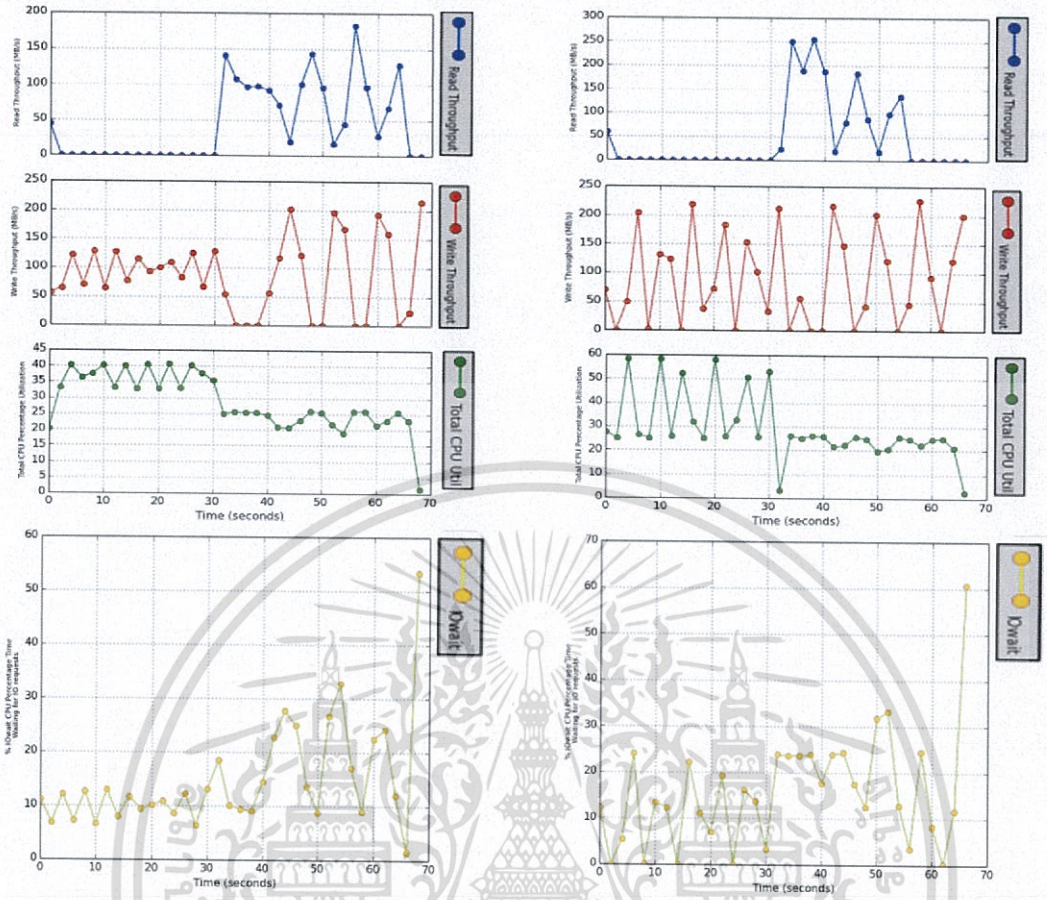
ตาราง 4.28 อัตราส่วนความเร็วโดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed แรม 16GB และมีแคช โดยเปรียบเทียบบัฟเฟอร์ขนาด 256 MB กับ 1024 MB

Speedup	Ratio 256MB:1024MB speedup			
	i3-2100	i7-2600	A6-3650	FX-8320
Read (Sort)	1.05	1.05	1.01	1.04
Merge	0.27	0.25	0.99	0.24
Write	1.09	1.14	1.05	1.35
Total	1.03	1.04	1.04	1.23

จากผลอัตราส่วนวามเร็วพบว่าการเรียงข้อมูลแบบ Reversed จะทำให้บัฟเฟอร์ 256 MB และ 1024 MB ใช้เวลาในขั้นตอน Write ใกล้เคียงกันมากขึ้นซึ่งทำให้อัตราส่วนการเขียนข้อมูลไม่แตกต่างกันมากเหมือนกับการเรียงข้อมูลแบบ Random

256 MB

1024 MB



รูป4.15 iostat ของเครื่อง i3-2100 โดย ใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed แรม 16GB และมีแคช โดยเปรียบเทียบ บัฟเฟอร์ขนาด 256 MB กับ 1024 MB

จากกราฟในขั้นตอน Read พบว่า CPU utilization ของบัฟเฟอร์ขนาด 1024 MB จะขึ้นลงระหว่าง 30-60% แต่ บัฟเฟอร์ขนาด 256 MB อยู่ที่ 32-40% โดยมีความสัมพันธ์กับ Read Throughput , Write Throughput รวมถึง iowait โดยลักษณะกราฟของ บัฟเฟอร์ขนาด 256 MB จะแกว่งขึ้นลงน้อยกว่า บัฟเฟอร์ 1024 MB

ขั้นตอน Merge เวลาในขั้นตอน Write ข้อมูลพบว่า กราฟ CPU utilization , Read Throughput ,Write Throughput และ iowait ที่ได้จากบัฟเฟอร์ขนาด 256 MB ไม่ต่างจากบัฟเฟอร์ขนาด 1024 MB มากนัก

ตาราง 4.29 ผล perf โดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed แรม 16GB และมีแคช โดยเปรียบเทียบ บัฟเฟอร์ขนาด 256 MB กับ 1024 MB

CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Buffer size	256 MB	1024MB	256 MB	1024MB	256 MB	1024MB	256 MB	1024MB
cpu-cycles	2.39E+11	2.31E+11	3.37E+11	3.36E+11	3.14E+11	3.10E+11	1.27E+12	1.13E+12
instructions	4.06E+11	3.89E+11	4.35E+11	4.16E+11	4.08E+11	4.09E+11	1.56E+12	1.38E+12
cache-ref	3.96E+08	**4.98E+08**	4.80E+08	**5.49E+08**	9.66E+10	9.36E+10	3.76E+11	3.33E+11
cache-misses	1.63E+08	2.07E+08	2.02E+08	2.37E+08	1.95E+08	1.96E+08	3.02E+08	3.48E+08
LLC-loads	4.84E+08	**6.14E+08**	5.92E+08	6.91E+08	4.22E+09	4.20E+09	**4.39E+09**	2.08E+09
LLC-load-misses	-	-	-	-	2.37E+08	2.28E+08	1.26E+08	1.42E+08
LLC-stores	1.27E+08	2.07E+08	1.83E+08	2.52E+08	4.25E+09	4.21E+09	8.27E+08	9.31E+08
dTLB-loads	1.16E+11	1.08E+11	1.23E+11	1.16E+11	1.68E+11	1.63E+11	9.28E+11	8.22E+11
dTLB-load-misses	2.72E+07	2.37E+07	3.46E+07	3.78E+07	2.36E+05	3.07E+05	**1.26E+06**	2.24E+05
dTLB-store	4.92E+10	5.04E+10	5.29E+10	5.31E+10	-	-	-	-
dTLB-store-miss	2.97E+06	2.79E+06	4.71E+06	2.36E+06	-	-	-	-
iTLB-loads	**8.80E+06**	5.99E+06	1.63E+06	**4.57E+06**	9.75E+10	9.21E+10	3.79E+11	3.33E+11
iTLB-load-misses	**8.06E+05**	1.66E+05	**4.12E+06**	2.87E+05	**1.35E+05**	5.66E+04	5.49E+04	3.17E+04
branch-loads	8.79E+10	8.69E+10	9.40E+10	9.22E+10	8.81E+10	8.87E+10	2.51E+11	2.25E+11
branch-load-misses	6.46E+07	5.41E+07	5.58E+07	5.39E+07	6.51E+08	2.75E+08	2.37E+08	2.16E+08
page-faults	3,120	4,725	**81,244**	10,356	14,331	14,332	**79,895**	5,369
context-switches	18,117	17,514	14,231	13,560	5,211	5,074	35,604	32,254

ตาราง 4.29 ผล perf โดยใช้ข้อมูลบน SSD ชนิด Uint32 ขนาด 800 ล้านตัวเรียงข้อมูลแบบ Reversed แรม 16GB และมีแคช โดยเปรียบเทียบ บัฟเฟอร์ขนาด 256 MB กับ 1024 MB(ต่อ)

CPU	i3-2100		i7-2600		A6-3650		FX-8320	
Buffer size	256 MB	1024MB	256 MB	1024MB	256 MB	1024MB	256 MB	1024MB
CPU-migrations	88	54	98	80	266	332	519	268

ในกรณีของ Reversed ที่ข้อมูลเรียงลำดับจากมากไปน้อยจะมีจำนวน page-faults ไปในทิศทางเดียวกันกับ Random กล่าวคือ มีบัฟเฟอร์ขนาด 256 MB จะมีจำนวน page-faults มากกว่าบัฟเฟอร์ขนาด 1024 MB ยกเว้นแต่ในกรณีของเครื่อง i3-2100 ที่ page-faults ของกรณีบัฟเฟอร์ 256 MB มีจำนวนน้อยกว่า

บทที่5
บทสรุป

5.1 สรุปผลการทดลอง

โครงงานนี้ใช้ STXXL ในการทำงานกับหน่วยความจำภายนอกโดยนำเสนอผลการทดลองจัดเรียงข้อมูลขนาดใหญ่เพื่อวัดประสิทธิภาพด้านต่างๆของฟังก์ชันการจัดเรียงข้อมูลด้วยเครื่องมือ iostat และ perf บนเครื่อง i3-2100, i7-2600, AMD-A6 และ FX-8320 ซึ่งทำงานอยู่บนระบบปฏิบัติการลินิกส์ 14.04 LTS สามารถเปรียบเทียบความเร็วเฉลี่ยของทุกการทดลองและทุกๆเครื่อง โดยเรียงลำดับผลการทดที่เพิ่มความเร็วได้มากที่สุดไปจนถึงน้อยที่สุดคือ SSD เร็วกว่า HDD อยู่ 121% ข้อมูลขนาด 800 ล้านตัวเร็วกว่าข้อมูลขนาด 1600 ล้านตัวอยู่ 115% ข้อมูลประเภท Uint32 เร็วกว่าข้อมูลประเภท Double อยู่ 68% ข้อมูลเรียงลำดับแบบ Reversed เร็วกว่าการเรียงข้อมูลแบบ Random อยู่ 15% บัฟเฟอร์ขนาด 1024 MB เร็วกว่า บัฟเฟอร์ขนาด 256 MB อยู่ 12% การมีแคชของระบบปฏิบัติการเร็วการไม่มีแคชของระบบปฏิบัติการอยู่ 7% และ แรม 16 GB เร็วกว่าแรม 8 GB อยู่ 6%

การใช้ SSD สามารถเพิ่มประสิทธิภาพในการอ่านข้อมูล,การผสานข้อมูลและการเขียนข้อมูลเมื่อเทียบกับ HDD เพราะอ่าน SSD สามารถเข้าถึงข้อมูลได้ไวกว่า การเพิ่มแรมจะมีผลก็ต่อเมื่อมีแคชของระบบปฏิบัติการทำให้ข้อมูลไปอยู่บนแรมและอ่านจากแรมแทนการอ่านจากหน่วยความจำรอง การเข้าถึงข้อมูลบนแรมจะเร็วกว่าหน่วยความจำรองจึงช่วยเพิ่มประสิทธิภาพในการอ่าน ขนาดข้อมูลและประเภทข้อมูล จะมีผลต่อประสิทธิภาพด้านการอ่านข้อมูลและการเขียนข้อมูล โดยถ้าขนาดข้อมูลมีจำนวนมาก หรือ ประเภทข้อมูลมีขนาดใหญ่จะต้องใช้เวลาในการอ่านและการเขียน นานกว่า ข้อมูลจำนวนน้อยกว่า หรือ ประเภทข้อมูลที่มีขนาดเล็กกว่า แต่จำไม่ส่งผลกับการผสานข้อมูล การจัดเรียงข้อมูลแบบ Reversed จะเพิ่มประสิทธิภาพในการอ่านข้อมูล , การผสานข้อมูล และการเขียนข้อมูล เมื่อเทียบกับการจัดเรียงข้อมูลแบบ Random เพราะมี Branch predicition ในการทำนายผลข้อมูล การใช้บัฟเฟอร์ขนาดใหญ่จะช่วยเพิ่มประสิทธิภาพในเขียนข้อมูลแม้บัฟเฟอร์ขนาดเล็กจะผสานข้อมูลได้เร็วกว่าแต่เมื่อมีปริมาณข้อมูลที่ต้องเขียนเท่ากันแล้วบัฟเฟอร์ขนาดเล็กมี page-faults สูงกว่า บัฟเฟอร์ขนาดใหญ่ค่อนข้างมากทำให้ใช้เวลานานกว่าบัฟเฟอร์ขนาดใหญ่

5.2 แนวทางในการพัฒนาต่อ

ในขั้นตอนการอ่านข้อมูลเราสามารถเปลี่ยนอัลกอลิทึมที่ใช้ในการจัดเรียงข้อมูลในหน่วยความจำหลักเป็นอัลกอลิทึมที่มีประสิทธิภาพสูงกว่าและจากแนวโน้มในการใช้งานในปัจจุบันที่หน่วยความจำหลักมีขนาดใหญ่ขึ้นอาจใช้ Massively Parallel Sort-Merge Joins แทนอัลกอลิทึมเดิมที่ใช้ในการทดลอง

บรรณานุกรม

R. Dementiev, P. Sanders: "Asynchronous Parallel Disk Sorting." 138-148. In 15th ACM Symposium on Parallelism in Algorithms and Architectures (June 7-9, 2003, San Diego, California, USA)

M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. PVLDB, 1064–1075, 2012.

STXXL. 2014. "STXXL: Standard Template Library for Extra Large Data Sets" [ออนไลน์]. Available:http://stxxl.sourceforge.net

Wikipedia. 2015. "Linux kernel profiling with perf" [ออนไลน์]. Available:https://perf.wiki.kernel.org/index.php/Tutorial.

Die.net. "perf-stat (1) - Linux man page" [ออนไลน์] Available: http://linux.die.net/man/1/perf-stat

Prof. Erik Demaine. 2012. "Advanced Data Structures" [ออนไลน์] Available: https://courses.csail.mit.edu/6.851/spring12/scribe/lec7.pdf

Wikipedia, 2015 "External sorting" [ออนไลน์] Available: https://en.wikipedia.org/wiki/External_sorting