การเรียงลำดับแบบขนานที่มีประสิทธิภาพสูง ด้วยวิธีการติดต่อสื่อสารแบบไดนามิค

HIGH PERFORMANCE PARALLEL MERCED SORT USING DYNAMIC COMMUNICATION

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาปรัชญาดุษฎีบัณฑิต
สาขาวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
พ.ศ. 2558
KMITL-2015-SC-D-002-043

การเรียงลำดับแบบขนานที่มีประสิทธิภาพสูง
ด้วยวิธีการติดต่อสื่อสารแบบไดนามิค

HIGH PERFORMANCE PARALLEL MERGED SORT
USING DYNAMIC COMMUNICATION

ธิปภร ธนกุลวรภาส

TIPRAPORN THANAKULWARAPAS

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาปรัชญาดุษฎีบัณฑิต

สาขาวิทยาการคอมพิวเตอร์

คณะวิทยาศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2558

KMITL-2015-SC-D-002-043

HIGH PERFORMANCE PARALLEL MERGED SORT USING DYNAMIC COMMUNICATION

TIPRAPORN THANAKULWARAPAS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

FACULTY OF SCIENCE

2015

KMITL-2015-SC-D-002-043

คณะวิทยาศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ใบรับรองวิทยานิพนธ์

หัวข้อวิทยานิพนธ์ "การเรียงลำดับแบบขนานที่มีประสิทธิภาพสูงด้วยวิธีการติดต่อสื่อสารแบบไดนามิค"
"High Performance Parallel Merged Sort Using Dynamic Communication"

ชื่อนักศึกษา นางสาวธิปภร ธนกุลวรภาส

รหัสประจำตัว 50067102

ปริญญา ปรัชญาดุษฎีบัณฑิต (วิทยาการคอมพิวเตอร์)

ภาควิชา วิทยาการคอมพิวเตอร์

อาจารย์ที่ปรึกษาวิทยานิพนธ์ รองศาสตราจารย์ ดร.จีรพร วีระพันธุ์

อาจารย์ที่ปรึกษาวิทยานิพนธ์ร่วม -

คณะกรรมการสอบวิทยานิพนธ์	ลายมือชื่อ
ผศ.ดร.นวลสวาท หิรัญสกลวงศ์ ประธานกรรมการ	
รศ.ดร.วีระ บุญจริง อาจารย์บัณฑิตประจำ (ในสาขาวิชาที่เกี่ยวข้อง)	
ผศ.ดร.ศรัณย์ อินทโกสุม อาจารย์บัณฑิตประจำ (ในสาขาวิชาที่เกี่ยวข้อง)	
ดร.เฉลิมศักดิ์ เลิศวงศ์เสถียร ผู้ทรงคุณวุฒิจากภายนอกสถาบันฯ	
รศ.ดร.จีรพร วีระพันธุ์ อาจารย์ที่ปรึกษาวิทยานิพนธ์	

วัน/ เดือน/ ปี ที่สอบ พุธที่ 15 พฤษภาคม พ.ศ.2556 เวลา 17.00-19.00 น.

สถานที่สอบ ห้อง 304 อาคารปฏิบัติการหลังใหม่

คณะวิทยาศาสตร์รับรองแล้ว

(รองศาสตราจารย์ ดร.ดุษณี ธนะบริพัฒน์)
คณบดีคณะวิทยาศาสตร์
วันที่ ๑๖ เดือน ๓๑ พ.ศ. ๒๕๕๖

หัวข้อวิทยานิพนธ์	การเรียงลำดับแบบขนานที่มีประสิทธิภาพสูงด้วยวิธีการติดต่อสื่อสารแบบไดนามิค
นักศึกษา	นางสาวธิปภร ธนกุลวรภาส
รหัสประจำตัว	50067102
ปริญญา	ปรัชญาดุษฎีบัณฑิต
สาขาวิชา	วิทยาการคอมพิวเตอร์
พ.ศ.	2558
อาจารย์ผู้ควบคุมวิทยานิพนธ์	รศ. ดร. จีรพร วีระพันธุ์

บทคัดย่อ

งานวิจัยนี้ได้นำเสนอกลยุทธิ์การปรับปรุงประสิทธิภาพของการเรียงลำดับแบบขนานด้วยวิธีไบโทนิคโดยใช้ค่ามิดพอยท์ ส่งผลให้การติดต่อสื่อสารแบบไดนามิคเกิดประโยชน์สูงสุด อีกทั้งช่วยเพิ่มประสิทธิภาพด้านเวลาบนระบบแบบขนานและแบบกระจาย ซึ่งเรียกวิธีการเรียงลำดับนี้ว่า “โอบีเอส” สำหรับวิธีการนี้มุ่งเน้นที่การได้มาซึ่งคีย์ที่ดีในเวลาที่เหมาะสมก่อให้เกิดการติดต่อสื่อสารที่มีประสิทธิผล จนได้มาซึ่งจำนวนรอบการทำงานน้อยที่สุด ในแต่ละรอบทำงานเต็มศักยภาพ และมุ่งสู่ผลลัพธ์อย่างรวดเร็ว ในงานวิจัยที่ผ่านมานั้นมุ่งเน้นการเพิ่มประสิทธิภาพด้านการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผล โดยงานวิจัยเหล่านั้นยังคงใช้รูปแบบการติดต่อสื่อสารแบบคงที่ ส่งผลให้จำนวนรอบการทำงานเท่ากับ $(\log_2 P (\log_2 P +1)/2)$ เสมอ เมื่อ P คือ จำนวนหน่วยประมวลผล โดยการเรียงลำดับแบบโอบีเอส จะมีรอบการทำงานที่ดีที่สุดเพียง 1 รอบ หรือ 2 หรือ 3 หรือ,…, หรือ $\log_2 P$ (สำหรับรอบการทำงานที่ช้าที่สุด) โดยแสดงการเปรียบเทียบประสิทธิภาพจากวิธีที่เสนอและวิธีเดิมด้วยการพัฒนาโปรแกรมบนระบบคอมพิวเตอร์แบบมัลติคอร์ พบว่าวิธีที่ได้นำเสนอในงานวิจัยนี้ให้ผลดีกว่าวิธีการอื่นๆ ที่มีอยู่ในปัจจุบัน อย่างน้อย 35-40% เมื่อเทียบกับการเรียงลำดับแบบดีซีอีเอส และ 51-54% เมื่อเทียบกับการเรียงลำดับแบบแอลบีเอ็ม เมื่อ N = 10-100 ล้านชุดข้อมูล และ P = 8 หน่วยประมวลผล

คำสำคัญ : การเรียงลำดับแบบไบโทนิคที่มีประสิทธิภาพสูง, การติดต่อสื่อสารแบบไดนามิคที่ใช้ค่ามิดพอยท์, การเรียงลำดับแบบขนานด้วยค่ามิดพอยท์

Thesis Title	High Performance Parallel Merged Sort using Dynamic Communication
Student	Tipraporn Thanakulwarapas
Student ID	50067102
Degree	Doctor of philosophy
Program	Computer Science
Year	2015
Thesis Advisor	Assoc. Prof. Dr. Jeeraporn Werapun

ABSTRACT

This thesis proposes an optimized *Bitonic* sorting (*OBS*) strategy with *midpoint*-based *dynamic* communication. Our *OBS* strategy uses the *midpoint*-weight parallel list ranking to improve complexity and reduce time of sorting on parallel and distributed systems. Applying a *better* key in the *PE*-list ranking can find the right place of (P_i, P_j) and improve communication time significantly (i.e., fewer iterations, better synchronization in each iteration, and faster convergence to the sorting result), while most of coarse-grain parallel sorting ($P < N$) approaches improve only a large amount of data exchange (N/P) in each of *static* ($log_2P(log_2P+1)/2$) iterations. Theoretically, the *OBS* method can reduce fixed ($log_2P(log_2P+1)/2$) iterations to *1, 2, 3, ...* , or log_2P iterations, which are improved over those ($\leq log_2P(log_2P+1)/2$ iterations) of the *dynamic DCES* (*D*ynamic *C*ommunication *E*fficient *S*ort) method. In performance evaluation, sorting was accomplished on a *multicore* machine. Experimental results showed that the optimized *OBS* outperforms those of the *dynamic DCES* about 35% - 40% and those of the *static LBM* (*L*oad-*B*alance *M*erge Sort) about 51% - 54% (for $N = 10$ to *100* million elements on an *8-multicore* computer).

Keywords : Optimized Bitonic sorting; midpoint-based dynamic communication; midpoint-weight parallel list-ranking.

กิตติกรรมประกาศ

วิทยานิพนธ์นี้มิอาจสำเร็จลุล่วงได้ด้วยดี หากมิได้รับคำแนะนำ คำชี้แจง ความรู้ และความเอาใจใส่จาก รศ. ดร. จีรพร วีระพันธุ์ ผู้เป็นอาจารย์ที่ปรึกษา ซึ่งท่านได้สละเวลาให้กับข้าพเจ้าอย่างเต็มที่ จึงใคร่กราบขอบพระคุณเป็นอย่างสูง

ขอขอบพระคุณ รศ. ดร. วีระ บุญจริง ผศ. ดร. ศรัณย์ อินทโกสุม ผศ. ดร. นวลสวาท หิรัญ-สกลวงศ์ และดร. เฉลิมศักดิ์ เลิศวงศ์เสถียร คณะกรรมการสอบวิทยานิพนธ์ที่กรุณาให้คำแนะนำตลอดจนคำชี้แนะในด้านต่างๆ ส่งเสริมให้ผู้วิจัยมีวิสัยทัศน์กว้างไกลยิ่งขึ้น

ขอขอบพระคุณ ผศ.ดร.ดวงกมล กลีสัน และคุณ Paul Gleeson สำหรับความช่วยเหลือด้านภาษาในผลงานตีพิมพ์ระดับนานาชาติ

ขอขอบพระคุณคณะทำงานของวารสาร “Journal of Parallel and Distributed Computing” สำหรับคำวิจารณ์ที่สร้างสรรค์และคำแนะนำที่มีประโยชน์อย่างสูงต่อผลงานวิจัยฉบับนี้

ขอขอบคุณเพื่อนๆ ทุกคน สำหรับกำลังใจ และความห่วงใยที่มีให้ รวมทั้งความช่วยเหลือในเรื่องต่างๆ ทำให้ผู้วิจัยมีแรงใจมุมานะพยายามทำวิทยานิพนธ์นี้ให้สำเร็จลุล่วงได้

สำหรับคุณงามความดีและประโยชน์อันใดที่เกิดขึ้นจากวิทยานิพนธ์ฉบับนี้ ข้าพเจ้าขอมอบให้แด่บิดา มารดา ผู้ซึ่งให้โอกาสทางการศึกษาและสนับสนุนทั้งแรงกาย แรงใจ เป็นผู้คอยห่วงใยเอาใจใส่เสมอมา ทำให้ผู้วิจัยได้รับโอกาสที่จะแสวงหาความรู้และพัฒนาตนเอง จนเกิดวิทยานิพนธ์ฉบับนี้ขึ้นมาได้ ผู้วิจัยสำนึกถึงพระคุณในข้อนี้เป็นอย่างสูง

ธิปภร ธนกุลวรภาส

พ.ค. 2558

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

ตารางที่ **หน้า**

สารบัญตาราง (ต่อ)

ตารางที่ **หน้า**

สารบัญรูปภาพ

รูปที่ **หน้า**

สารบัญรูปภาพ (ต่อ)

รูปที่ **หน้า**

สารบัญรูปภาพ (ต่อ)

รูปที่ **หน้า**

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

ข้อมูลข่าวสารในปัจจุบันมีการขยายตัวเพิ่มขึ้นอย่างรวดเร็ว ดังนั้นจะเห็นว่าในแต่ละวันมีเนื้อหาของข้อมูลข่าวสารเพิ่มขึ้นจำนวนมหาศาลจากอัตราส่วนดังกล่าว ทำให้การเรียงลำดับ (Sorting) เข้ามามีบทบาทสำคัญและจำเป็น เพื่ออำนวยความสะดวกในการเข้าถึงข้อมูลได้อย่างรวดเร็วและทันเวลา การเรียงลำดับจึงเป็นงานหลักที่สำคัญสำหรับงานด้านวิศวกรรมศาสตร์และด้านวิทยาศาตร์ จนมีคำกล่าวที่ว่า “การทำงานของคอมพิวเตอร์มากกว่า 25% ต้องเสียไปกับงานด้านการเรียงลำดับ” [14] ดังนั้นการศึกษาเทคนิคและวิธีการเรียงลำดับที่เหมาะสมจึงเป็นเรื่องที่น่าสนใจและจำเป็น เพื่อ

1) คิดค้นวิธีการเรียงลำดับแบบใหม่ๆ ที่สามารถประมวลผลได้เร็วขึ้น
2) สามารถประยุกต์ใช้ในการเลือกวิธีการเรียงลำดับที่เหมาะสมกับความต้องการของงาน
3) สามารถประยุกต์ใช้เครื่องคอมพิวเตอร์ในปัจจุบันให้เหมาะสมกับวิธีการเรียงลำดับ

การเรียงลำดับ (Sorting) สามารถแบ่งได้เป็น 2 ประเภทใหญ่ๆ ดังนี้

การเรียงลำดับแบบอนุกรม (Sequential Sort) โดยทั่วไปแล้วการเรียงลำดับข้อมูลเป็นการเรียงลำดับแบบอนุกรม ซึ่งเรียงตามลำดับข้อมูลจากน้อยไปหามาก (Ascending-order Sequence) หรือเรียงลำดับข้อมูลจากมากไปหาน้อย (Descending-order Sequence) ขั้นตอนวิธี (Algorithm) สำหรับการเรียงลำดับข้อมูลแบบอนุกรมแบบต่างๆ นี้ มีผู้คิดขึ้นมากมายด้วยความซับซ้อนด้านเวลา (Time Complexity) เท่ากับ $O(N \log_2 N)$ เมื่อ N คือขนาดของข้อมูลที่ใช้ในการเรียงลำดับ อาทิ การเรียงลำดับแบบเร็ว (Quick Sort) [29] การเรียงลำดับแบบรวมผสานกัน (Merge Sort) [26][29] การเรียงลำดับแบบฮีป (Heap Sort)[29] และความซับซ้อนด้านเวลาเท่ากับ O(N) เช่น การเรียงลำดับแบบเอสซีเอสเรดิกซ์ (SCS-Radix Sort) [13] โดยแต่ละวิธีต่างมีข้อดีและข้อจำกัดที่แตกต่างกันขึ้นอยู่กับลักษณะและปริมาณข้อมูลที่ต้องการเรียงลำดับ รวมถึงประสิทธิภาพของเครื่องคอมพิวเตอร์ที่ใช้ในการเรียงลำดับข้อมูลอีกด้วย

การเรียงลำดับแบบขนาน (Parallel Sort) [3] การเรียงลำดับแบบขนานถูกเสนอขึ้นสำหรับการเรียงลำดับข้อมูลขนาดใหญ่ที่เร็วขึ้น ทดแทนการเรียงลำดับข้อมูลแบบอนุกรม (Sequential Sort) ที่ใช้เพียงหนึ่งหน่วยประมวลผล ซึ่งใช้เวลาในการเรียงลำดับข้อมูลเป็นระยะเวลานาน จากปัญหาดังกล่าวจึงใช้วิธีการเรียงลำดับแบบขนานที่ใช้หลายๆ หน่วยประมวลผลในการทำงาน และการแบ่งข้อมูล (Data Partitioning) ในจำนวนที่เหมาะสมให้แต่ละหน่วยประมวลผล ซึ่งช่วยให้การเรียงลำดับข้อมูลขนาดใหญ่ทำได้ภายในระยะเวลาอันสั้น

การเรียงลำดับแบบขนาน (Parallel Sort) ที่นิยมใช้ในปัจจุบัน เช่น การเรียงลำดับแบบขนานด้วยวิธีไบโทนิค (Bitonic Sort) [2][22][27] การเรียงลำดับแบบขนานด้วยวิธีรวมผสานกัน (Parallel Merge Sort) [19][23] การเรียงลำดับแบบขนานด้วยวิธีคู่คี่ (Odd-Even Sort) [22][23] โดยการเรียงลำดับดังกล่าวนี้มีความซับซ้อนด้านเวลา (Time Complexity) เท่ากับ $O(\log_2 N)^2$ เมื่อจำนวนหน่วยประมวลผลเท่ากับจำนวนข้อมูลที่ใช้ในการเรียงลำดับ (P=N) เป็นต้น

ในวิทยานิพนธ์ฉบับนี้ให้ความสนใจเรื่องการเรียงลำดับแบบขนานในกรณีที่ $P < N$ โดยมุ่งเน้นเรื่องการเพิ่มประสิทธิภาพของการเรียงลำดับแบบขนานใน 4 ประเด็นสำคัญ กล่าวคือ

1) ลดจำนวนรอบในการเรียงลำดับแบบขนานเพื่อประสิทธิภาพในการทำงานสูงสุด
2) ลดจำนวนข้อมูลที่ใช้ในการแลกเปลี่ยนระหว่างหน่วยประมวลผลลงอย่างน้อย 50% เมื่อเทียบกับงานวิจัยที่มีอยู่ในปัจจุบัน
3) ลดเวลาในการรวมผสานข้อมูลที่ได้จากการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลแล้ว
4) หลีกเลี่ยงการติดต่อสื่อสารและการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลที่ไม่จำเป็น

จากจุดมุ่งหมายของงานวิจัยที่ได้ตั้งไว้ ผู้วิจัยทำการเปรียบเทียบประสิทธิภาพด้วยการประเมินผลจากเวลาที่ใช้ในการเรียงลำดับข้อมูล (Response Time) อัตราการเพิ่มขึ้นของความเร็ว (Speedup) และประสิทธิภาพ (Efficiency) ซึ่งจะพัฒนาโปรแกรมการเรียงลำดับแบบขนานด้วยภาษาซี (C Language) และมาตรฐานภาษาเอ็มพีไอ (MPI Standard) [21] บนระบบคอมพิวเตอร์แบบมัลติคอร์ (Multi-Core Computer) เพื่อใช้ในการทดลองและหาผลสรุปของงานวิจัย

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา

วิทยานิพนธ์นี้มีวัตถุประสงค์เพื่อศึกษาวิธีการเรียงลำดับแบบขนานในกรณีที่ $P < N$ ที่มีอยู่ในปัจจุบัน โดยนำเสนอวิธีการใหม่เพื่อเพิ่มประสิทธิภาพของการเรียงลำดับแบบขนานในกรณีดังกล่าว ($P < N$) คือ

1) เพิ่มประสิทธิภาพด้านการติดต่อสื่อสารระหว่างหน่วยประมวลผล จากงานวิจัยเดิมที่มีในปัจจุบันการติดต่อสื่อสารแบบสแตทิค (Static Communication) เป็นแบบการติดต่อสื่อสารแบบไดนามิค (Dynamic Communication) ที่ได้นำเสนอขึ้นใหม่ในวิทยานิพนธ์นี้
2) เพิ่มประสิทธิภาพด้านการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผล โดยลดจำนวนข้อมูลในการแลกเปลี่ยนลงอย่างน้อย 50% เมื่อเทียบกับงานวิจัยเดิมที่มีอยู่ในปัจจุบัน (3 แบบ คือ แบบคงที่ (Hold Pattern) แบบสลับที่ (Swap Pattern) และแบบ

เฉพาะส่วน (Partial Pattern)) ด้วยการวิเคราะห์รูปแบบของข้อมูลที่ใช้แลกเปลี่ยนระหว่างหน่วยประมวลผลที่มีประสิทธิภาพมากขึ้นโดยเฉพาะสองแบบหลัง (Efficient Data Exchange) แบ่งเป็น 4 แบบคือ แบบคงที่ (Hold Pattern), แบบสลับที่เป็นคงที่ (Swap-to-Hold Pattern), แบบบางส่วน-1 (Partial-1 Pattern) และแบบบางส่วน-2 (Partial-2 Pattern)

3) เพิ่มประสิทธิภาพด้านการรวมข้อมูลหลังจากแลกเปลี่ยนแล้ว ด้วยวิธีการรวมข้อมูลที่มีประสิทธิภาพ (Efficient Data Merging) โดยแบ่งเป็น 2 แบบคือ การรวมข้อมูลแบบบล็อก (Block Merging) และการรวมข้อมูลแบบหนึ่งต่อหนึ่ง (1-1 Merging)

4) หลีกเลี่ยงการติดต่อสื่อสารและการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลที่ไม่จำเป็น เช่น รูปแบบสลับที่ (Swap Pattern) ที่นำเสนอในงานวิจัยที่ผ่านๆ มา ผู้วิจัยสามารถเปลี่ยนรูปแบบสลับที่เป็นคงที่ (Swap-to-Hold Pattern) ซึ่งเป็นรูปแบบที่ดีที่สุด ภายใต้การติดต่อสื่อสารแบบไดนามิค

1.3 สมมติฐานของการศึกษา

ในงานวิจัยนี้ให้ความสนใจเรื่องการเพิ่มประสิทธิภาพสูงสุดในการติดต่อสื่อสารในกรณีที่ $P < N$ โดยมุ่งเน้นการเพิ่มประสิทธิภาพ 4 ด้านหลักๆ คือ

1) เพิ่มประสิทธิภาพในการติดต่อสื่อสารระหว่างหน่วยประมวลผล จากแบบสแตทิค (Static Communication) เป็นแบบไดนามิค (Dynamic Communication)

2) เพิ่มประสิทธิภาพในการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผล ด้วยวิธีการวิเคราะห์รูปแบบของข้อมูลที่ใช้แลกเปลี่ยน (Efficient Data Exchanging)

3) เพิ่มประสิทธิภาพการรวมข้อมูลหลังจากแลกเปลี่ยนแล้ว ด้วยวิธีการรวมข้อมูลอย่างมีประสิทธิภาพ (Efficient Data Merging)

4) หลีกเลี่ยงการติดต่อสื่อสารและการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลที่ไม่จำเป็นภายใต้การติดต่อสื่อสารแบบไดนามิค

ซึ่งวิธีการต่างๆ ที่ได้กล่าวมาช่วยสนับสนุนให้การเรียงลำดับแบบขนานที่ได้นำเสนอมีศักยภาพสูงสุดในการเรียงลำดับข้อมูล

1.4 ขอบเขตของการวิจัย

วิทยานิพนธ์นี้เป็นการศึกษาและนำเสนอวิธีการเรียงลำดับแบบขนาน โดยมีขอบเขตของการวิจัยในวิทยานิพนธ์ ซึ่งแบ่งเป็น 2 ส่วนดังนี้

<u>ส่วนที่ 1</u> ด้านทฤษฎี

ทำการศึกษาการเรียงลำดับแบบขนานในกรณีที่ P < N ที่มีอยู่ในปัจจุบัน แล้วนำวิธีการดังกล่าวมาปรับปรุงเพื่อเพิ่มประสิทธิภาพในการทำงานดังนี้

1) เพิ่มประสิทธิภาพในการติดต่อระหว่างหน่วยประมวลผล
2) เพิ่มประสิทธิภาพในการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผล
3) เพิ่มประสิทธิภาพในการรวมข้อมูลหลังจากแลกเปลี่ยนแล้ว
4) หลีกเลี่ยงการติดต่อสื่อสารและการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลที่ไม่จำเป็น
5) ข้อมูลที่ใช้ในการเรียงลำดับข้อมูลเป็นข้อมูลชนิดตัวเลข (Number) ซึ่งมีขนาด 10^8 โดยมีค่าอยู่ระหว่าง 0-100,000,000

<u>ส่วนที่ 2</u> ด้านการประยุกต์ใช้งาน (การทดลองและผลการทดลอง)

นำทฤษฎีที่ได้ทำการศึกษาในส่วนแรกมาพัฒนาเพื่อทำการทดลองโดยใช้มาตรฐานภาษาซี (C Language) และมาตรฐานภาษาเอ็มพีไอ (MPI Standard) ที่สนับสนุนการโปรแกรมแบบขนานบนระบบคอมพิวเตอร์แบบมัลติคอร์ (Multi-Core Computer) และผลลัพธ์ที่ได้จากการทดลองใช้เป็นแนวทางในการวิเคราะห์เพื่อหาผลสรุป

1.5 ขั้นตอนการศึกษาและการดำเนินงานวิจัย

วิทยานิพนธ์นี้มีขั้นตอนการศึกษาและการดำเนินงานวิจัย ดังนี้

1) ศึกษาขั้นตอนวิธี (Algorithm) การเรียงลำดับแบบขนานในกรณีที่ P < N ที่มีอยู่ในปัจจุบัน
2) ศึกษางานวิจัยที่เกี่ยวข้องกับการเรียงลำดับแบบขนาน ทั้งกรณีที่ P < N และ P = N
3) ทำการตั้งสมมติฐาน โดยคาดว่าการเรียงลำดับแบบขนานที่นำเสนอในวิทยานิพนธ์นี้สามารถเพิ่มประสิทธิภาพในด้านต่างๆ ดังที่ได้ตั้งสมมติฐานไว้ ส่งผลให้วิธีการที่นำเสนอมีประสิทธิภาพสูงสุดสำหรับการเรียงลำดับแบบขนานในกรณีที่ P < N
4) นำเสนอวิธีการเรียงลำดับแบบขนานที่มีประสิทธิภาพสูง ดังที่ได้ตั้งสมมติฐานไว้ในข้อ (3)
5) พัฒนาโปรแกรมการเรียงลำดับแบบขนานในกรณีที่ P < N ที่มีอยู่ในปัจจุบันและแบบที่นำเสนอในวิทยานิพนธ์โดยใช้มาตรฐานภาษาซี (C Language) โดยทำการทดลองบนระบบคอมพิวเตอร์แบบมัลติคอร์ที่ใช้ระบบปฏิบัติการวินโดว์ 7 (Windows 7) โดยใช้มาตรฐานภาษาเอ็มพีไอ (MPI Standard) ที่สนับสนุนการโปรแกรมแบบขนาน

6) วิเคราะห์ผลการทดลองโดยการเปรียบเทียบประสิทธิภาพของการเรียงลำดับแบบขนานที่ได้นำเสนอในวิทยานิพนธ์นี้ โดยประเมินผลได้จากเวลาที่ใช้ในการเรียงลำดับข้อมูล (Response Time) อัตราการเพิ่มขึ้นของความเร็ว (Speedup) และประสิทธิภาพ (Efficiency)
7) สรุปผลการทดลอง พร้อมเสนอแนวทางการพัฒนางานวิจัย
8) เขียนวิทยานิพนธ์

1.6 ประโยชน์ที่คาดว่าจะได้รับ

ประโยชน์ที่คาดว่าจะได้รับจากวิทยานิพนธ์นี้คือ การเรียงลำดับแบบขนานในกรณีที่ $P < N$ ที่มีประสิทธิภาพสูงด้วยวิธีการติดต่อสื่อสารแบบไดนามิคที่ทำการทดลองบนระบบคอมพิวเตอร์แบบมัลติคอร์ ดังนี้

1) นำวิธีการเรียงลำดับแบบขนานที่มีประสิทธิภาพสูงไปประยุกต์ใช้ในงานที่ต้องการความเร็วในการเรียงลำดับข้อมูลขนาดใหญ่กรณีที่ $P < N$
2) นำการติดต่อสื่อสารแบบไดนามิคไปใช้เป็นแนวทางสำหรับการประมวลผลแบบขนานที่มีการติดต่อสื่อสารระหว่างหน่วยประมวลผลกรณีที่ $P < N$
3) นำมาใช้เป็นแนวทางในการพัฒนาโปรแกรมแบบขนานบนระบบคอมพิวเตอร์แบบมัลติคอร์และระบบคอมพิวเตอร์แบบขนาน ในกรณีทั่วไปได้อย่างมีประสิทธิภาพ

บทที่ 2
ทฤษฎีและงานวิจัยที่เกี่ยวข้อง

การใช้งานเครื่องคอมพิวเตอร์ในปัจจุบันนี้เป็นเรื่องง่ายและใช้กันอย่างแพร่หลาย ดังนั้น การผลิตคอมพิวเตอร์เพื่อจำหน่ายต้องคำนึงถึงปัจจัยต่างๆ เช่น วัตถุประสงค์การใช้งาน ปริมาณการผลิต และโครงสร้างที่เหมาะสม อาทิ งานวิจัยเพื่อควบคุมเตาปฏิกรณ์นิวเคลียร์ งานคำนวณทางสถิติของตลาดหุ้น งานวิเคราะห์ภาพถ่ายทางการแพทย์ งานผลิตสื่อสิ่งพิมพ์ หรืองานฝึกอบรมสำหรับเด็ก เป็นต้น ดังนั้นการผลิตคอมพิวเตอร์ที่มีขนาดและความสามารถรวมไปถึงราคาที่เหมาะสมกับการใช้งานด้านต่างๆ จึงเป็นเรื่องจำเป็น การแบ่งประเภทของเครื่องคอมพิวเตอร์เพื่อประยุกต์ใช้ในการประมวลผลตั้งแต่อดีตจนถึงปัจจุบัน สามารถแบ่งได้ดังนี้

1) แบ่งตามลักษณะการประมวลผล กล่าวคือ การแบ่งตามสัญญาณข้อมูลที่ใช้ในการประมวลผล จำแนกได้เป็น 3 ประเภท คือ
   ก. คอมพิวเตอร์แบบแอนะล็อก (Analog Computer) เครื่องประมวลผลข้อมูลที่แสดงออกในลักษณะของสัญญาณแอนะล็อก (Analog Signal) เช่น เครื่องวัดความเร็วของรถยนต์ในลักษณะเข็มชี้ หรือเครื่องตรวจคลื่นหัวใจที่แสดงผลเป็นรูปกราฟ เป็นต้น
   ข. คอมพิวเตอร์แบบดิจิทัล (Digital Computer) เครื่องประมวลผลข้อมูลที่แสดงออกในลักษณะของสัญญาณไฟฟ้า (Digital Signal) เช่น เครื่องวัดความดันโลหิตแบบดิจิทัล เป็นต้น
   ค. คอมพิวเตอร์แบบผสม (Hybrid Computer) เครื่องประมวลผลข้อมูลที่อาศัยเทคนิคการทำงานแบบผสมผสาน ระหว่างคอมพิวเตอร์แบบแอนะล็อก (Analog Computer) และคอมพิวเตอร์แบบดิจิทัล (Digital Computer) ส่วนใหญ่ใช้ในงานที่มีลักษณะพิเศษ โดยเฉพาะงานด้านวิทยาศาสตร์ เช่น เครื่องคอมพิวเตอร์ในยานอวกาศ เป็นต้น
2) แบ่งตามวัตถุประสงค์ของการใช้งาน กล่าวคือ การแบ่งตามความต้องการใช้งานเฉพาะด้านที่มีคุณลักษณะพิเศษ จำแนกได้เป็น 2 ประเภท คือ
   ก. เครื่องคอมพิวเตอร์เพื่องานเฉพาะด้าน (Special Purpose Computer) กล่าวคือ เครื่องประมวลผลข้อมูลที่ออกแบบเครื่องและระบบควบคุม ให้ทำงานอย่างใดอย่างหนึ่งเป็นการเฉพาะ (Inflexible) เช่น เครื่องคอมพิวเตอร์ควบคุมระบบอัตโนมัติในรถยนต์ เป็นต้น
   ข. เครื่องคอมพิวเตอร์เพื่องานอเนกประสงค์ (General Purpose Computer) กล่าวคือ เครื่องประมวลผลข้อมูลที่มีความยืดหยุ่นในการทำงาน (Flexible) ออกแบบให้สามารถ

ประยุกต์ใช้งานได้หลากหลายประเภท เช่น ในขณะหนึ่งเราอาจใช้เครื่องนี้ในงานประมวลผลระบบเงินเดือน และในเวลาต่อมาสามารถใช้ดูหนัง ฟังเพลง เป็นต้น

3) แบ่งตามขนาดและความสามารถของเครื่องคอมพิวเตอร์ที่มีหน่วยประมวลผลเดียว กล่าวคือ การแบ่งตามสมรรถนะและหน่วยความจำของเครื่องคอมพิวเตอร์เป็นหลัก จำแนกได้เป็น 4 ประเภท คือ

ก. ซุปเปอร์คอมพิวเตอร์ (Super Computer) กล่าวคือ เครื่องประมวลผลข้อมูลที่มีประสิทธิภาพในการประมวลผลสูงที่สุด โดยทั่วไปสร้างเพื่อใช้งานเฉพาะด้าน เช่น งานด้านวิทยาศาสตร์ที่ต้องการการประมวลผลที่ซับซ้อน และความเร็วสูง เช่น งานวิจัยขีปนาวุธ งานสื่อสารดาวเทียม หรืองานพยากรณ์อากาศ เป็นต้น

ข. เมนเฟรมคอมพิวเตอร์ (Mainframe Computer) กล่าวคือ เครื่องประมวลผลข้อมูลที่มีความสามารถรองลงมาจากเครื่องซุปเปอร์คอมพิวเตอร์ อย่างไรก็ตามเครื่องยังมีประสิทธิภาพสูงและสามารถรองรับการทำงานในระบบเครือข่าย (Network) ได้เป็นอย่างดี สามารถทำงานพร้อมๆ กันหลายงาน (Multi Tasking) และใช้งานได้พร้อมกันหลายคน (Multi User) เช่น คอมพิวเตอร์ของธนาคารที่เชื่อมต่อไปยังตู้กดเงินอัตโนมัติ (ATM Machine) ทั่วประเทศ เป็นต้น

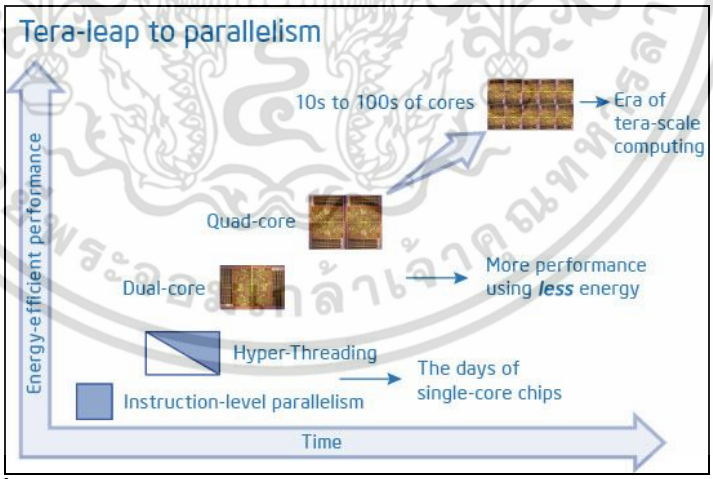
ค. มินิคอมพิวเตอร์ (Mini Computer) กล่าวคือ เครื่องประมวลผลที่มีขนาดเล็กและราคาถูกกว่าสองประเภทแรก โดยสามารถทำงานร่วมกับอุปกรณ์ประกอบต่างๆ ที่มีความเร็วสูงได้ หน่วยงานที่ใช้เครื่องมินิคอมพิวเตอร์ได้แก่ โรงพยาบาล มหาวิทยาลัย กระทรวง หรือโรงงานอุตสาหกรรม เป็นต้น

ง. ไมโครคอมพิวเตอร์ (Micro Computer) กล่าวคือ เครื่องประมวลผลข้อมูลขนาดเล็กที่มีประสิทธิภาพในการประมวลผลน้อย เรียกอีกอย่างหนึ่งว่า “คอมพิวเตอร์ส่วนบุคคล” (Personal Computer: PC) การทำงานในปัจจุบันนี้ไมโครคอมพิวเตอร์มีประสิทธิภาพสูงกว่าในอดีตมาก อาจเท่ากับหรือมากกว่าเครื่องซุปเปอร์คอมพิวเตอร์ในสมัยก่อน นอกจากนั้นราคาลดลงมาก จำแนกออกได้เป็น 2 ประเภทๆ คือ แบบติดตั้งใช้งานอยู่กับที่บนโต๊ะทำงาน (Desktop Computer) และแบบเคลื่อนย้ายได้ (Portable Computer) อาทิเช่น แลบท้อป (Laptop Computer) หรือ โน้ตบุ๊ค (Notebook Computer) เป็นต้น

จากการแบ่งประเภทของคอมพิวเตอร์ที่กล่าวมาข้างต้นจะเห็นว่าสามารถแบ่งได้หลากหลายแง่มุม ส่วนในวิทยานิพนธ์นี้จะเน้นไปที่ระบบคอมพิวเตอร์ที่มีหลายหน่วยประมวลผลในเครื่องเดียวกัน โดยแบ่งประเภทของคอมพิวเตอร์ตามสถาปัตยกรรมคอมพิวเตอร์แบบมัลติคอร์ ซึ่งแบ่งการทำงานออกเป็น 3 ลักษณะ ดังนี้ 1) การทำงานแบบหนึ่งหน่วยประมวลผล (Single Core) 2) การทำงานแบบไฮเปอร์เทรดดิ้ง (Hyper-Threading) และ 3) การทำงานแบบมัลติคอร์ (Multi-Cores) โดยใช้ข้อมูลอ้างอิงของหน่วยประมวลผล (CPU) จากบริษัทอินเทล (Intel) [20] เป็นหลัก

2.1 สถาปัตยกรรมคอมพิวเตอร์แบบมัลติคอร์ (Multi-Cores Computer Architecture)

สถาปัตยกรรมคอมพิวเตอร์แบบมัลติคอร์ [20] นี้สามารถจำแนกได้ตามลักษณะการทำงานของหน่วยประมวลผลกลาง ซึ่งลักษณะการทำงานดังกล่าวมีวิวัฒนาการของหน่วยประมวลผลกลางแบ่งเป็น 3 ลักษณะ ดังนี้ 1) การทำงานแบบหนึ่งหน่วยประมวลผล (Single Core) มีเพียงหน่วยประมวลผลเดียวในการทำงาน สำหรับงานที่มีปริมาณมากอาจต้องใช้เวลานาน จากความต้องการในการประมวลผลที่มากขึ้น ทำให้มีการพัฒนาความสามารถของหน่วยประมวลผลให้มีความเร็วเพิ่มขึ้นเรื่อยๆ จนเมื่อถึงจุดที่ยากจะพัฒนาต่อไปได้ จึงเข้าสู่ลักษณะการทำงานแบบที่ 2) การทำงานแบบไฮเปอร์เทรดดิ้ง (Hyper-Threading) เทคโนโลยีที่เรียกว่า “ไฮเปอร์เทรดดิ้ง” เข้ามาช่วยในการประมวลผล โดยหนึ่งหน่วยประมวลผลทำงานได้มากกว่า 1 คำสั่ง แต่มิได้ทำการเพิ่มจำนวนหน่วยประมวลผล อย่างไรก็ตามวิธีการนี้ยังไม่ใช่การทำงานแบบขนานอย่างแท้จริง ในขณะที่ความต้องการด้านประสิทธิภาพในการประมวลผลมีมากขึ้น ทำให้เข้าสู่การทำงานแบบที่ 3) การทำงานแบบหลายคอร์ หรือ มัลติคอร์ (Multi-Cores) เพื่อตอบสนองการประมวลผลที่ต้องการสมรรถนะสูง เช่น งานด้านการคำนวณข้อมูลขนาดใหญ่ การจำลองทางการแพทย์ การประมวลผลภาพถ่ายทางอากาศ เป็นต้น โดยการทำงานของหน่วยประมวลผลทั้ง 3 ลักษณะที่ได้กล่าวมานี้สามารถแสดงวิวัฒนาการได้ดังรูปที่ 2.1 การทำงานแบบหลายคอร์ เริ่มจาก 2 คอร์ (Dual Core) หรือแบบ 4 คอร์ (Quad Core) เรื่อยไปจนระดับ 10-100 คอร์ ซึ่งเรียกว่า “เทราสเกล” (Tera-scale Computing) [27] ซึ่งรายละเอียดในแต่ละแบบได้อธิบายในหัวข้อถัดไป

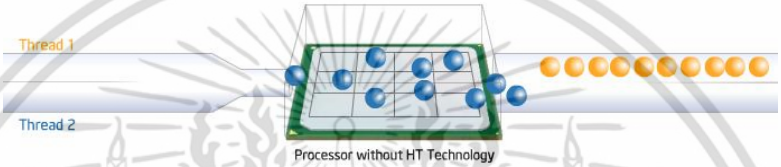


รูปที่ 2.1 วิวัฒนาการของหน่วยประมวลผลกลางแบบขนานแบบมัลติคอร์

2.1.1 การทำงานแบบหนึ่งหน่วยประมวลผล (Single Processor)

การทำงานแบบหนึ่งหน่วยประมวลผล (Single Processor) [20] สามารถรับคำสั่งเข้าสู่กระบวนการทำงานได้เพียงที่ละคำสั่งเท่านั้น ดังนั้นการเพิ่มความเร็วในการทำงานของหน่วยประมวลผลกลาง ในยุคนี้จึงเน้นที่การเพิ่มความเร็วในการรับคำสั่งเข้าไปประมวลผลโดยมีหน่วย

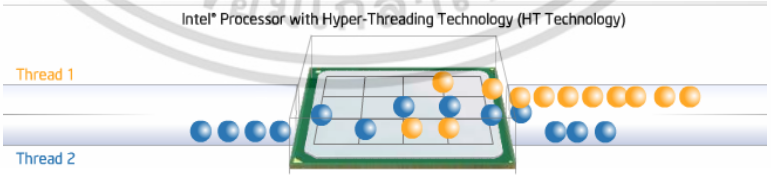
เป็นเฮิร์ต (Hertz) หรือรอบต่อวินาที เริ่มจากปี 1982 บริษัทอินเทลได้ผลิตหน่วยประมวลผลกลางรุ่น 80286 ความเร็วอยู่ที่ 6 เมกะเฮิร์ต (MHz) หรือ 6 ล้านรอบต่อวินาที เป็นหน่วยประมวลผลกลางรุ่นแรกๆ ที่นำมาใช้ในเครื่องคอมพิวเตอร์และได้รับความนิยมอย่างแพร่หลาย ต่อมาในปี 1999 ความเร็วของประมวลผลกลาง ได้เพิ่มขึ้นมีหน่วยเป็นกิกะเฮิร์ต (GHz) ซึ่งถือว่าเร็วมากในสมัยนั้น จากรูปที่ 2.2 แสดงให้เห็นว่ามีคำสั่งในการทำงานรออยู่ 2 คำสั่ง (Instruction) สามารถทำงานได้ที่ละคำสั่งเท่านั้น นอกจากนี้ต้องทำงานตามคำสั่งนั้นตั้งแต่ต้นจนจบให้เสร็จ จึงสามารถทำงานในคำสั่งถัดไปได้ อย่างไรก็ตามเมื่อถึงจุดหนึ่งการเพิ่มความเร็วในการประมวลผลทำได้ยากขึ้น ทำให้เข้าสู่ยุคของการนำเทคโนโลยีไฮเปอร์เทรดดิ้งเข้ามาใช้งาน



รูปที่ 2.2 การทำงานแบบหนึ่งหน่วยประมวลผล (Single Processor)

2.1.2 การทำงานแบบไฮเปอร์เทรดดิ้ง (Hyper-Threading)

การทำงานแบบไฮเปอร์เทรดดิ้ง [20] เป็นการเพิ่มความเร็วในการทำงานของหน่วยประมวลผลกลาง เสมือนว่ามีหน่วยประมวลผลมากกว่าหนึ่งหน่วยประมวลผลทำงานไปพร้อมๆ กัน ดังรูปที่ 2.3 แสดงการทำงานแบบหนึ่งหน่วยประมวลผลที่ใช้เทคโนโลยีไฮเปอร์เทรดดิ้ง จากรูปแสดงการจัดเรียงคิวในการทำงานสลับกันทั้ง 2 คำสั่งให้สามารถทำงานไปพร้อมๆ กันได้ แทนที่จะทำคำสั่งแรกให้เสร็จก่อน แล้วคำสั่งที่สองค่อยเริ่มทำงานเหมือนในแบบแรก ด้วยวิธีการนี้ส่งผลให้ทำงานได้เร็วขึ้น แต่อย่างไรก็ตามยังไม่ใช่การประมวลผลแบบขนานที่แท้จริง จึงนำไปสู่การทำงานแบบมัลติคอร์ (Multi-Cores)

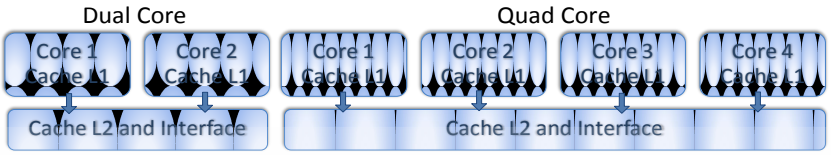


รูปที่ 2.3 การทำงานแบบหนึ่งหน่วยประมวลผลใช้ไฮเปอร์เทรดดิ้ง (Hyper-Threading)

2.1.3 การทำงานแบบมัลติคอร์ (Multi-Cores)

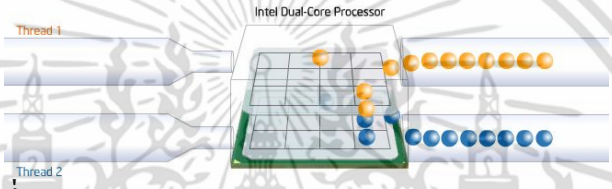
การทำงานแบบมัลติคอร์ [20] ภายในหน่วยประมวลผลจะมีหน่วยประมวลผลย่อย เรียกว่าคอร์ (Core) มากกว่าหนึ่งคอร์ ซึ่งในแต่ละคอร์มีหน่วยความจำ เรียกว่าแคชระดับที่1 (Cache L1) แต่ละคอร์มีการใช้หน่วยความจำร่วมกันเรียกว่าแคชระดับที่2 (Cache L2) การเข้าถึงข้อมูลที่อยู่ภายใน

แคชระดับที่1 ทำได้รวดเร็วกว่าการเข้าถึงข้อมูลภายในแคชระดับ2 หรือการเข้าถึงข้อมูลจากหน่วยความจำหลัก (Main Memory) ในรูปที่ 2.4 แสดงโครงสร้างของหน่วยประมวลผลแบบสองคอร์ (Dual Core) และแบบสี่คอร์ (Quad Core) ซึ่งการทำงานของหน่วยประมวลผลในแบบมัลติคอร์นี้สามารถประมวลได้พร้อมๆ กัน ซึ่งเป็นการทำงานแบบขนานอย่างแท้จริง



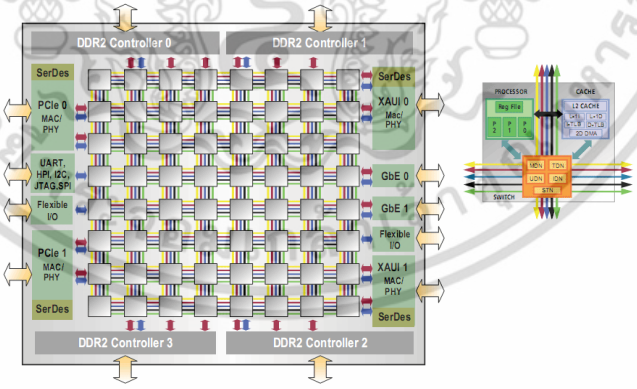
รูปที่ 2.4 โครงสร้างของหน่วยประมวลผลแบบสองคอร์ และสี่คอร์

ในรูปที่ 2.5 แสดงการทำงานของหน่วยประมวลผลแบบสองคอร์ ในแต่ละคอร์ประมวลผลคำสั่งไปพร้อมๆ กันได้



รูปที่ 2.5 การประมวลผลของหน่วยประมวลผลแบบสองคอร์

นอกจากการประมวลผลแบบมัลติคอร์ซึ่งตอบสนองการทำงานที่ต้องการสมรรถนะสูงแล้ว การพัฒนาให้มีจำนวนคอร์ตั้งแต่หลักสิบคอร์ไปจนถึงหลักร้อยคอร์อยู่บนหน่วยประมวลผลเดียวกันเรียกชิพ (Chip) แบบนี้ว่า “ไทล์โปรเซสเซอร์” (Tile Processor) [27] โดยมีโครงสร้างหน่วยประมวลผลแบบไทล์ ดังรูปที่ 2.6



รูปที่ 2.6 โครงสร้างของหน่วยประมวลผลของ “TILE64 Processor”

2.2 การเรียงลำดับแบบอนุกรม (Sequential Sorting)

โจทย์ปัญหาทางคอมพิวเตอร์ส่วนใหญ่ทั้งที่เป็นทฤษฎีขั้นตอนวิธีและการประยุกต์ใช้ จะเกี่ยวข้องอย่างมากกับการเรียงลำดับข้อมูลชนิดตัวเลขจากน้อยไปหามากหรือจากมากไปหาน้อย ตลอดจนการจัดลำดับกลุ่มของตัวอักษรต่างๆ การเรียงลำดับนี้มีประโยชน์โดยตรง ในการ

ประยุกต์ใช้เพื่อแก้ปัญหาเกี่ยวกับการใช้งานฐานข้อมูล โดยเฉพาะอย่างยิ่งการค้นหาข้อมูลที่ต้องการภายในระยะเวลาอันรวดเร็ว นอกจากนี้การเรียงลำดับข้อมูลยังมีความสำคัญต่องานด้านต่างๆ ทางธุรกิจ ทางวิทยาศาสตร์ และวิศวกรรมศาสตร์ ที่ประมวลผลด้วยคอมพิวเตอร์ ดังนั้นการเรียนรู้เทคนิคการเรียงลำดับจึงจำเป็น เพื่อ

1) สามารถเลือกวิธีการเรียงลำดับที่เหมาะสมกับงานที่ทำมากที่สุด

2) เป็นแนวทางในการคิดค้นวิธีการเรียงลำดับใหม่ ๆ ที่ดียิ่งขึ้น (ไม่ซ้ำแบบเก่า)

นอกจากนี้การเรียงลำดับยังมีประโยชน์อื่นๆ อีกหลายประการดังนี้

1) ช่วยในการจัดหมวดหมู่ข้อมูล (Classify) ให้สะดวกขึ้น

2) ช่วยในการค้นหาข้อมูล (Searching) ให้เร็วขึ้น

3) ช่วยให้การจับคู่ข้อมูล (Matching) เช่น การปรับปรุงแฟ้มลำดับมีประสิทธิภาพยิ่งขึ้น

การเรียงลำดับข้อมูลในปัจจุบันสามารถแบ่งได้เป็น 2 แบบใหญ่ๆ คือ

1. การเรียงลำดับแบบอนุกรม (Sequential Sort)
2. การเรียงลำดับแบบขนาน (Parallel Sort)

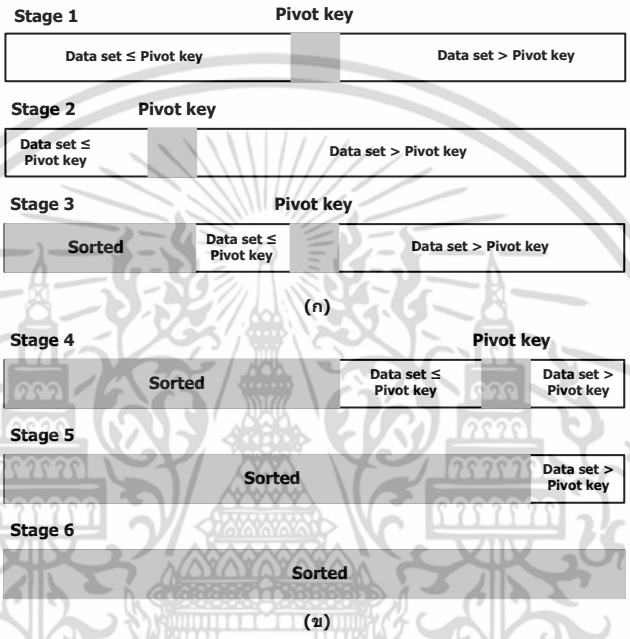
การเรียงลำดับข้อมูลโดยทั่วไปเป็นการเรียงลำดับข้อมูลแบบอนุกรม ซึ่งเป็นกระบวนการจัดเรียงลำดับข้อมูลในตาราง หรือแฟ้มข้อมูลให้เรียงตามลำดับข้อมูลจากน้อยไปหามาก (Ascending-order Sequence) หรือเรียงลำดับข้อมูลจากมากไปหาน้อย (Descending-order Sequence) ซึ่งขั้นตอนวิธี สำหรับการเรียงลำดับข้อมูลแบบอนุกรมขนาด N แบบต่างๆ นี้ มีผู้คิดขึ้นมากมายด้วยความซับซ้อนด้านเวลา เท่ากับ O(N $\log_2$ N) อาทิ การเรียงลำดับแบบเร็ว (Quick Sort) [15][29] การเรียงลำดับแบบรวมผสานกัน (Merge Sort) [15][29] การเรียงลำดับแบบฮีป (Heap Sort) [15][29] โดยแต่ละวิธีต่างมีข้อดีข้อด้อยที่แตกต่างกัน ขึ้นอยู่กับลักษณะและปริมาณข้อมูลที่ต้องการเรียงลำดับ รวมถึงประสิทธิภาพของเครื่องคอมพิวเตอร์ที่ใช้ในการเรียงลำดับข้อมูลอีกด้วย ในวิทยานิพนธ์นี้ให้ความสนใจการเรียงลำดับแบบอนุกรม คือ การเรียงลำดับแบบเร็ว ซึ่งได้รับความนิยมอย่างแพร่หลายในปัจจุบัน

การเรียงลำดับแบบเร็ว [24] เป็นวิธีการเรียงลำดับที่เหมาะสำหรับข้อมูลขนาดใหญ่ เนื่องจากใช้เวลาไม่นาน ผู้นำเสนอวิธีนี้ คือ C.A.R. Hoare ในปี 1962 ซึ่งเป็นวิธีการเรียงลำดับที่ได้รับความนิยมเป็นอย่างมากเนื่องจากความซับซ้อนด้านเวลาเฉลี่ย เพียง O(N $\log_2$ N) เมื่อ N คือขนาดของข้อมูลที่ใช้ในการเรียงลำดับ

การเรียงลำดับแบบเร็วมีแนวความคิดคือ การเลือกคีย์ (Pivot Key) จากกลุ่มข้อมูลมา 1 คีย์ จากนั้นใช้คีย์ที่เลือกเป็นคีย์หลักในการแบ่งข้อมูลออกเป็นสองส่วน โดยข้อมูลในส่วนแรกจะมีค่าน้อยกว่าหรือเท่ากับคีย์หลัก และข้อมูลในส่วนหลังจะมีค่ามากกว่าคีย์หลัก จากนั้นทำซ้ำเช่นเดิม จนกระทั่งข้อมูลไม่สามารถแบ่งต่อไปได้อีก สำหรับวิธีการเลือกคีย์เพื่อใช้ในการแบ่ง

ข้อมูลนี้มีด้วยกันหลายวิธี เช่น เลือกคีย์จากตำแหน่งแรก/ท้ายสุดของข้อมูล หรือเลือกคีย์จากค่ากลาง (Median Value) ของจำนวนข้อมูล เป็นต้น

จากรูปที่ 2.7 แสดงขั้นตอนการเรียงลำดับแบบเร็ว โดย รูปที่ 2.7 (ก) ในรอบที่ 1 (Stage 1) เป็นการนำค่าคีย์หลักจากค่าแรกไปไว้ในตำแหน่งที่เหมาะสม เช่น ค่ากลางของจำนวนข้อมูล จะได้ข้อมูลสองส่วนคือ ข้อมูลส่วนที่น้อยกว่าหรือเท่ากับค่าของคีย์หลักดังแสดงในรูปด้านซ้าย และข้อมูลส่วนที่มากกว่าคีย์หลักดังแสดงในรูปด้านขวา ทำเช่นนี้จนเรียงลำดับข้อมูลเสร็จดังรูปที่ 2.7 (ข) ในรอบที่ 6 แสดงการเรียงลำดับข้อมูลแบบเร็วเรียบร้อยแล้ว



รูปที่ 2.7 แสดงขั้นตอนการเรียงลำดับแบบเร็ว (Quick Sort)

ขั้นตอนวิธีที่ 1 ขั้นตอนวิธีการเรียงลำดับแบบเร็ว (Quick Sort)

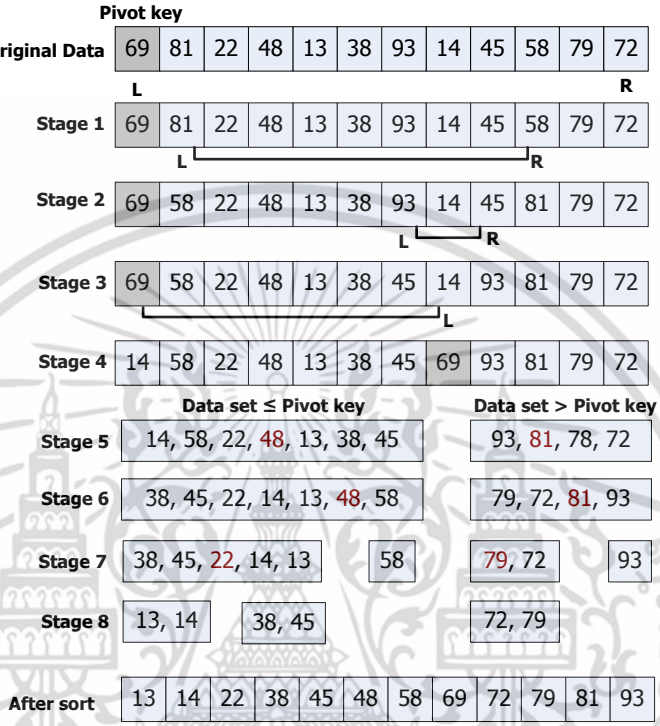
```
Quicksort(A[i...j])
    IF (i < j) THEN
      p = j + 1
      p = Partition(i, p)
      Quicksort(i, p-1)
      Quicksort(p+1, j)
    END IF
```

```
Partition(m, p)
  t = A[m]
  i = m
  while (i ≤ p)
    do i = i + 1 while (A[i] < t)
    do p = p - 1 while (A[p] > t)
    if(i < p) Exchange (A[i], A[p])
    else      Exchange (A[m], A[p])
  END while
  RETURN p
```

ตัวอย่างที่ 2.1 แสดงตัวอย่างการเรียงลำดับแบบเร็ว (Quick Sort)

สมมติให้ข้อมูลนำเข้าคือ 69, 81, 22, 48, 13, 38, 93, 14, 45, 58, 79, 72 โดยเลือกคีย์หลักคือ 69 (ค่าแรก) และให้ **L** แทนตำแหน่งทางซ้าย และ **R** แทนตำแหน่งทางขวา ในรอบที่ 1 เปรียบเทียบคีย์หลักกับข้อมูลด้านซ้ายคือ ถ้าข้อมูลน้อยกว่าจะเลื่อนไป 1 ตำแหน่งและหยุดเมื่อพบ

ข้อมูลมากกว่าคีย์หลักคือ 81 (A_i) จากนั้นเปรียบเทียบข้อมูลด้านขวาโดยถ้าข้อมูลด้านขวามีค่ามากกว่าคีย์หลักจะเลื่อนกลับ 1 ตำแหน่งและหยุดเมื่อพบข้อมูลน้อยกว่าคีย์หลักคือ 58 (A_p) กรณีนี้ค่า $i < p$ ดังนั้นสลับค่า $A_i = 81$ กับ ค่า $A_p = 58$ ทำต่อไปจนพบว่า $i \geq p$ จึงสลับค่า A_m กับ A_p ในรอบที่ 4 ทำเช่นนี้เรื่อยไปจนข้อมูลเรียงได้ถูกต้องสมบูรณ์ ดังแสดงในรูปที่ 2.8



รูปที่ 2.8 แสดงตัวอย่างการเรียงลำดับแบบเร็ว (Quick Sort)

2.3 การเรียงลำดับแบบขนานและงานวิจัยที่เกี่ยวข้อง (Parallel Sort and Related Work)

การเรียงลำดับแบบขนานถูกเสนอขึ้นสำหรับการเรียงลำดับข้อมูลขนาดใหญ่ เพราะหากข้อมูลมีขนาดใหญ่มากๆ การเรียงลำดับข้อมูลแบบอนุกรม ที่ใช้เพียงหนึ่งหน่วยประมวลผล ดังที่ได้กล่าวไว้ในข้อก่อน ต้องใช้เวลาในการเรียงลำดับข้อมูลเป็นเวลานาน จากปัญหาข้างต้นจึงมีการศึกษาวิจัยเกี่ยวกับการเรียงลำดับแบบขนานโดยใช้หลายๆ หน่วยประมวลผลและการจัดเก็บข้อมูลจำนวนที่เหมาะสมในแต่ละหน่วยประมวลผล จึงทำให้ความเร็วในการเรียงลำดับข้อมูลสูงขึ้น หรือได้ผลลัพธ์รวดเร็วขึ้นโดยเฉพาะความเร็วในการติดต่อสื่อสารระหว่างหน่วยประมวลผล

การเรียงลำดับแบบขนาน [3][22] เป็นขั้นตอนวิธีการที่มีความซับซ้อนกว่าแบบอนุกรมมาก นักวิจัยจึงจำเป็นต้องศึกษาค้นคว้า เพื่อทำความเข้าใจในเรื่องของการติดต่อสื่อสารระหว่างหน่วยประมวลผล และการประมวลผลแบบขนานที่ทำให้การทำงานเร็วขึ้น โดยขั้นตอนวิธีการเรียงลำดับแบบขนานที่นิยมใช้ในปัจจุบัน เช่น การเรียงลำดับแบบขนานด้วยวิธีไบโทนิค (Bitonic Sort)

[2][19][21][23] การเรียงลำดับแบบขนานด้วยวิธีรวมผสานกัน (Parallel Merge Sort) [19][23] และการเรียงลำดับแบบขนานด้วยวิธีคู่คี่ (Parallel Odd-Even Sort) [19][23] ด้วยความซับซ้อนด้านเวลาเท่ากับ $O(\log_2 N)^2$ เมื่อจำนวนหน่วยประมวลผลเท่ากับขนาดของข้อมูล (P=N) เป็นต้น นอกจากนี้ยังมีผู้นำเสนอวิธีการเรียงลำดับแบบใหม่ๆ เพิ่มขึ้นในปัจจุบันโดยมุ่งเน้นกรณี P<N เช่น การเรียงลำดับแบบขนานด้วยวิธีซีอีบีเอส (Communication-Effilicent Bitonic Sort: CEBS) ที่นำเสนอโดยคิม (KIM) [14] หรือ การเรียงลำดับแบบขนานด้วยวิธีแอลบีเอ็ม (Load-Balanced Parallel Merge Sort: LBM) [12] ที่พัฒนาต่อยอดจากวิธีซีอีบีเอส ซึ่งรายละเอียดจะกล่าวถึงในลำดับถัดไป ในวิทยานิพนธ์นี้ได้ศึกษาวิธีการเรียงลำดับแบบขนานแบบต่างๆ ที่มีอยู่ในปัจจุบันกรณี P<N เพื่อใช้เป็นแนวทางในการพัฒนาการเรียงลำดับแบบขนานที่ดีที่สุดต่อไป

2.3.1 การเรียงลำดับแบบขนานด้วยวิธีไบโทนิค (Parallel Bitonic Sorting)

การเรียงลำดับแบบไบโทนิค [2][15][19][23] นี้ได้ถูกคิดค้นขึ้นเมื่อ ค.ศ. 1968 โดย Kenneth E. Batcher ซึ่งเป็นการเรียงลำดับแบบขนานที่ได้รับความนิยมอย่างมากเนื่องจากความซับซ้อนด้านเวลา เพียง $O(\log_2 N)^2$ นับว่าเป็นเวลาที่ดีที่สุด โดยคำนิยามของการเรียงลำดับแบบขนานด้วยวิธีไบโทนิค คือ

นิยามที่ 1 การเรียงลำดับแบบขนานด้วยวิธีไบโทนิคเป็นการเรียงลำดับชุดของข้อมูลที่เรียงจากน้อยไปหามาก ($s_0 \le s_1 \le \ldots \le s_{N-1}$) หรือเรียงจากมากไปหาน้อย ($s_0 \ge s_1 \ge \ldots \ge s_{N-1}$) โดยก่อนที่มีการเรียงข้อมูลนั้น ข้อมูลนำเข้าต้องอยู่ในรูปแบบที่เรียกว่า “Bitonic Sequence” $a_0, a_1, a_2, \ldots, a_{N-1}$ ซึ่งประกอบด้วยข้อมูล 2 ชุดข้อมูลที่เรียงต่อกัน เมื่อ $a_0, \ldots, a_i$ เป็นชุดของข้อมูลย่อยที่เรียงจากน้อยไปหามาก และ $a_{i+1}, \ldots, a_{N-1}$ เป็นชุดของข้อมูลย่อยที่เรียงจากมากไปหาน้อย

ตัวอย่างเช่น ชุดข้อมูล 5, 6, 10, 15, 30, 28, 20, 12, 3, 1 เป็นข้อมูลแบบ ”Bitonic Sequence” เนื่องจากประกอบด้วยข้อมูล 2 ชุด ที่มีคุณสมบัติดังนี้ คือ

5, 6, 10, 15, 30 เป็นชุดของข้อมูลย่อยที่เรียงจากน้อยไปหามากและ

28, 20, 12, 3, 1 เป็นอีกชุดของข้อมูลย่อยที่เรียงจากมากไปหาน้อย

นิยามที่ 2 การเรียงลำดับแบบขนานด้วยวิธีไบโทนิค (กรณีที่ข้อมูลอยู่ในรูปแบบของ “Bitonic Sequence”) เมื่อกำหนดให้จำนวนข้อมูล (N) เท่ากับจำนวนหน่วยประมวลผล (P) หรือ P=N สามารถคำนวณหาจำนวนรอบในการทำงานได้ ดังสมการที่ 2.1 [15][31]

$$\text{จำนวนรอบ} = \log_2 P \qquad (2.1)$$

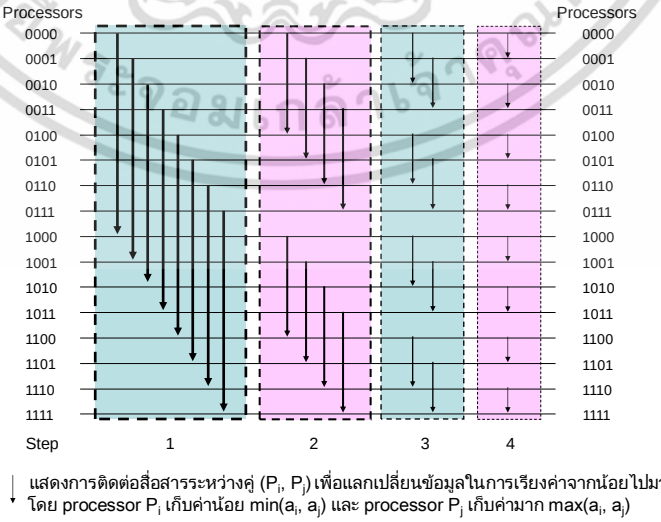
ในแต่ละรอบทุกคู่ของหน่วยประมวลผล (P_i, P_j) จะประมวลผลไปพร้อมๆ กัน การคำนวณค่า P_i และ P_j ที่ติดต่อสื่อสารเพื่อแลกเปลี่ยนข้อมูลระหว่างกันพิจารณาจากหลักการทางด้านการออกแบบสถาปัตยกรรม (Architecture Design) ของการเชื่อมต่อระหว่างหน่วยประมวลผลที่เหมาะสมเช่น การเชื่อมต่อแบบไฮเปอร์คิวบ์ (Hypercube Network) ก่อน ซึ่งจะพิจารณาค่า i และ j

เป็นเลขฐานสอง (Binary Number) แล้วจึงนำมาประยุกต์สำหรับการออกแบบขั้นตอนวิธี (Algorithm Design) ซึ่งพิจารณาค่า i และ j เป็นเลขฐานสิบ (Decimal Number) ในระบบเลขฐานสองค่า i และ j แสดงด้วยเลขฐานสิบจำนวน m บิต คือ i_{10} และ j_{10} โดย $m = \log_2 P$ คือ b_{m-1} b_{m-2} ... b_2 b_1 b_0 เมื่อ b มีค่าเท่ากับศูนย์หรือหนึ่ง

- ในรอบที่ 1 คู่ (P_i, P_j) มีค่าแตกต่างกันในบิตที่ b_{m-1} โดยที่ P_i มีค่า $b_{m-1} = 0$ หรือ $i_{10} =$ 0 b_{m-2} b_{m-3} ... b_2 b_1 b_0 และค่า P_j มีค่า $b_{m-1} = 1$ หรือ $j_{10} = 1$ b_{m-2} b_{m-3} ... b_2 b_1 b_0
- ในรอบที่ 2 คู่ (P_i, P_j) มีค่าแตกต่างกันในบิตที่ b_{m-2} โดยที่ P_i มีค่า $b_{m-2} = 0$ หรือ $i_{10} =$ b_{m-1} 0 b_{m-3} ... b_2 b_1 b_0 และค่า P_j มีค่า $b_{m-2} = 1$ หรือ $j_{10} = b_{m-1}$ 1 b_{m-3} ... b_2 b_1 b_0
- ในรอบที่ s ใดๆ คู่ (P_i, P_j) มีค่าแตกต่างกันในบิตที่ b_{m-s} โดยที่ P_i มีค่า $b_{m-s} = 0$ หรือ $i_{10} = b_{m-1}$ … b_{m-s+1} 0 b_{m-s-1} ... b_0 และค่า P_j มีค่า $b_{m-s} = 1$ หรือ $j_{10} = b_{m-1}$ … b_{m-s+1} 1 b_{m-s-1} ... b_0
- ในรอบที่ m (รอบสุดท้าย) คู่ (P_i, P_j) มีค่าแตกต่างกันในบิตที่ b_0 โดยที่ P_i มีค่า $b_0 = 0$ หรือ i_{10} $= b_{m-1}$ ... b_2 b_1 0 และค่า P_j มีค่า $b_0 = 1$ หรือ $j_{10} =$ b_{m-1} ... b_2 b_1 1

ในระบบเลขฐานสิบค่า i และ j คำนวณจากเงื่อนไขดังนี้คือ ในรอบที่ s ใดๆ (s = 1, 2, ... , m) สำหรับทุกค่าของ P_i เมื่อ i = 0, 1, 2, ..., P-1 (จำนวนหน่วยประมวลผล) จะได้ค่า j = i+d สำหรับ P_j ถ้า (i mod 2d) < d โดยที่ $d = 2^{m-s}$

ตัวอย่างเช่น รูปที่ 2.9 แสดงการเรียงลำดับแบบขนานด้วยวิธีไบโทนิค ใช้หน่วยประมวลผล 16 หน่วย (P=16) ดังนั้นจำนวนรอบเท่ากับ $\log_2 16=4$ รอบ ในแต่ละรอบทุกคู่ของหน่วยประมวลผล (P_i, P_j) จะประมวลผลไปพร้อมๆ กัน ส่วนการคำนวณค่า (P_i, P_j) ในแต่ละรอบแสดงไว้ในตารางที่ 2.1 เช่น ในรอบที่ 1 คู่(P_0, P_8), คู่(P_1, P_9), …, คู่(P_6, P_{14}), คู่(P_7, P_{15}) จะทำการติดต่อกันแบบขนานกัน ในรอบที่ 2 คู่(P_0, P_4), คู่(P_1, P_5), …, คู่(P_{10}, P_{14}), คู่(P_{11}, P_{15}) ในรอบที่ 3 คู่(P_0, P_2), คู่(P_1, P_3), …, คู่(P_{12}, P_{14}), คู่(P_{13}, P_{15}) และรอบที่ 4 คู่(P_0, P_1), คู่(P_2, P_3), …, คู่ (P_{12}, P_{13}), คู่(P_{14}, P_{15}) เป็นต้น



รูปที่ 2.9 การเรียงลำดับแบบไบโทนิค 16 หน่วยประมวลผล (กรณีข้อมูลเป็น Bitonic Sequence)

ตารางที่ 2.1 แสดงการหาค่า (P_i, P_j) สำหรับการเรียงลำดับแบบไบโทนิค เมื่อข้อมูลนำเข้าเป็น Bitonic Sequence จำนวน $m = \log_2 P$ รอบ กรณีที่จำนวนหน่วยประมวลผลเท่ากับขนาดข้อมูล (N=P=16)

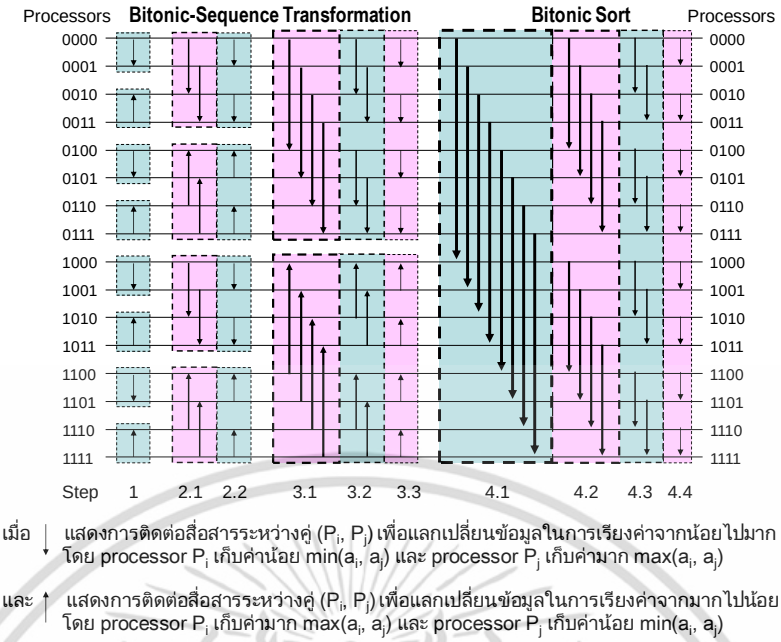
รอบ Step (s)	การหาค่า i และ j เมื่อ x เท่ากับศูนย์หรือหนึ่ง (0/1)			(P_i, P_j) เมื่อ i = 0, 1, …, 15 และ j = i+d ถ้า (i mod 2d) < d	
	$d = 2^{m-s}$	i	j	ฐานสอง	ฐานสิบ
1	$2^3 = 8$	0xxx	1xxx	(0000, 1000) (0001, 1001) … (0111, 1111)	(P_0, P_8) (P_1, P_9) … (P_7, P_{15})
2	$2^2 = 4$	x0xx	x1xx	(0000, 0100) (0001, 0101) … (1011, 1111)	(P_0, P_4) (P_1, P_5) … (P_{11}, P_{15})
3	$2^1 = 2$	xx0x	xx1x	(0000, 0010) (0001, 0011) … (1101, 1111)	(P_0, P_2) (P_1, P_3) … (P_{13}, P_{15})
4	$2^0 = 1$	xxx0	xxx1	(0000, 0001) (0010, 0011) … (1110, 1111)	(P_0, P_1) (P_2, P_3) … (P_{14}, P_{15})

นิยามที่ 3 การเรียงลำดับแบบขนานด้วยวิธีไบโทนิค (กรณีที่ข้อมูลนำเข้าเป็นแบบทั่วไป) ซึ่งประกอบด้วย 2 ขั้นตอนคือ ขั้นแรกจะต้องทำการแปลง (Transformation) ข้อมูลทั่วไป (Random Sequence) ให้อยู่ในรูปของ “Bitonic Sequence” เสียก่อน และขั้นตอนที่สองเป็นการเรียงข้อมูลแบบขนานด้วยวิธีไบโทนิค โดยกรณีทั่วไปนี้ การคำนวณหาจำนวนรอบในการเรียงลำดับข้อมูลแบบขนานด้วยวิธีไบโทนิค สามารถคำนวณรอบในการทำงานได้ $1+2+3+,\ldots,+\log_2 P$ (ผลรวมของรอบย่อยในแต่ละรอบใหญ่จำนวน $\log_2 P$ รอบ) ซึ่งเป็นค่าที่แสดงในสมการที่ 2.2 [15][31]

$$\text{จำนวนรอบ} = \frac{(1+\log_2 P)(\log_2 P)}{2} \qquad (2.2)$$

เนื่องจากจำนวนรอบในขั้นตอนที่สอง (ตามนิยามที่ 2) เท่ากับ $\log_2 P$ ดังนั้น จำนวนรอบในขั้นตอนแรกเท่ากับ $(1+\log_2 P)(\log_2 P)/2 - (\log_2 P) = (1+\log_2 P)/2-1)(\log_2 P)$ในแต่ละรอบ ทุกคู่ของหน่วยประมวลผล (P_i, P_j) จะประมวลผลไปพร้อมๆ กัน การคำนวณค่า P_i และ P_j ที่ติดต่อสื่อสารแลกเปลี่ยนข้อมูลระหว่างกัน จะพิจารณาได้ดังรูปที่ 2.10 (P=16) ในทำนองเดียวกับที่กล่าวมาในนิยามที่ 2 ดังรูปที่ 2.9 (P=16)

ตัวอย่างเช่น รูปที่ 2.10 แสดงการเรียงลำดับข้อมูลแบบขนานด้วยวิธีไบโทนิค โดยใช้หน่วยประมวลผล 16 หน่วย (P=16) ดังนั้นจำนวนรอบเท่ากับ (1+4)(4)/2 = 10 รอบ ในแต่ละรอบทุกคู่ของหน่วยประมวลผล (P_i, P_j) จะประมวลผลไปพร้อมๆ กันโดยการคำนวณค่า (P_i, P_j) ที่ติดต่อสื่อสารเพื่อแลกเปลี่ยนข้อมูลระหว่างกันดังแสดงไว้ในตารางที่ 2.2



รูปที่ 2.10 การเรียงลำดับแบบไบโทนิคใช้ 16 หน่วยประมวลผล (กรณีข้อมูลทั่วไป)

ตารางที่ 2.2 แสดงการหาค่า (P_i, P_j) สำหรับการเรียงลำดับแบบไบโทนิค เมื่อข้อมูลนำเข้าเป็นแบบทั่วไปจำนวน $(1+\log_2 P)(\log_2 P)/2$ รอบ กรณีที่จำนวนหน่วยประมวลผลเท่ากับขนาดข้อมูล (N=P=16)

รอบนอก Stage	รอบย่อย (s)	การหาค่า i และ j เมื่อ x เท่ากับศูนย์หรือหนึ่ง (0/1)			(P_i, P_j) เมื่อ i = 0, 1, …, 15 และ j = i+d ถ้า (i mod 2d) < d	
		$d = 2^{m-s}$	i	j	ฐานสอง	ฐานสิบ
1	1	$2^0 = 1$	xxx0	xxx1	(0000, 0001), ..., (1110, 1111)	(P_0, P_1), ..., (P_{14}, P_{15})
2	1	$2^1 = 2$	xx0x	xx1x	(0000, 0010), ..., (1101, 1111)	(P_0, P_2), ..., (P_{13}, P_{15})
	2	$2^0 = 1$	xxx0	xxx1	(0000, 0001), ..., (1110, 1111)	(P_0, P_1), ..., (P_{14}, P_{15})
3	1	$2^2 = 4$	x0xx	x1xx	(0000, 0100), ..., (1011, 1111)	(P_0, P_4), ..., (P_{11}, P_{15})
	2	$2^1 = 2$	xx0x	xx1x	(0000, 0010), ..., (1101, 1111)	(P_0, P_2), ..., (P_{13}, P_{15})
	3	$2^0 = 1$	xxx0	xxx1	(0000, 0001), ..., (1110, 1111)	(P_0, P_1), ..., (P_{14}, P_{15})
4	1	$2^3 = 8$	0xxx	1xxx	(0000, 1000), ..., (0111, 1111)	(P_0, P_8), ..., (P_7, P_{15})
	2	$2^2 = 4$	x0xx	x1xx	(0000, 0100), ..., (1011,1111)	(P_0, P_4), ..., (P_{11}, P_{15})
	3	$2^1 = 2$	xx0x	xx1x	(0000, 0010), ..., (1101,1111)	(P_0, P_2), ..., (P_{13}, P_{15})
	4	$2^0 = 1$	xxx0	xxx1	(0000, 0001), ..., (1110,1111)	(P_0, P_1), ..., (P_{14}, P_{15})

หมายเหตุ กรณีนี้ m เป็นจำนวนรอบนอก โดย m = 1, 2, 3, ..., $\log_2 P$ และเมื่อ m=4 ในตารางที่ 2.2 จะเหมือนกับกรณีที่แสดงไว้ในตารางที่ 2.1

การเรียงลำดับแบบขนานด้วยวิธีไบโทนิค โดยปกติจะเป็นกรณีที่สมมติว่ามีจำนวนหน่วยประมวลผล (P) มากเพียงพอสำหรับข้อมูลขนาด N (P=N) แต่ในทางปฏิบัติหรือกรณีทั่วๆ ไป จำนวนหน่วยประมวลผลมักจะน้อยกว่าขนาดของข้อมูล (P<N) ขั้นตอนวิธีของแต่ละกรณีจะแสดงในหัวข้อย่อยต่อไปนี้

2.3.1.1 **ขั้นตอนวิธีแบบไบโทนิคกรณีที่จำนวนหน่วยประมวลผลเท่ากับจำนวนข้อมูล (P = N)**

การเรียงลำดับแบบขนานด้วยวิธีไบโทนิคสำหรับกรณีที่ 1 เป็นข้อมูล “Bitonic Sequence” และกรณีที่ 2 เป็นข้อมูลทั่วไป นั้นจะมีขั้นตอนวิธี [23] โดยทั่วไปดังนี้ เมื่อจำนวนหน่วยประมวลผล (P) เท่ากับจำนวนข้อมูล (N) หรือ P=N

ขั้นตอนวิธีที่ 2 ขั้นตอนวิธีการเรียงลำดับด้วยวิธีไบโทนิค (P=N) สำหรับข้อมูล “Bitonic Sequence”

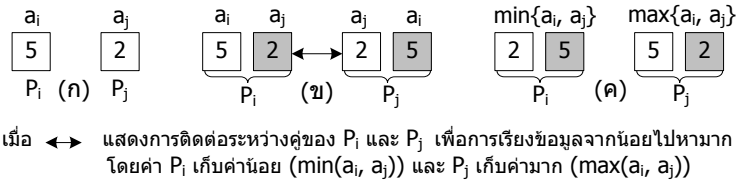
```
Global      d: Distance between elements being compare
Local       a: One of the elements to be sorted
            t: Element retrieved from adjacent processor
Begin
        for J = m-1 downto 0 do
            d = 2^J
        // For All Processor
                for all Processor k where 0 <= k <= 2m-1 pardo
                    if k mod 2d < d then
                        t = [k + d]a
                        [k+d]a = max(t, a) //sort low to high
                        a =  min(t, a)
                    end if
                end for  all
        end for J
end
```

ขั้นตอนวิธีที่ 3 ขั้นตอนวิธีการเรียงลำดับด้วยวิธีไบโทนิค (P=N) สำหรับข้อมูลทั่วไป

```
Global      d: Distance between elements being compare
Local       a: One of the elements to be sorted
            t: Element retrieved from adjacent processor
Begin
        for I = 0 up to m-1 do   // m = log2 N
            for J = I downto 0 do
                d = 2
                for all Processor k where 0 <= k <= 2m-1 pardo
                    if k mod 2d < d then
                        t =[k + d]a
                        if  k mod  2i+2 < 2i+1  then
                            [k+d]a =  max(t, a) //sort low to high
                                a =  min(t, a)
                        else
                            [k+d]a =  min(t, a) //sort high to low
                                a =  max(t, a)
                        end if
                    end if
                end for all
            end for J
        end for I
end
```

ในกรณีที่จำนวนหน่วยประมวลผล (P) เท่ากับจำนวนข้อมูล (N) ดังนั้นเริ่มแรกหน่วยประมวลผลแต่ละหน่วย (P_i) จะเก็บค่าของข้อมูลไว้เพียงหนึ่งค่า (a_i) ขั้นต่อไปจะทำการเปรียบเทียบค่าในหน่วยประมวลผลที่ P_i กับ P_j เป็นคู่ๆ ($i < j$) โดยค่าที่ได้หลังจากเปรียบเทียบนั้นจะกำหนดให้ค่าน้อยถูกเก็บที่หน่วยประมวลผล P_i และค่ามากถูกเก็บไว้ที่หน่วยประมวลผล P_j ซึ่งเรียกตัวดำเนินการแบบนี้ว่า “Compare-Exchange” [15][31] หลักการของ Compare-Exchange สามารถอธิบายดังรูปที่ 2.11 (ก) แสดงการโหลดข้อมูลเข้าสู่หน่วยประมวลผลของ P_i และ P_j โดย P_i เก็บค่า 5 และ P_j เก็บค่า 2 รูปที่ 2.11 (ข) แสดงการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผล P_i และ P_j

โดยที่ข้อมูลในกรอบสีเทาคือข้อมูลที่ถูกส่งมาจากหน่วยประมวลผลอื่น รูปที่ 2.11(ค) เปรียบเทียบค่าที่รับมากับค่าที่มีอยู่โดย P_i จะเก็บค่าน้อย ($\min(a_i, a_j)$) ส่วน P_j จะเก็บค่ามาก ($\max(a_i, a_j)$)



รูปที่ 2.11 ตัวดำเนินการแบบ “Compare-Exchange”

สมมติให้หน่วยประมวลผลที่ P_i และ P_j อยู่ติดกันและการติดต่อกัน (เช่นการเชื่อมต่อแบบไฮเปอร์คิวบ์) สามารถติดต่อถึงกันได้โดยตรง ทำให้เวลาที่ใช้ในการติดต่อสื่อสารในขั้นตอนของ Compare-Exchange ตามสมการที่ 2.3

$$T_S + T_W \tag{2.3}$$

เมื่อ T_S คือ เวลาเริ่มต้นในการติดต่อกับหน่วยประมวลผลอื่น

T_W คือ เวลาที่ใช้ในการแลกเปลี่ยนข้อมูล 1 ค่า

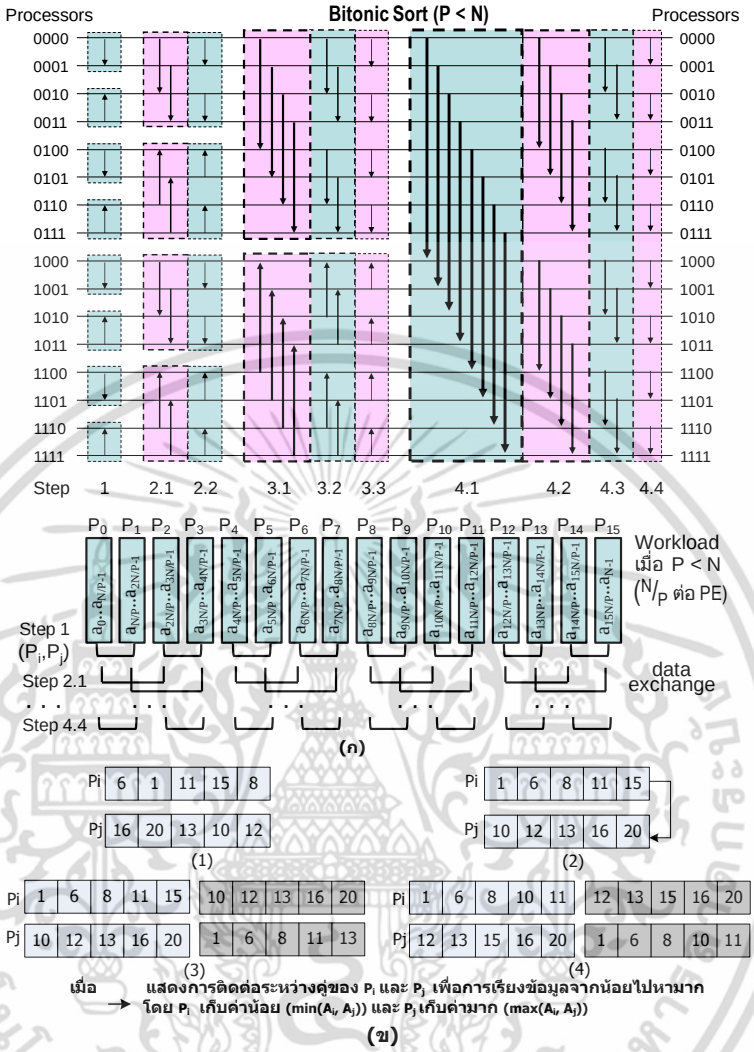
2.3.1.2 **ขั้นตอนวิธีแบบไบโทนิคกรณีที่จำนวนหน่วยประมวลผลน้อยกว่าจำนวนข้อมูล (P < N)**

ในกรณีที่จำนวนหน่วยประมวลผลน้อยกว่าจำนวนข้อมูล (P < N) กำหนดให้ P เป็นจำนวนของหน่วยประมวลผลโดยเรียงตั้งแต่ $P_0, P_1, P_2, \ldots, P_{P-1}$ และให้ N เป็นจำนวนข้อมูล ($a_0, a_1, \ldots, a_{N-1}$) ซึ่งข้อมูลทั้งหมดจะถูกแบ่งออกเป็นกลุ่มของข้อมูลขนาด n ซึ่งเท่ากับ N/P ค่า โดยกำหนดให้ $A_0, A_1, A_2, \ldots, A_{P-1}$ เป็นชุดของข้อมูลจำนวน P กลุ่มๆ ละ n จำนวน เมื่อ $A_i = a_{in}, a_{in+1}, a_{in+2}, \ldots, a_{in+(n-1)}$ และ i = 0, 1, 2, ..., P-1 (เช่น $A_0 = a_0, a_1, \ldots, a_{n-1}$, $A_1 = a_n, a_{n+1}, \ldots, a_{2n-1}, \ldots, A_{P-1} = a_{(P-1)n}, a_{(P-1)n+1}, \ldots, a_{N-1}$) ที่เป็นภาระงาน (Workload) ของแต่ละหน่วยประมวลผล (P_i) ดังแสดงในรูปที่ 2.12 (ก)

ในการเปรียบเทียบการเรียงลำดับแบบไบโทนิค เมื่อจำนวนหน่วยประมวลผลน้อยกว่าจำนวนข้อมูล ในกรณีข้อมูลทั่วๆ ไป ต้องพยายามจัดข้อมูลเป็น 2 ชุด ตามเงื่อนไขดังนี้คือ $A_i <= A_j$ (ซึ่งทุกๆ ค่าที่อยู่ใน A_i จะน้อยกว่าหรือเท่ากับค่าที่อยู่ใน A_j) หรือ $A_i >= A_j$ (ซึ่งทุกๆ ค่าที่อยู่ใน A_i จะมากกว่าหรือเท่ากับค่าที่อยู่ใน A_j) ตามรูปแบบของไบโทนิค ในการจัดการเพื่อให้ได้ข้อมูลในรูปแบบดังกล่าวจะใช้ตัวดำเนินการในการเปรียบเทียบค่า ซึ่งเรียกว่า “Compare-Split” [15][31]

ตัวอย่างเช่น รูปที่ 2.12 (ข) แสดงการโหลดข้อมูลขนาด w=N/P= 5 เมื่อแต่ละหน่วยประมวลผล P_i เก็บค่า (6, 1, 11, 15, 8) และ P_j เก็บค่า (16, 20, 13, 10, 12) ไว้ในหน่วยความจำส่วนตัว และทำการเรียงลำดับข้อมูลย่อยๆ ขนาด N/P ด้วยวิธีแบบเร็ว ในทุกหน่วยประมวลผลพร้อมๆ กัน รูปที่ 2.12 (ข-2) แสดงข้อมูลที่เรียงแล้วและการติดต่อสื่อสารระหว่าง P_i กับ P_j รูปที่ 2.12 (ข-3) แสดงการ

แลกเปลี่ยนชุดข้อมูลระหว่าง P_i กับ P_j รวมถึงการรวมชุดข้อมูลทั้ง 2 ชุดเข้าด้วยกัน รูปที่ 2.12 (ข-4) เปรียบเทียบค่าที่รับมากับค่าที่มีอยู่โดย P_i เก็บค่าน้อย ($\min(A_i, A_j)$) และ P_j เก็บค่ามาก ($\max(A_i, A_j)$)



รูปที่ 2.12 (ก) การเรียงแบบไบโทนิคกรณี P < N, P=16 (ข) ตัวดำเนินการแบบ "Compare-Split"

สมมติให้ P_i กับ P_j เป็นหน่วยประมวลผลที่เชื่อมต่อถึงกันโดยตรง ดังนั้นเวลาที่ใช้ในการเปรียบเทียบค่าโดยใช้ตัวดำเนินการดังกล่าว ตามสมการที่ 2.4

$$T_s + wT_w \qquad (2.4)$$

เมื่อ T_s คือ เวลาเริ่มต้นในการติดต่อกับหน่วยประมวลผลอื่น

$w = \frac{N}{P}$ คือ จำนวนข้อมูลที่เป็นภาระงาน (Workload) สำหรับแต่ละหน่วยประมวลผล

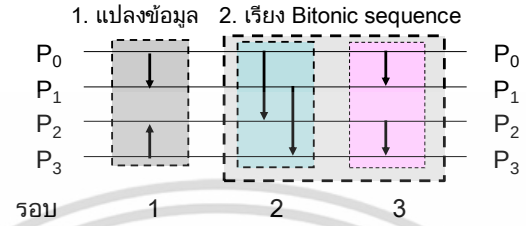
T_w คือ เวลาที่ใช้ในการแลกเปลี่ยนข้อมูล

ตัวอย่างที่ 2.2 แสดงตัวอย่างการเรียงลำดับแบบไบโทนิค กรณีข้อมูลนำเข้าอยู่ในรูปทั่วไป

สมมติข้อมูลนำเข้า (N=20) คือ 29, 21, 22, 27, 19, 5, 13, 0, 9, 3, 16, 7, 8, 14, 6, 17, 15, 18, 24, 20 และจำนวนหน่วยประมวลผล (P=4) น้อยกว่าจำนวนข้อมูล (P<N) และข้อมูลยังไม่ได้ถูก

เรียงให้อยู่ในรูปของ “Bitonic Sequence” ดังนั้นการเรียงลำดับแบบขนานด้วยวิธีไบโทนิคจึงจำเป็นต้องใช้ 2 ขั้นตอนหลักในการเรียงลำดับข้อมูล ตามนิยามที่ 3 และสามารถคำนวณหาจำนวนรอบในการเรียงลำดับข้อมูลได้ คือ $(1+\log_2 P)(\log_2 P)/2 = (1+\log_2 4)(\log_2 4)/2 = 3$ รอบโดยมีรายละเอียดดังต่อไปนี้คือ

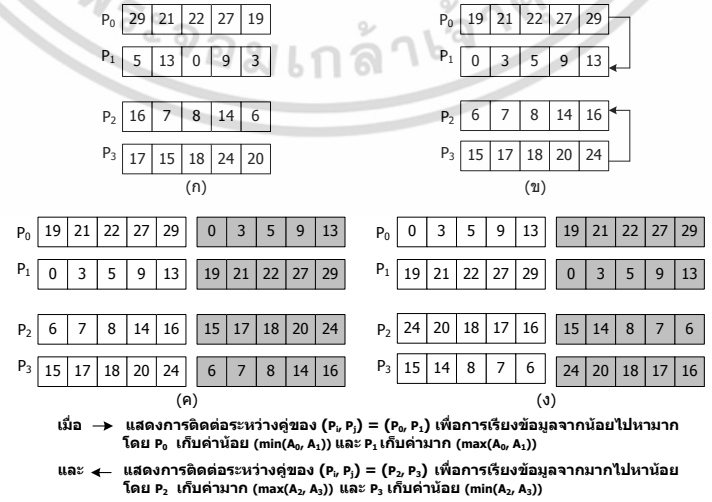
รูปแบบการติดต่อสื่อสารของ (P_i, P_j) ในแต่ละรอบ



แสดงดัง รูปที่ 2.13 รูปที่ 2.14 รูปที่ 2.15

ขั้นแรก การแปลงข้อมูลทั่วไปให้อยู่ในรูปของ “Bitonic Sequence” ซึ่งสามารถคำนวณหาจำนวนรอบการทำงานได้ โดยนำจำนวนรอบในขั้นที่สอง ($\log_2 P$) มาหักออกจากจำนวนทั้งหมดซึ่งเท่ากับ $(1+\log_2 P)(\log_2 P)/2 - (\log_2 P) = 3 - 2 = 1$ รอบ

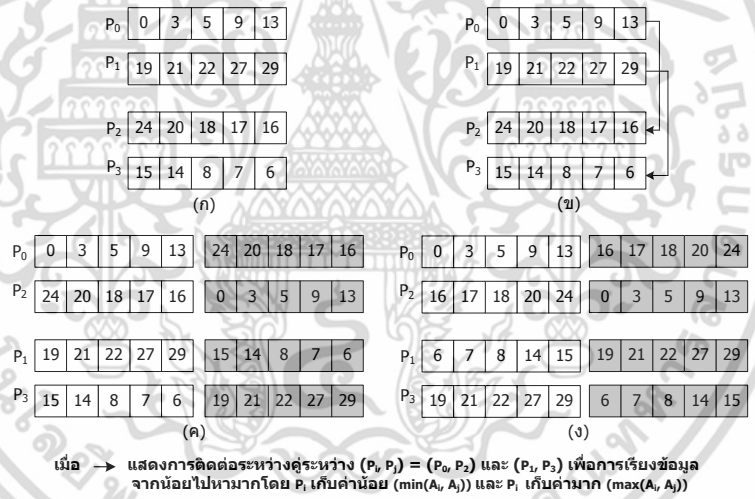
รอบที่ 1 การแปลงข้อมูลทั่วไปให้อยู่ในรูปของ “Bitonic Sequence” ดัง รูปที่ 2.13 (ก) แสดงการโหลดข้อมูลขนาด w=N/P เมื่อแต่ละหน่วยประมวลผล P_0 เก็บค่า (29, 21, 22, 27, 19) P_1 เก็บค่า (5, 13, 0, 9, 3) P_2 เก็บค่า (16, 7, 8, 14, 6) และ P_3 เก็บค่า (17, 15, 18, 24, 20) ไว้ในหน่วยความจำส่วนตัวของแต่ละหน่วยประมวลผล (P_i) และทำการเรียงลำดับข้อมูลขนาด N/P ด้วยวิธีแบบเร็วในทุกหน่วยประมวลผลพร้อมๆ กัน ส่วนรูปที่ 2.13 (ข) แสดงข้อมูลที่เรียงแล้วและแสดงการติดต่อสื่อสารระหว่างหน่วยประมวลผล (P_0,P_1, P_2, P_3: กรณี P=4) โดยทุกๆ คู่ของหน่วยประมวลผล (P_i, P_j) จะแลกเปลี่ยนข้อมูลไปพร้อมๆ กัน เช่น คู่(P_0, P_1) และ คู่ (P_2, P_3) ในรูปที่ 2.13 (ค) แสดงการรวมชุดข้อมูลทั้ง 2 ชุดเข้าด้วยกัน และรูปที่ 2.13 (ง) แสดงภาระงานหลังจากเปรียบเทียบข้อมูลระหว่างหน่วยประมวลผล (P_i, P_j)



รูปที่ 2.13 การแปลงข้อมูลจากข้อมูลทั่วไปเป็นข้อมูล “Bitonic Sequence”

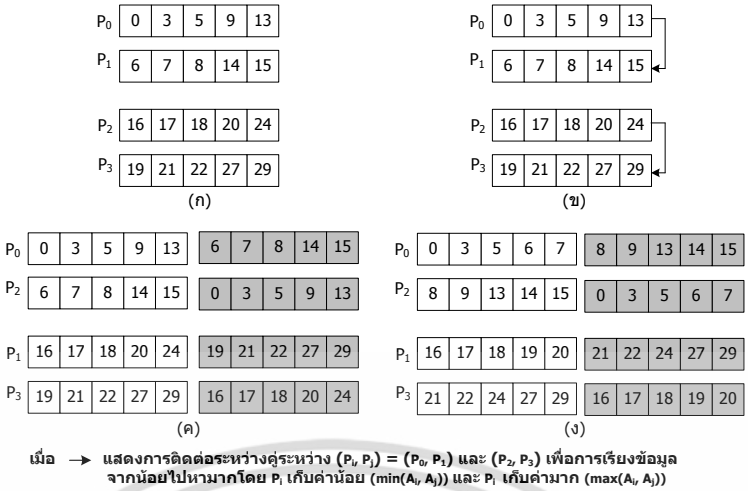
<u>ขั้นที่สอง</u> การเรียงลำดับแบบขนานด้วยวิธีไบโทนิค สำหรับข้อมูลที่ได้รับการแปลงจากขั้นแรกคือ (0, 3, 5, 9, 13), (19, 21, 22, 27, 29) ใน P_0, P_1 และ (24, 20, 18, 17, 16), (15, 14, 8, 7, 6) ใน P_2, P_3 ซึ่งอยู่ในรูปของ “Bitonic Sequence” สามารถทำการเรียงลำดับข้อมูลแบบไบโทนิคด้วยจำนวนรอบการทำงานที่คำนวณได้จากนิยามที่ 2 คือ $(\log_2 P) = (\log_2 4) = 2$ รอบ ดังนี้

รอบที่ 1 การเรียงลำดับแบบขนานด้วยวิธีไบโทนิค ดังรูปที่ 2.14 (ก) แสดงการโหลดข้อมูลขนาด w=N/P เมื่อแต่ละหน่วยประมวลผล P_0 เก็บค่า (0, 3, 5, 9, 13), P_1 เก็บค่า (19, 21, 22, 27, 29), P_2 เก็บค่า (24, 20, 18, 17, 16) และ P_3 เก็บค่า (15, 14, 8, 7, 6) ภายในหน่วยความจำส่วนตัวของแต่ละหน่วยประมวลผล (P_i) โดยไม่ต้องทำการเรียงลำดับข้อมูลย่อยขนาด w=N/P เพราะข้อมูลนี้อยู่ในรูปของ “Bitonic Sequence” และมีการเรียง 2 ชุดข้อมูลย่อยไว้แล้ว ส่วนรูปที่ 2.14 (ข) แสดงการติดต่อสื่อสารระหว่างหน่วยประมวลผล (P_0,P_1, P_2, P_3: P=4) โดยทุกๆ คู่ของหน่วยประมวลผล (P_i, P_j) จะแลกเปลี่ยนข้อมูลไปพร้อมๆ กัน เช่น คู่ (P_0, P_2) และคู่ (P_1, P_3) ในรูปที่ 2.14 (ค) แสดงการรวมผลสานชุดข้อมูล 2 ชุดเข้าด้วยกัน และรูปที่ 2.14 (ง) แสดงภาระงานหลังจากเปรียบเทียบข้อมูลระหว่างหน่วยประมวลผล (P_i, P_j) ซึ่ง P_i เก็บค่าน้อย ($\min(A_i, A_j)$) และ P_j เก็บค่ามาก ($\max(A_i, A_j)$)



รูปที่ 2.14 การเรียงลำดับแบบขนานด้วยวิธีไบโทนิค (ข้อมูล “Bitonic Sequence”) รอบที่ 1

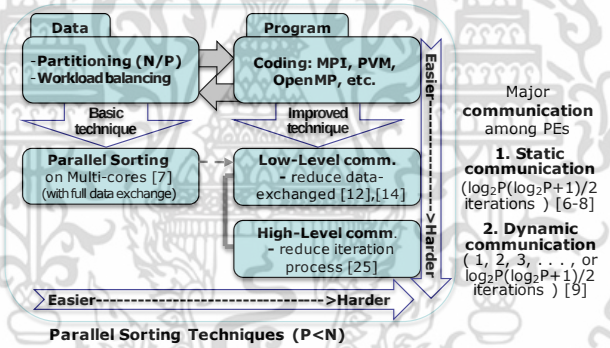
รอบที่ 2 การเรียงลำดับแบบขนานด้วยวิธีไบโทนิค ดังรูปที่ 2.15 (ก) แสดงข้อมูลจากรอบที่ 1 เมื่อแต่ละหน่วยประมวลผล P_0 เก็บค่า (0, 3, 5, 9, 13), P_1 เก็บค่า (6, 7, 8, 14, 15), P_2 เก็บค่า (16, 17, 18, 20, 24) และ P_3 เก็บค่า (19, 21, 22, 27, 29) ภายในหน่วยความจำส่วนตัว ของแต่ละหน่วยประมวลผล (P_i) รูปที่ 2.15 (ข) แสดงการติดต่อสื่อสารระหว่างหน่วยประมวลผล (P_0,P_1, P_2, P_3 : P=4) โดยทุกๆ คู่ของหน่วยประมวลผล (P_i, P_j) จะแลกเปลี่ยนข้อมูลไปพร้อมๆ กัน เช่น คู่ (P_0, P_1) และคู่ (P_2, P_3) ในรูปที่ 2.15 (ค) แสดงการรวมชุดข้อมูลทั้ง 2 ชุดที่เรียงลำดับแล้วเข้าด้วยกัน และรูปที่ 2.15 (ง) แสดงภาระงานหลังจากเปรียบเทียบข้อมูลระหว่างหน่วยประมวลผล (P_i, P_j) โดยให้ P_i เก็บค่าน้อย ($\min(A_i, A_j)$) และ P_j เก็บค่ามาก ($\max(A_i, A_j)$)



รูปที่ 2.15 การเรียงลำดับแบบขนานด้วยวิธีไบโทนิค (Bitonic Sequence) รอบที่ 2

2.3.2 งานวิจัยที่เกี่ยวข้องกับการเรียงลำดับแบบขนานกรณี P < N

สำหรับวิวัฒนาการและแนวทางการศึกษาวิจัยงานด้านการเรียงลำดับข้อมูลแบบขนาน ในกรณีที่ P< N นั้นสามารถแสดงแผนผังได้ดังรูปที่ 2.16



รูปที่ 2.16 โครงสร้างแผนผังงานวิจัยที่เกี่ยวข้องในการเรียงลำดับแบบขนาน P < N ในปัจจุบัน

การเรียงลำดับแบบขนานแบ่งเวลาที่ใช้ในการเรียงลำดับเป็น 2 ส่วนคือ เวลาที่ใช้ในการติดต่อสื่อสารระหว่างหน่วยประมวลผล (Communication Time) และเวลาที่ใช้ในการประมวลผล (Computation Time) ซึ่งส่วนใหญ่เวลาที่เสียไปในการเรียงลำดับแบบขนานนี้จะเกิดจากการติดต่อสื่อสารระหว่างหน่วยประมวลผล เนื่องจากขั้นตอนวิธีของการเรียงลำดับแบบขนานนั้นมีการติดต่อสื่อสารระหว่างหน่วยประมวลผลตลอดเวลาเพื่อแลกเปลี่ยนข้อมูลกัน ดังนั้นจากงานวิจัยการเรียงลำดับแบบขนานที่ได้ศึกษาจากอดีตจนถึงปัจจุบันส่วนใหญ่จะเน้นที่การติดต่อสื่อสารระหว่างหน่วยประมวลผลแบบขนาน ซึ่งสามารถแบ่งได้เป็น 4 กลุ่ม ดังนี้

1) การเรียงลำดับแบบขนานบนระบบคอมพิวเตอร์แบบมัลติคอร์
2) การเรียงลำดับแบบขนานบนระบบการใช้หน่วยความจำร่วมกัน
3) การเรียงลำดับแบบขนานบนระบบการใช้หน่วยความจำแบบกระจาย
4) การเรียงลำดับแบบขนานบนระบบคลัสเตอร์

การเรียงลำดับแบบขนานทั้ง 4 แบบข้างต้นนี้ [2] ได้รับความสนใจจากนักวิจัยรุ่นต่อๆ มาอย่างมากโดยเฉพาะระบบคอมพิวเตอร์แบบมัลติคอร์ โดยงานวิจัยส่วนใหญ่จะให้ความสนใจในเรื่องของการเพิ่มประสิทธิภาพด้านการติดต่อสื่อสารระหว่างหน่วยประมวลผล และการแบ่งภาระงาน (Workload) สำหรับแต่ละหน่วยประมวลผล

สำหรับงานวิจัยที่นำเสนอในวิทยานิพนธ์นี้มุ่งเน้นเรื่องการเพิ่มประสิทธิภาพของการติดต่อสื่อสารระหว่างหน่วยประมวลผลในการเรียงลำดับแบบไบโทนิคกรณีที่ P<N เพื่อให้สามารถประมวลผลได้อย่างมีประสิทธิภาพสูงสุดบนระบบคอมพิวเตอร์แบบขนาน (อาทิ ระบบมัลติคอร์ ระบบคลัสเตอร์ ฯลฯ) ดังจะกล่าวรายละเอียดต่อไปในบทที่ 3 ขั้นตอนวิธีการลดจำนวนรอบของการติดต่อสื่อสารแบบไดนามิค (Dynamic Communication)

2.3.2.1 การเรียงลำดับแบบขนานด้วยวิธีซีอีบีเอส

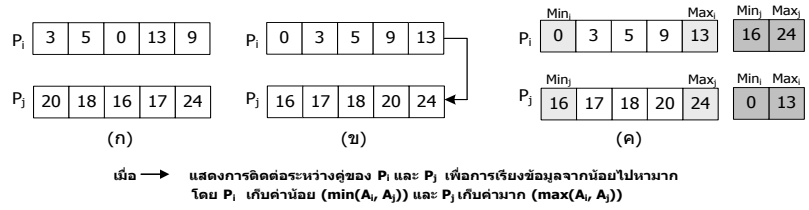
(Communication-Effilicent Bitonic Sort:CEBS)

การเรียงลำดับแบบขนานด้วยวิธีซีอีบีเอส (CEBS) [14] กรณี P<N เป็นวิธีที่เน้นเรื่องประสิทธิภาพของการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลด้วยภาระงาน N/P เพื่อค้นหาข้อมูลที่ใช้ในการแลกเปลี่ยนเฉพาะที่จำเป็นเท่านั้น เนื่องจากเวลาในการแลกเปลี่ยนข้อมูลจำนวนมากมีผลอย่างมากต่อเวลาในการเรียงลำดับ งานวิจัยนี้นำเสนอการแบ่งลักษณะการแลกเปลี่ยนข้อมูล 3 รูปแบบ จากแนวความคิดงานวิจัยเรื่องการเรียงลำดับข้อมูลแบบขนานด้วยวิธีไบโทนิคบนระบบคอมพิวเตอร์ Cray T3E [14] ซึ่งสามารถเพิ่มประสิทธิภาพในการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลได้อย่างน้อย 20% โดยการแลกเปลี่ยนข้อมูลในแต่ละแบบมีรายละเอียดดังนี้

แบบที่ 1) การแลกเปลี่ยนข้อมูลแบบคงที่ (Hold Pattern) การแลกเปลี่ยนข้อมูลแบบคงที่นี้เป็นกรณีที่ดีที่สุดสำหรับการเรียงลำดับ เนื่องจากข้อมูลได้ถูกเรียงลำดับไว้อย่างเหมาะสมแล้ว ดังนั้นจึงไม่ต้องมีการแลกเปลี่ยนข้อมูลทำให้เวลาที่ใช้ในการติดต่อสื่อสารระหว่างหน่วยประมวลผลลดลงและไม่ต้องทำการรวมผสานชุดข้อมูล 2 ชุดเข้าด้วยกัน เนื่องจากข้อมูลทั้ง 2 ชุดถูกจัดเรียงไว้เหมาะสมแล้วใน P_i และ P_j ทำให้เวลาที่ใช้ในการประมวลผลลดลง ดังรูปที่ 2.17 (ก) แสดงการโหลดข้อมูลขนาด N/P เมื่อแต่ละหน่วยประมวลผล P_i เก็บค่า 3 5 0 13 9 และ P_j เก็บค่า 20 18 16 17 24 ไว้ในหน่วยความจำส่วนตัว และทำการเรียงลำดับข้อมูลขนาด N/P ด้วยวิธีแบบเร็วในทุกหน่วยประมวลผลพร้อมๆ กัน รูปที่ 2.17 (ข) แสดงข้อมูลที่เรียงลำดับแล้วและแสดงการติดต่อสื่อสารระหว่างหน่วยประมวลผล P_i กับ P_j และรูปที่ 2.17 (ค) แสดงค่าน้อย (Min) และค่ามาก (Max) ในกรอบสีเทาเท่านั้นที่ต้องส่งและรับระหว่าง P_i และ P_j เพื่อตรวจสอบว่าเป็นการแลกเปลี่ยนข้อมูลแบบคงที่หรือไม่ ซึ่งมีเงื่อนไขการตรวจสอบดังต่อไปนี้

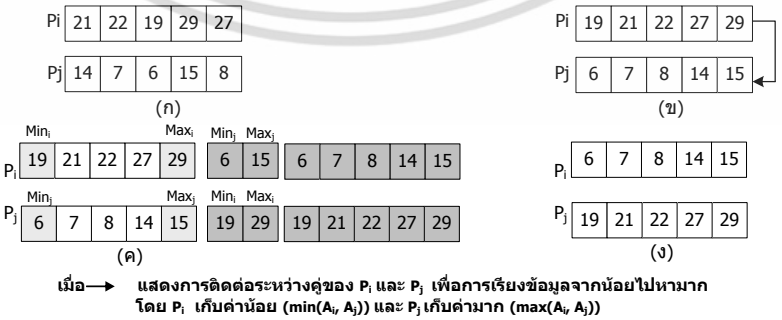
เมื่อค่า Min_j ที่รับมาจาก P_j มีค่ามากกว่าหรือเท่ากับค่า Max_i ใน P_i (หรือทุกๆ ค่าใน P_i นั้นเอง) และค่า Max_i ที่รับมาจาก P_i มีค่าน้อยกว่าหรือเท่ากับค่า Min_j ใน P_j (หรือทุกๆ ค่าใน P_j

นั้นเอง) แสดงว่าเป็นการแลกเปลี่ยนข้อมูลแบบคงที่ ดังนั้น ในกรณีนี้ไม่ต้องมีการแลกเปลี่ยนข้อมูลขนาด N/P ข้อมูลระหว่าง P_i และ P_j



รูปที่ 2.17 แสดงตัวอย่างวิธีการแลกเปลี่ยนข้อมูลแบบคงที่ (Hold Pattern)

แบบที่ 2) การแลกเปลี่ยนข้อมูลแบบสลับที่ (Swap Pattern) การแลกเปลี่ยนข้อมูลแบบสลับที่เป็นการแลกเปลี่ยนข้อมูลทั้งชุดโดยไม่ต้องทำการเรียงลำดับข้อมูลใหม่เนื่องจากข้อมูลทั้ง 2 ชุดถูกจัดเรียงไว้เหมาะสมแล้ว แต่ไม่ได้อยู่ใน P_i และ P_j ตามที่ต้องการ ดังนั้นกรณีนี้ P_i และ P_j ส่งข้อมูลทั้งหมดขนาด N/P ไปให้กับหน่วยประมวลผลที่ทำการติดต่อด้วยโดยไม่ต้องรวมผสานผลลัพธ์ ดังแสดงในรูปที่ 2.18 (ก) แสดงการโหลดข้อมูลขนาด N/P เมื่อแต่ละหน่วยประมวลผล P_i เก็บค่า 21, 22, 19, 29, 27 และ P_j เก็บค่า 14, 7, 6, 15, 8 ไว้ในหน่วยความจำส่วนตัว และทำการเรียงลำดับข้อมูลขนาด N/P ด้วยวิธีแบบเร็วในทุกหน่วยประมวลผลพร้อมๆ กัน รูปที่ 2.18 (ข) แสดงข้อมูลที่เรียงแล้วและแสดงการติดต่อสื่อสารระหว่างหน่วยประมวลผล P_i กับ P_j รูปที่ 2.18 (ค) แสดงค่าน้อย (Min) และค่ามาก (Max) ในกรอบสีเทาเท่านั้นที่ต้องส่งและรับระหว่าง P_i และ P_j เพื่อตรวจสอบว่าเป็นการแลกเปลี่ยนข้อมูลแบบสลับที่ โดยมีเงื่อนไขดังนี้ค่า Min_i ใน P_i มีค่ามากกว่าหรือเท่ากับค่า Max_j ที่รับมาจาก P_j (หรือทุกๆ ค่าใน P_j นั้นเอง) และค่า Max_jใน P_j มีค่าน้อยกว่าหรือเท่ากับค่า Min_i ที่รับมาจาก P_i (หรือทุกๆ ค่าใน P_i นั้นเอง) ถ้าเป็นไปตามเงื่อนไขดังกล่าวแสดงว่าเป็นการแลกเปลี่ยนข้อมูลแบบสลับที่ ดังนั้นในกรณีนี้จะทำการส่งและรับข้อมูลระหว่างหน่วยประมวลผล (P_i, P_j) ที่ทำการติดต่อด้วยเท่านั้น แต่ไม่ต้องทำการรวมผสานหรือเรียงลำดับข้อมูลใหม่ และรูปที่ 2.18 (ง) แสดงชุดข้อมูลที่รับมาเก็บแทนชุดข้อมูลที่มีอยู่เดิม โดย P_i จะเก็บค่าน้อย ($\min(A_i, A_j)$) ส่วน P_j จะเก็บค่ามาก ($\max(A_i, A_j)$)



รูปที่ 2.18 แสดงตัวอย่างวิธีการแลกเปลี่ยนข้อมูลแบบสลับที่ (Swap Pattern)

แบบที่ 3) การแลกเปลี่ยนข้อมูลแบบเฉพาะส่วน (Partial Pattern) โดยทั่วไปแล้วขนาดของข้อมูลหรือภาระงานในแต่ละหน่วยประมวลผลจะมีขนาดเท่ากับ N/P เมื่อ N คือจำนวนข้อมูล และ P คือจำนวนของหน่วยประมวลผล ดังนั้นการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลปกติมีค่าเท่ากับ N/P แต่สำหรับการแลกเปลี่ยนข้อมูลแบบเฉพาะส่วน จะต้องคำนวณหาชุดข้อมูลย่อยเฉพาะที่จำเป็นต้องส่งและรับเท่านั้น โดยส่วนใหญ่แล้วจะมีจำนวนน้อยกว่า N/P ทำให้สามารถลดทั้งเวลาที่ใช้ในการติดต่อสื่อสาร และเวลาที่ใช้ในการรวมผสานผลลัพธ์ ดังรูปที่ 2.19 (ก) แสดงการโหลดข้อมูลขนาด N/P เมื่อแต่ละหน่วยประมวลผล P_i เก็บค่า 6, 1, 11, 15, 8 และ P_j เก็บค่า 16, 20, 13, 10, 12 ไว้ในหน่วยความจำส่วนตัวของแต่ละหน่วยประมวลผล (P_i) และทำการเรียงลำดับข้อมูลขนาด N/P ด้วยวิธีแบบเร็วในทุกหน่วยประมวลผลพร้อมๆ กัน รูปที่ 2.19 (ข) แสดงการติดต่อสื่อสารระหว่าง P_i กับ P_j รูปที่ 2.19 (ค) แสดงการส่งค่าน้อย (Min) และค่ามาก (Max) ไปให้กับหน่วยประมวลผลที่ติดต่อด้วย เพื่อตรวจสอบหาชุดข้อมูลย่อยที่ต้องการแลกเปลี่ยนเท่านั้น จากนั้นนำค่า Min_j มาตรวจสอบกับค่าใน P_i เริ่มจากค่า Max_i ไปยังค่า Min_i โดยที่ค่า Min_j ต้องมีค่ามากกว่าค่าใน P_i ดังนั้น ชุดข้อมูลที่ต้องส่งไปให้กับ P_j เพื่อทำการเรียงลำดับข้อมูลคือค่าใน P_i ที่มากกว่าค่า Min_j (กรณีนี้กรอบเข้มใน P_i เป็นค่าส่งและกรอบสีเทาใน P_j เป็นค่ารับ) และในขณะเดียวกันนำค่า Max_i มาตรวจสอบกับค่าใน P_j เริ่มจากค่า Min_j ไปยังค่า Max_j โดยที่ค่า Max_i ต้องมีค่าน้อยกว่าค่าใน P_j ดังนั้นชุดข้อมูลที่ต้องส่งไปให้กับ P_i เพื่อทำการเรียงลำดับข้อมูลคือค่าใน P_j ที่น้อยกว่าค่า Max_i (กรณีนี้กรอบเข้มใน P_j เป็นค่าส่ง และกรอบสีเทาใน P_i เป็นค่ารับ) รูปที่ 2.19 (ง) เปรียบเทียบค่าที่รับมากับค่าที่มีอยู่ โดย P_i จะเก็บค่าน้อย ($\min(A_i, A_j)$) ส่วน P_j จะเก็บค่ามาก ($\max(A_i, A_j)$)

รูปที่ 2.19 แสดงตัวอย่างวิธีการแลกเปลี่ยนข้อมูลแบบเฉพาะส่วน (Partial Pattern)

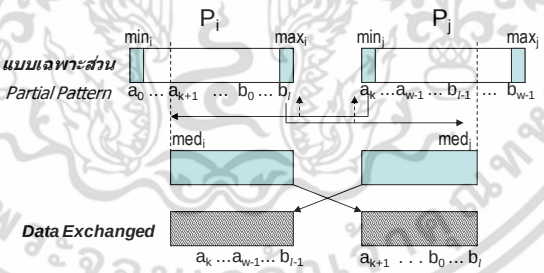
<u>หมายเหตุ</u> ในขั้นตอนการพัฒนาโปรแกรมด้วยวิธีซีอีบีเอส (CEBS) การแลกเปลี่ยนข้อมูลแบบคงที่จะถูกตรวจสอบก่อน ถ้าไม่เป็นไปตามเงื่อนไขดังกล่าว การแลกเปลี่ยนแบบสลับที่ จะถูกตรวจสอบลำดับถัดมา และถ้าไม่เป็นไปตามเงื่อนไขทั้งสองนี้ จึงทำการแลกเปลี่ยนแบบเฉพาะส่วน

ขั้นตอนวิธีที่ 4 การเรียงลำดับแบบขนานด้วยวิธีซีอีบีเอส (CEBS sort)

```
P: the total number of processors (assume P = 2^k)
Pi : a processor with index i
h : the number of active processors
Begin
    1. forall 0 ≤ i ≤ P-1
       Pi sorts a list of N/P keys locally
    2. for j = 0 to (log P)-1 do
       forall 0 ≤ i ≤ h-1
          if(i<h/2) then
             2.1 Pi receives N/h keys from Pi+h/2
             2.2 Pi merges two lists of N/h keys into a sorted list of 2N/h
          Else
             2.3 Pi sends its list to Pi+h/2
End
```

2.3.2.2 การเรียงลำดับแบบขนานด้วยวิธีแอลบีเอ็ม (Load-Balanced Parallel Merge Sort:LBM)

สำหรับวิธีการเรียงลำดับแบบขนานด้วยวิธีแอลบีเอ็ม (Load-Balanced Parallel Merge Sort:LBM) [12] นี้ได้นำเสนอในปี ค.ศ. 2003 เป็นการวิจัยต่อยอดจากการเรียงลำดับแบบขนานด้วยวิธีซีอีบีเอส โดยทำการศึกษาวิจัยเพิ่มเติมในส่วนของการแลกเปลี่ยนข้อมูลแบบเฉพาะส่วน ให้มีประสิทธิภาพเพิ่มขึ้นจากวิธีการเดิมซึ่งในบางครั้งจะพบว่าข้อมูลที่ส่งมีขนาดไม่เท่ากัน ทั้งที่แท้จริงการรวมข้อมูลของหน่วยประมวลผลทั้งสองฝั่งใช้ข้อมูลเท่ากันเสมอเมื่อเทียบกับค่ากลาง (median value) ของทั้ง 2 กลุ่ม ดังนั้นจึงเสนอวิธีการรวมผสานข้อมูล (Merge Data) ด้วยหลักการ “Median Computing” ส่งผลให้ประสิทธิภาพในการรวมชุดข้อมูลเพิ่มขึ้น ส่วนแบบคงที่ (Hold Pattern) และแบบสลับที่ (Swap Pattern) ยังคงใช้แบบเดียวกับวิธีการซีอีบีเอส รูปที่ 2.20 แสดงการแลกเปลี่ยนข้อมูลแบบเฉพาะส่วนของวิธีแอลบีเอ็ม



รูปที่ 2.20 แสดงการแลกเปลี่ยนข้อมูลแบบบางส่วนของวิธีแอลบีเอ็ม

ขั้นตอนวิธีที่ 5 การเรียงลำดับแบบขนานด้วยวิธีแอลบีเอ็ม (LBM Sort)

```
1.  Each processor sorts a list of N/P keys locally.
2.  Iterate log P times the following computation:
    i iteration, there are P/2^(i-1) groups, each of which includes 2^(i-1) PEs
    2.1  Each group of processors finds its pair (partner group).
    2.2  Exchange boundary values between paired groups.
    2.3  Determine overlapped interval(s) and communication partner(s).
    2.4  Determine the logical ids of processors that give a correct
         sequence of keys in the merged list.
    2.5  Perform the following with each communication partners.
         2.5.1  Find a splitter in each overlapped interval by binary search
         2.5.2  Exchange keys lying in overlapped intervals.
         2.5.3  Merge keys, and make a sorted list which has about N/P
                keys per processors
    2.6  Broadcast logical ids of processors
```

จากตัวอย่างเดียวกับการเรียงลำดับแบบขนานด้วยวิธีซีอีบีเอส กล่าวคือ ดังรูปที่ 2.21 (ก) แสดงการโหลดข้อมูลขนาด N/P เมื่อแต่ละหน่วยประมวลผล P_i เก็บค่า (6, 1, 11, 15, 8) และ P_j เก็บค่า (16, 20, 13, 10, 12) ไว้ในหน่วยความจำส่วนตัวของแต่ละหน่วยประมวลผล (P_i) และทำการเรียงลำดับข้อมูลขนาด N/P ด้วยการเรียงลำดับแบบเร็ว ในทุกหน่วยประมวลผลพร้อมๆ กัน รูปที่ 2.21 (ข) แสดงการติดต่อสื่อสารระหว่าง P_i กับ P_j รูปที่ 2.21 (ค) แสดงการส่งค่าน้อย (Min) และค่ามาก (Max) ไปให้กับหน่วยประมวลผลที่ติดต่อด้วย เพื่อตรวจสอบหาชุดข้อมูลย่อยที่ต้องการแลกเปลี่ยนเท่านั้น โดยคำนวณค่าประมาณของ Median ในตัวอย่างนี้ Med' = $(Min_j + Max_i)/2$ = (10+15)/2 = 12.5 เพราะค่า Median (11+12)/2 = 11.5 จริง ยังไม่สามารถคำนวณได้ เนื่องจากต้องมีการแลกเปลี่ยนข้อมูล จากนั้นนำค่า Med' มาตรวจสอบกับค่าใน P_i ด้วยวิธีการค้นหาแบบไบนารี โดยที่ค่า Med' ต้องมีค่ามากกว่าค่าใน P_i ดังนั้น ชุดข้อมูลที่ต้องส่งไปให้กับ P_j คือค่าที่มากกว่าหรือเท่ากับ Med' (15) เพื่อทำการเรียงลำดับข้อมูล (สำหรับวิธีการแบบซีอีบีเอส ชุดข้อมูลที่ต้องส่งคือ (11, 15) ซึ่งมากกว่า) และในขณะเดียวกันนำค่า Med' มาตรวจสอบกับค่าใน P_j ด้วยวิธีการค้นหาแบบไบนารี โดยที่ค่า Med' ต้องมีค่าน้อยกว่าค่าใน P_j ดังนั้นชุดข้อมูลที่ต้องถูกส่งไปให้กับ P_i คือค่าที่น้อยกว่าหรือเท่ากับ Med' (10, 12) เพื่อทำการเรียงลำดับข้อมูล (สำหรับวิธีการแบบซีอีบีเอส ชุดข้อมูลที่ต้องส่งคือ (10, 12, 13) ซึ่งมากกว่า) รูปที่ 2.21 (ง) เปรียบเทียบชุดข้อมูลที่รับมากับชุดข้อมูลที่มีอยู่ โดย P_i จะเก็บค่าน้อย ($\min(A_i, A_j)$) ส่วน P_j จะเก็บค่ามาก ($\max(A_i, A_j)$)

รูปที่ 2.21 แสดงตัวอย่างการแลกเปลี่ยนข้อมูลแบบบางส่วนของวิธีแอลบีเอ็ม

2.3.2.3 การเรียงลำดับแบบขนานบนระบบคอมพิวเตอร์แบบมัลติคอร์

การเรียงลำดับแบบขนานบนระบบคอมพิวเตอร์แบบมัลติคอร์ มีโครงสร้างที่ไม่ซับซ้อน เนื่องจากเครื่องคอมพิวเตอร์ส่วนบุคคล (Personal Computer: PC) ในปัจจุบันส่วนใหญ่รองรับการทำงานบนระบบคอมพิวเตอร์แบบมัลติคอร์ ทำให้ผู้ที่สนใจงานด้านการประมวลผลแบบขนานสามารถศึกษาวิจัยได้สะดวก ส่งผลให้งานวิจัยด้านการเรียงลำดับแบบขนานบนระบบคอมพิวเตอร์แบบมัลติคอร์ได้รับความสนใจ ศึกษาค้นคว้ากันอย่างกว้างขวาง ดังตัวอย่างงานวิจัยต่อไปนี้

ในปี ค.ศ. 2008 J. Chhuhani และคณะ [6] เสนองานวิจัยที่อธิบายถึงการเรียงลำดับแบบขนานด้วยวิธีรวมกัน (Parallel Merge Sort) บนระบบคอมพิวเตอร์แบบมัลติคอร์ ที่เน้นการทำงาน

แบบหนึ่งคำสั่งสามารถประมวลผลข้อมูลได้หลายๆ ชุดข้อมูลนั่นเอง (Single Instruction Multiple Data: SIMD) นอกจากนี้งานวิจัยได้นำเสนอวิธีการรวมข้อมูลแบบหลายทาง (Multi-Way Merge) อีกด้วย

ในปี ค.ศ. 2011 A. El-Nashar [7] เสนองานวิจัยที่อธิบายถึงการเรียงลำดับข้อมูลแบบขนานบนระบบมัลติคอร์ในระบบปฏิบัติการวินโดว์ (Windows) ซึ่งพัฒนาการเขียนโปรแกรมแบบขนานด้วยมาตราฐานภาษาเอ็มพีไอ (MPI) พัฒนาขั้นตอนวิธีการเรียงลำดับข้อมูล 3 วิธี คือ 1) การเรียงลำดับแบบเร็ว (Quick Sort) 2) การเรียงลำดับแบบรวมกัน (Merge Sort) และ 3) การเรียงลำดับแบบสบู่ (Bubble Sort) จากนั้นเปรียบเทียบการเรียงลำดับทั้งสามแบบโดยการเพิ่มจำนวนเทรด และการเพิ่มจำนวนคอร์ในการทำงาน ได้ข้อสรุปว่าการเพิ่มจำนวนคอร์ ส่งผลต่อการทำงานมากกว่าการเพิ่มจำนวนเทรด

2.3.2.4 การเรียงลำดับแบบขนานบนระบบการใช้หน่วยความจำร่วมกัน

การเรียงลำดับแบบขนานบนระบบการใช้หน่วยความจำร่วมกัน มีโครงสร้างการติดต่อสื่อสารระหว่างหน่วยประมวลผล (Interconnection Network) เป็นแบบไดนามิค (Dynamic) เช่น Omega, Baseline, Crossbar Switch และ Butterfly เป็นต้น ดังตัวอย่างงานวิจัยต่อไปนี้

ในปี ค.ศ. 2000 Adler, M. Byers, J.W. Karp, R.M [1] ได้นำเสนอผลงานวิจัยที่พยายามจะปรับปรุงประสิทธิภาพในการติดต่อสื่อสารระหว่างหน่วยประมวลผล โดยใช้โมเดลที่เรียกว่า “ER-PRAM Model” โดยจะแลกเปลี่ยน (Tradeoff) ระหว่างเวลาที่ใช้ในการติดต่อสื่อสารระหว่างหน่วยประมวลผล ที่อาจจะเพิ่มขึ้นเมื่อใช้หน่วยประมวลผล (P) มากขึ้น กับเวลาที่ใช้ในการประมวลผลที่ลดลงเมื่อใช้หน่วยประมวลผลมากขึ้น ด้วยวิธี “Lower Bound” และนำการเรียงลำดับแบบคอลัมน์ (Column Sort) ที่เสนอโดย Leighton [17] มาทำการจับคู่ (Matching) ด้วยวิธีการ “Upper Bound” ซึ่งสามารถประยุกต์เป็นโมเดลใหม่ที่เรียกว่า “Bridging Model” ที่สามารถใช้ในสถาปัตยกรรมอื่นๆ ได้

ในปี ค.ศ. 2000 Jae-Dong Lee และ Kenneth E. Batcher [16] เสนองานวิจัยนี้โดยทำการศึกษาการเรียงลำดับแบบไบโทนิค บนสถาปัตยกรรมแบบใช้หน่วยความจำร่วมกัน (Shared Memory) แต่การเข้าถึงข้อมูลในหน่วยความจำที่ใช้ร่วมกันนี้ค่อนข้างใช้เวลานาน ดังนั้นจึงเกิดแนวความคิดของกลยุทธ์แบบพาริตี้ (Parity Strategy) ขึ้น โดยการลดเวลาในการติดต่อสื่อสารระหว่างหน่วยประมวลผลเพื่อช่วยให้การเรียงลำดับข้อมูลมีประสิทธิภาพมากขึ้น

2.3.2.5 การเรียงลำดับแบบขนานบนระบบการใช้หน่วยความจำแบบกระจาย

การเรียงลำดับแบบขนานบนระบบการใช้หน่วยความจำแบบกระจายนี้ จะมีโครงสร้างการติดต่อสื่อสารระหว่างหน่วยประมวลผลแบบคงที่ (Static) เช่น แบบอะเรย์เชิงเส้น (Linear Array)

แบบดาว (Star) แบบแหวน (Ring) แบบเม็สซ์ (Mesh) แบบไฮเปอร์คิว (HyperCube) เป็นต้น ซึ่งมีงานวิจัยที่น่าสนใจดังนี้

ในปี ค.ศ. 1997 M. F. Ionescu และ K. E. Schauser [11] เสนองานวิจัยที่ได้ศึกษาการเรียงลำดับแบบไบโทนิค บนระบบที่ทันสมัย ซึ่งมีความสัมพันธ์กันแบบคอร์สเกรน (Coarse grained) ซึ่งประกอบด้วยจำนวนโหนดที่เหมาะสมและแต่ละโหนดมีประสิทธิภาพสูง ดังนั้นจึงต้องทำการกำหนดข้อมูลจำนวนมากไปให้หน่วยประมวลผลแต่ละตัว (Mapping) ผลที่ตามมาก็คือเวลาที่ใช้ในการติดต่อระหว่างหน่วยประมวลผลจะสูงทำให้เกิดแนวความคิดที่จะลดขั้นตอนของการติดต่อสื่อสารและลดเวลาที่ใช้ในการคำนวณให้น้อยที่สุด โดยเสนอขั้นตอนวิธีแบบฟาสเตอร์ (Faster Algorithm) ซึ่งได้ทำการทดลองบนระบบคอมพิวเตอร์ Meiko CS-2 64 โหนด

ในปี ค.ศ. 2001 Y. C. Kim และคณะ [14] เสนองานวิจัยที่แสดงการเพิ่มประสิทธิภาพของการติดต่อสื่อสารระหว่างหน่วยประมวลผล และเวลาที่ใช้ในการเปรียบเทียบ ซึ่งบ่อยครั้งที่หน่วยประมวลผลไม่มีความจำเป็นที่จะแลกเปลี่ยนข้อมูลกัน หรือต้องการแลกเปลี่ยนข้อมูลเพียงบางส่วนเท่านั้น ในงานวิจัยนี้จะเน้นที่การแลกเปลี่ยนข้อมูลเฉพาะส่วนที่ต้องใช้ในกระบวนการเรียงลำดับข้อมูลเท่านั้น โดยหลักการนี้จะช่วยลดเวลาในการติดต่อสื่อสารระหว่างหน่วยประมวลผลลงได้อย่างน้อย 20% บนระบบคอมพิวเตอร์ Cray T3E

2.3.2.6 การเรียงลำดับแบบขนานด้วยวิธีไบโทนิคบนระบบคลัสเตอร์

การเรียงลำดับแบบขนานด้วยวิธีไบโทนิคบนระบบคลัสเตอร์ ระบบนี้ใช้หน่วยความจำแบบกระจายเช่นกัน และมีโครงสร้างการติดต่อสื่อสารระหว่างหน่วยประมวลผล (Interconnection Network) ส่วนใหญ่เป็นแบบ สวิตซ์เน็ตเวิร์ก (Switch Network) ดังตัวอย่างงานวิจัยต่อไปนี้

ในปี ค.ศ. 1997 Helman, D. R. and JaJa, J. [9] ได้นำเสนอผลงานวิจัยบนระบบคลัสเตอร์แบบเอสเอ็มพีเอส (SMPs: Symmetric Multiprocessors) ซึ่งได้คิดค้นขั้นตอนวิธีที่เรียกว่า “SMP Algorithm” โดยขั้นตอนวิธีนี้ใช้หลักการในการผสมผสานระหว่างวิธี “Random Sampling” กับวิธี “Deterministic Sampling” ซึ่งงานวิจัยดังกล่าวพัฒนาด้วยภาษาซี (C Language) โดยใช้โปรแกรมโพสิก (POSIX)

ในปี ค.ศ. 2000 J. Brest และคณะ [4] ทำงานวิจัยร่วมกันโดยนำเสนอวิธีการติดต่อสื่อสารข้อมูลระหว่างงานย่อย (Subtask) ของโปรแกรมประยุกต์ และแสดงผลการทดลองขั้นตอนวิธีการเรียงลำดับแบบขนาน (Parallel Sorting Algorithms) ด้วยวิธีการเรียงลำดับแบบเร็ว (Quick Sort) บนระบบคลัสเตอร์

2.4 การวัดประสิทธิภาพของการประมวลผลแบบขนาน

การวัดประสิทธิภาพของการเรียงลำดับข้อมูลแบบขนานในงานวิจัยนี้จะประเมินผลจากเวลาที่ใช้ในการเรียงลำดับข้อมูล (Response Time) อัตราการเพิ่มขึ้นของความเร็ว (Speedup) และประสิทธิภาพ (Efficiency) ในการประมวลผลจริงบนระบบมัลติคอร์สามารถวัดเวลาที่ใช้ได้โดยการจับเวลาที่ใช้ในการเรียงลำดับ ทั้งการเรียงลำดับแบบอนุกรม (Sequential Sort) และการเรียงลำดับแบบขนาน (Parallel Sort) โดยจะทำการวัดเวลาที่ใช้ในการเรียงลำดับ ซึ่งไม่รวมเวลาที่ใช้ในการอ่านข้อมูลมาเก็บไว้ในหน่วยความจำและการตั้งค่าระบบ (Initialization) ของ MPI ซึ่งกระบวนการทั้งสองจะถูกทำครั้งเดียวในการเรียงลำดับข้อมูล

1. เวลาที่ใช้ในการประมวลผล (Response Time)

การวัดเวลาที่ใช้ในการประมวลผลสำหรับการเรียงลำดับแบบขนาน ในทางทฤษฎี (Theoretical Approach) เวลาที่ใช้ในการประมวลผลแบบขนานนี้ (Parallel Computation) สามารถคำนวณได้ดังสมการที่ 2.5 [19]

$$T_p = \frac{T_s}{P} \tag{2.5}$$

เมื่อ T_p คือ เวลาที่ใช้ในการประมวลผลแบบขนานด้วย P หน่วยประมวลผล

T_s คือ เวลาที่ใช้ในการประมวลผลแบบขนานด้วยหนึ่งหน่วยประมวลผล หรือเวลาที่ใช้ในการประมวลผลข้อมูลแบบอนุกรมนั้นเอง

P คือ จำนวนหน่วยประมวลผล (Processor) ที่ใช้ในระบบมัลติคอร์

ซึ่งกรณีนี้จะเรียกว่าเป็นกรณีอุดมคติ (Ideal Case) เพราะว่ามีการสมมติให้เวลาที่ใช้ในการติดต่อสื่อสารระหว่างหน่วยประมวลผลมีค่าเป็นศูนย์ แต่ในทางปฏิบัติ (Practical Approach) บนระบบมัลติคอร์ การวัดเวลาที่ใช้ในการประมวลผลแบบขนานจะวัดโดยการจับเวลา ตั้งแต่เริ่มประมวลผล จนกระทั่งสิ้นสุดการประมวลผล หรือได้คำตอบที่ต้องการ ซึ่งเวลาที่วัดได้ดังกล่าวจะรวมเวลาที่ใช้ในการติดต่อสื่อสารด้วย ดังนั้นถึงแม้ว่าจำนวนของหน่วยประมวลผล (P) ในระบบจะมีเพิ่มขึ้นมากๆ แต่เวลาจากการประมวลผลแบบขนานจะไม่ลดลงจนเข้าใกล้ศูนย์ เพราะสาเหตุมาจากเวลาที่ใช้ในการประมวลผลทั้งหมดประกอบด้วยเวลาที่ใช้ในการประมวลผลแบบขนานในแต่ละหน่วยประมวลผล และเวลาที่ใช้ในการติดต่อสื่อสารระหว่างหน่วยประมวลผลซึ่งปกติจะไม่เป็นศูนย์ ดังกรณีอุดมคติ โดยจะแสดงได้ดังสมการที่ 2.6

$$T_p = t_{computation} + t_{communication} \tag{2.6}$$

เมื่อ T_p คือ เวลาที่ใช้ในการประมวลผลแบบขนานทั้งหมดด้วย P หน่วย

$t_{computation}$ คือ เวลาที่ใช้ประมวลผลแบบขนาน (เช่น การเรียงลำดับข้อมูลด้วยภาระงาน w=N/P) ซึ่งไม่รวมเวลาที่ใช้ในการติดต่อสื่อสารระหว่างหน่วยประมวลผล

$t_{communicaton}$ คือ เวลาที่ใช้ในการสื่อสารระหว่างหน่วยประมวลผล (ถ้ามี)

ดังนั้นในทางปฏิบัติเราสามารถเพิ่มหน่วยประมวลผลเข้าไปในระบบมากขึ้น และเวลาจะลดลงช่วงหนึ่งเท่านั้น เพราะเวลาที่ใช้ในการประมวลผลยังคงมากกว่าเวลาที่ใช้ในการติดต่อสื่อสารระหว่างหน่วยประมวลผล แต่ ณ จุดหนึ่งเมื่อเพิ่มหน่วยประมวลผลเข้าไปอีก เวลาจะไม่ลดลงแต่อาจจะเพิ่มขึ้นเนื่องจากเวลาที่ใช้ในการสื่อสารระหว่างหน่วยประมวลผล $t_{communicaton}$ มากขึ้นและมีค่ามากกว่า เวลาที่ใช้ในการประมวลผล

2. อัตราการเพิ่มของความเร็ว (Speedup)

นอกจากการวัดเวลาที่ใช้ในการประมวลผลแล้วยังสามารถวัดอัตราการเพิ่มของความเร็ว (Speedup) ของการประมวลผลแบบขนานได้ ทั้งในทางทฤษฎีและทางปฏิบัติจากสมการที่ 2.7 [19]

$$S_p = \frac{T_s}{T_p} \leq P \qquad (2.7)$$

เมื่อ S_p คือ อัตราการเพิ่มของความเร็วที่ใช้ในประมวลผลแบบขนาน โดยใช้ P หน่วยประมวลผล ซึ่งมีค่าสูงสุดในอุดมคติเท่ากับ P

T_s คือ เวลาที่ใช้ในการประมวลผลแบบขนานด้วยหนึ่งหน่วยประมวลผล

T_p คือ เวลาที่ใช้ในการประมวลผลแบบขนานด้วย P หน่วยประมวลผล

3. ประสิทธิภาพ (Efficiency)

นอกจากการวัดเวลาที่ใช้ในการประมวลผล และอัตราการเพิ่มขึ้นของความเร็วแล้ว ยังสามารถวัดประสิทธิภาพของการประมวลผลแบบขนานเป็นค่าที่อยู่ในช่วง 0-1 (หรือ 0-100%) ทุกค่าของหน่วยประมวลผล โดยใช้สมการที่ 2.8 ดังนี้

$$E_p = \frac{S_p}{P} \leq 1 \qquad (2.8)$$

เมื่อ E_p คือ ประสิทธิภาพของการประมวลผลแบบขนานมีค่าในอุดมคติเท่ากับ 1

S_p คือ อัตราการเพิ่มของความเร็วที่คำนวณได้จากสมการที่ 2.7

P คือ จำนวนหน่วยประมวลผลที่ใช้บนระบบมัลติคอร์

บทที่ 3
การติดต่อสื่อสารที่มีประสิทธิภาพสูงด้วยวิธีไดนามิค

งานวิจัยในบทนี้กล่าวถึงการติดต่อสื่อสารระหว่างหน่วยประมวลผล (Processors: P) ที่มีประสิทธิภาพสูงด้วยวิธีไดนามิค (Dynamic Communication) เพื่อใช้สำหรับงานด้านการเรียงลำดับข้อมูล จากงานวิจัยที่ได้กล่าวไว้ในบทที่ผ่านมานั้น ตั้งแต่อดีตจนปัจจุบันการติดต่อสื่อสารระหว่างหน่วยประมวลผลเป็นแบบสแตทิค (Static Communication) ซึ่งวิธีการดังกล่าวมีจำนวนรอบในการทำงานคือ $(1+\log_2 P)(\log_2 P)/2$ รอบ เนื่องจากการติดต่อสื่อสารระหว่างหน่วยประมวลผลแบบสแตทิค กำหนดรูปแบบการติดต่อสื่อสารไว้อย่างชัดเจนดังรูปที่ 2.12(ก) ในบทที่ 2 ซึ่งการทำงานดังกล่าวเมื่อเริ่มกระบวนการเรียงลำดับข้อมูลแล้วต้องทำครบทุกรอบเพื่อแลกเปลี่ยนข้อมูลที่เหมาะสมในแต่ละหน่วยประมวลผล จึงจะถือว่ากระบวนการนั้นเสร็จสิ้นสมบูรณ์ แต่ในบางครั้งการเรียงลำดับข้อมูลอาจต้องการเพียง 1 รอบ หรือ 2 หรือ 3 หรือ,…, หรือ $(\log_2 P\ (\log_2 P+1))/2$ รอบในการทำงาน ดังนั้นวิธีที่ผู้วิจัยได้นำเสนอในวิทยานิพนธ์นี้คือ “การติดต่อสื่อสารที่มีประสิทธิภาพสูงด้วยวิธีไดนามิค” ซึ่งสามารถเรียงลำดับข้อมูลใช้เพียง 1 รอบ (กรณีที่ดีที่สุด) หรือ 2 หรือ 3 หรือ, …, หรือ $\log_2 P$ รอบ (กรณีที่ช้าที่สุด) โดยแต่ละรอบสามารถจบการทำงานได้หากข้อมูลเรียงได้อย่างถูกต้องเหมาะสมแล้วตามลำดับของหน่วยประมวลผล (List Ranking) โดยเรียงตามค่าของตัวแทนชุดข้อมูล เช่น ค่าน้อยสุด (min) ในแต่ละกลุ่ม หรือค่ามิดพอยท์ (midpoint) ที่คำนวณจาก (min+max)/2 เป็นต้น จากวิธีที่ได้นำเสนอทำให้จำนวนรอบในการทำงานลดลง

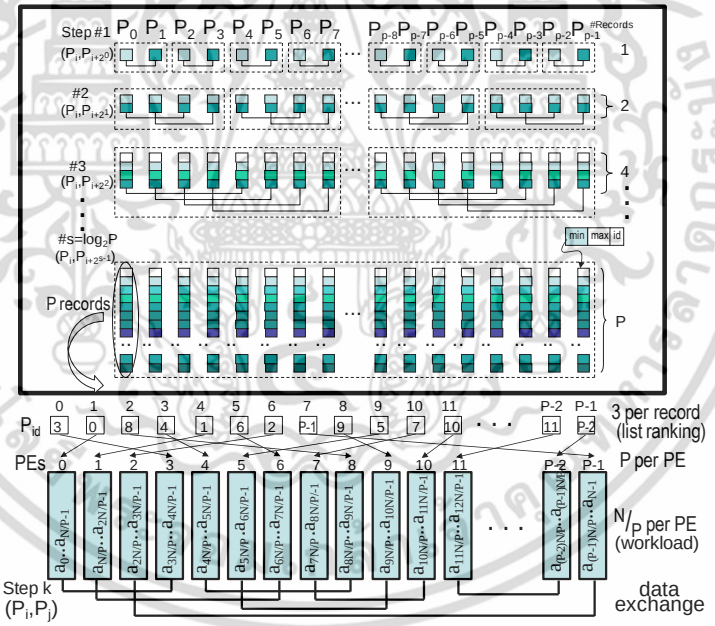
ผู้วิจัยได้นำเสนอ “การติดต่อสื่อสารที่มีประสิทธิภาพสูงด้วยวิธีไดนามิค” เป็นการเลือกคู่หน่วยประมวลผลที่เหมาะสมในการแลกเปลี่ยนข้อมูลเพื่อเรียงลำดับ โดยแบ่งวิธีการคำนวณหาคู่หน่วยประมวลผลที่เหมาะสม (จากข้อมูลที่เป็นค่าของตัวแทนกลุ่มทั้งหมด) แบ่งเป็น 2 แบบ ดังนี้

1) การกระจายข้อมูลที่มีประสิทธิภาพ (Efficient Broadcast Table: B-Table) เป็นวิธีที่ทุกๆ หน่วยประมวลผลส่งค่าของตัวแทนชุดข้อมูลให้หน่วยประมวลผลที่ติดต่อด้วยทุกหน่วย ซึ่งให้ค่าความซับซ้อนด้านเวลาคือ O(P) เมื่อ P คือ จำนวนหน่วยประมวลผล
2) การรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูง (More Efficient Point-to-Point Table: P-Table) แต่ละหน่วยประมวลผลส่งค่าของตัวแทนชุดข้อมูลให้กับหน่วยประมวลผลที่ติดต่อสื่อสารด้วยโดยตรงไปพร้อมๆ กัน (Parallel Point-to-Point) จากนั้นประยุกต์ใช้การเรียงลำดับแบบขนานด้วยวิธีไบโทนิคตามค่าของตัวแทนชุดข้อมูล ส่งผลให้ประสิทธิภาพในการทำงานสูงกว่าแบบแรก ซึ่งพิจารณาได้จากค่าความซับซ้อนด้านเวลาคือ $O(\log_2 P)^2$ เมื่อ P คือ จำนวนหน่วยประมวลผล

ทั้งสองวิธีดังกล่าวที่นำเสนอพัฒนาขึ้นโดยใช้เทคนิคการโปรแกรมแบบขนานด้วยมาตรฐานภาษาเอ็มพีไอ เพื่อใช้เป็นเครื่องมือในการทดลองเปรียบเทียบประสิทธิภาพที่แตกต่างกัน โดยวัดผลขั้นต้นจากเวลาที่ใช้ในการประมวลผล (Response Time) จากนั้นสามารถแสดงอัตราการเพิ่มของความเร็ว (Speedup) และประสิทธิภาพ (Efficiency) ของการเรียงลำดับข้อมูล

3.1 การติดต่อสื่อสารที่มีประสิทธิภาพสูงด้วยวิธีไดนามิค โดยใช้การกระจายข้อมูลที่มีประสิทธิภาพ

การติดต่อสื่อสารที่มีประสิทธิภาพสูงด้วยวิธีไดนามิค ที่ใช้การกระจายข้อมูลที่มีประสิทธิภาพนี้ ทุกๆ หน่วยประมวลผลส่งตัวแทนชุดข้อมูลให้หน่วยประมวลผลที่ติดต่อด้วยทุกตัว ข้อมูลที่ส่งคือ ค่าน้อย (Minimal Value: min), ค่ามาก (Maximal Value: max) และดัชนีของหน่วยประมวลผลแต่ละหน่วย (Processor Id: id) ดังรูปที่ 3.1 แสดงการติดต่อสื่อสารด้วยการกระจายข้อมูลแบบมีประสิทธิภาพ ให้ค่าความซับซ้อนด้านเวลาคือ O(P) เมื่อ P คือ จำนวนหน่วยประมวลผล



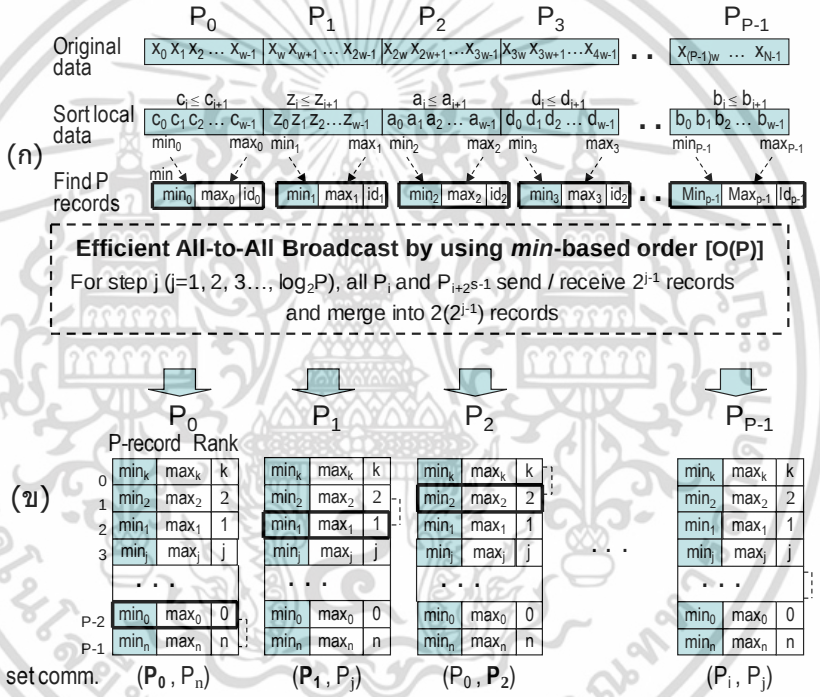
รูปที่ 3.1 แสดงโครงสร้างตารางกระจายข้อมูลแบบมีประสิทธิภาพ (Broadcast Table: B-Table)

การติดต่อสื่อสารที่มีประสิทธิภาพสูงด้วยวิธีไดนามิค ที่ใช้การกระจายข้อมูลแบบมีประสิทธิภาพ (B-Table) นี้สามารถแบ่งการเลือกคู่หน่วยประมวลผลได้เป็น 2 วิธี กล่าวคือ

1) ประยุกต์ใช้ในการเรียงลำดับข้อมูลแบบดีซีอีเอส (DCES: Dynamic Communication Efficient Sort) [25] ด้วยการเลือกคู่จากค่าน้อย (Min-based)
2) ประยุกต์ใช้ในการเรียงลำดับข้อมูลแบบดีซีพีเอส (DCPS: Dynamic Communication Parallel Sort) [26] ด้วยการเลือกคู่จากค่ามิดพอยท์ (Midpoint-based)

3.1.1 การเรียงลำดับข้อมูลแบบดีซีอีเอส

วิธีการเรียงลำดับแบบดีซีอีเอส **(DCES: Dynamic Communication Efficient Sort)** [25] นี้จะเรียงลำดับตัวแทนชุดข้อมูลจากน้อยไปมากภายในตารางการกระจายข้อมูลที่มีประสิทธิภาพตามคอลัมน์ค่าน้อย เมื่อได้รับตัวแทนชุดข้อมูลจากหน่วยประมวลผลครบทุกหน่วยแล้ว โดยประยุกต์ใช้การเรียงลำดับแบบเร็วสำหรับข้อมูลย่อยในแต่ละหน่วยประมวลผล จากรูปที่ 3.2 (ก) แสดงการส่งข้อมูลค่าน้อย, ค่ามาก และดัชนีของหน่วยประมวลผล ให้แต่ละหน่วยประมวลผลเพื่อสร้างตารางการกระจายข้อมูลแบบมีประสิทธิภาพ ในรูปที่ 3.2 (ข) แสดงตารางการกระจายข้อมูลแบบมีประสิทธิภาพที่สร้างขึ้นในแต่ละหน่วยประมวลผลโดยเรียงลำดับข้อมูลจากน้อยไปมากตามคอลัมน์ค่าน้อย

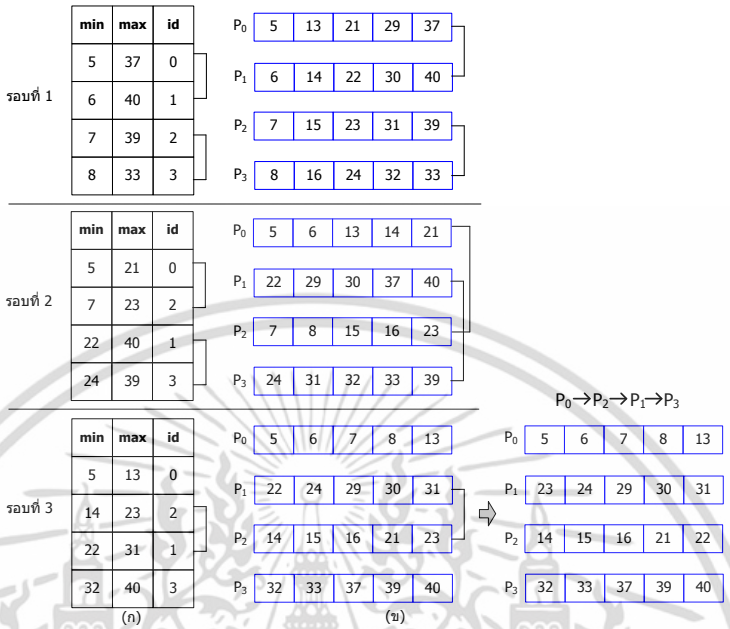


รูปที่ 3.2 (ก) แสดงการแลกเปลี่ยนข้อมูล (ข) ตาราง B-Table โดยเรียงจากค่าน้อย (Min)

<u>ตัวอย่างที่ 3.1</u> **แสดงการเรียงลำดับข้อมูลแบบดีซีอีเอส (DCES)**

สมมติให้ข้อมูลนำเข้า (N=20) 5, 21, 13, 37, 29, 14, 6, 30, 22, 40, 23, 7, 39, 15, 31, 8, 16, 24, 33, 32 ซึ่งจำนวนหน่วยประมวลผล (P=4) โดยจำนวนหน่วยประมวลผล (P) น้อยกว่าจำนวนข้อมูล(N) แบ่งข้อมูลให้แต่ละหน่วยประมวลผลเท่าๆ กัน (N/P=5) และเรียงลำดับข้อมูลย่อย คือ P_0 {5, 13, 21, 29, 37}, P_1 {6, 14, 22, 30, 40}, P_2 {7, 15, 23, 31, 39} และ P_3 {8, 16, 24, 32, 33} จากนั้นทุกๆ หน่วยประมวลผลส่งข้อมูลค่าน้อย, ค่ามาก และดัชนีของหน่วยประมวลผลให้กับหน่วยประมวลผลอื่นๆ ทุกหน่วย ดังแสดงในรูปที่ 3.3 (ก) เพื่อใช้สร้างตารางการกระจายข้อมูลแบบมีประสิทธิภาพ โดยเรียงข้อมูลจากน้อยไปมากตามคอลัมน์ค่าน้อย จากนั้นพิจารณาคู่ของหน่วยประมวลผลที่เหมาะสมในการเรียงลำดับตัวแทนชุดข้อมูลต่อไป รูปที่ 3.3(ข) แสดงตัวอย่างตัวแทน

ชุดข้อมูลในแต่ละหน่วยประมวลผล พร้อมทั้งแสดงคู่ของหน่วยประมวลผลที่ติดต่อสื่อสารกันในแต่ละรอบการทำงาน ในรอบแรก P_0 คู่กับ P_1 และ P_2 คู่กับ P_3 ($P_0 \rightarrow P_1$, $P_2 \rightarrow P_3$) รอบสอง P_0 คู่กับ P_2 และ P_1 คู่กับ P_3 ($P_0 \rightarrow P_2$, $P_1 \rightarrow P_3$) ส่วนรอบสุดท้าย P_2 คู่กับ P_1 ($P_2 \rightarrow P_1$)

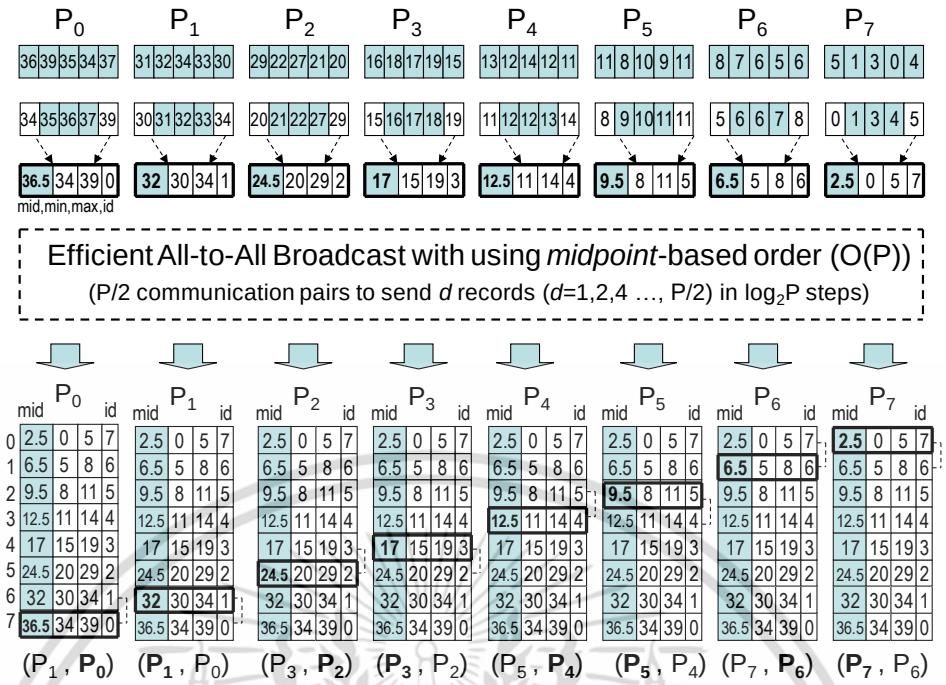


รูปที่ 3.3 (ก) แสดงการสร้าง B-Table ด้วยค่าน้อย (ข) ตัวอย่างข้อมูลในแต่ละหน่วยประมวลผล

เนื่องจากการเรียงลำดับข้อมูลแบบดีซีอีเอส [25] ด้วยการคำนวณจากคอลัมน์ค่าน้อยยังไม่ครอบคลุมข้อมูลบางลักษณะ ซึ่งในบางครั้งใช้จำนวนรอบในการทำงานเท่ากับ $(1+\log_2 4)(\log_2 4)/2 = 3$ รอบ เพื่อเรียงลำดับข้อมูลชุดดังกล่าวซึ่งใช้เวลาเท่ากับวิธีการเดิม (CEBS) ที่ได้กล่าวไว้ในบทที่ 2 ดังนั้นผู้วิจัยได้นำเสนอวิธีการเรียงลำดับแบบดีซีพีเอสที่ปรับจากดีซีอีเอส โดยการสร้างตารางการกระจายข้อมูลแบบมีประสิทธิภาพด้วยค่ามิดพอยท์ เพื่อจัดการกับข้อจำกัดดังกล่าว อีกทั้งสามารถเรียงลำดับข้อมูลได้อย่างมีประสิทธิภาพเพิ่มขึ้น ในหัวข้อถัดไป

3.1.2 วิธีการเรียงลำดับข้อมูลแบบดีซีพีเอส

การเรียงลำดับแบบดีซีพีเอส **(DCPS: Dynamic Communication Parallel Sort)** นี้ เรียงลำดับตัวแทนชุดข้อมูลจากน้อยไปมากภายในตารางการกระจายข้อมูลแบบมีประสิทธิภาพตามคอลัมน์ค่ามิดพอยท์ หลังจากได้รับตัวแทนชุดข้อมูลที่จำเป็นจากหน่วยประมวลผลครบทุกหน่วยแล้ว ซึ่งคำนวณค่ามิดพอยท์ได้จาก (min+max)/2 โดยประยุกต์ใช้การเรียงลำดับแบบเร็วสำหรับข้อมูลย่อยในแต่ละหน่วยประมวลผล จากรูปที่ 3.4 (ก) แสดงการส่งข้อมูลค่ามิดพอยท์, ค่าน้อย, ค่ามาก และดัชนีของหน่วยประมวลผล ในรูปที่ 3.4 (ข) แสดงตารางการกระจายข้อมูลที่มีประสิทธิภาพ ที่สร้างขึ้นในแต่ละหน่วยประมวลผล โดยเรียงลำดับข้อมูลจากน้อยไปมากตามคอลัมน์ค่ามิดพอยท์

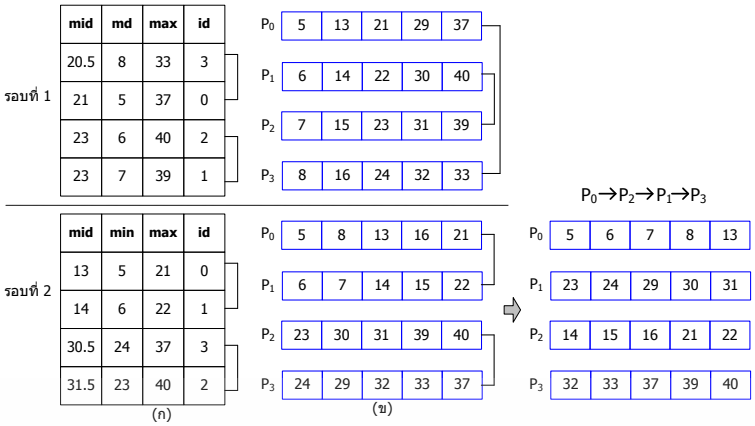


รูปที่ 3.4 (ก) แสดงการแลกเปลี่ยนข้อมูล (ข) B-Table โดยเรียงจากค่ามิดพอยท์ (Midpoint)

หมายเหตุ ค่ามิดพอยท์เป็นคีย์ ที่ใช้ในหัวข้อนี้เป็นค่ามิดพอยท์แบบคร่าวๆ ที่นำค่าน้อยและค่ามากของชุดข้อมูลในแต่ละหน่วยประมวลผลมารวมกันแล้วหารสอง (min+max)/2 ที่สามารถจัดลำดับของหน่วยประมวลผล (List Ranking) ได้ดีกว่าการใช้ค่าน้อย (min) เป็นคีย์ โดยเฉพาะในกรณีที่ข้อมูลมีลักษณะข้อมูลแบบเอนเอียงที่มีค่านอกกลุ่มแฝงอยู่ (Skew Data with Outliner) ซึ่งเป็นกรณีที่ข้อมูลไม่ปกติใช้จำนวนรอบมากที่สุด

ตัวอย่างที่ 3.2 **แสดงการเรียงลำดับแบบดีซีพีเอส (DCPS)**

สมมติให้ข้อมูลนำเข้าเดียวกับตัวอย่างที่ 3.1 โดย P_0 {5, 13, 21, 29, 37}, P_1 {6, 14, 22, 30, 40}, P_2 {7, 15, 23, 31, 39} และ P_3 {8, 16, 24, 32, 33} ส่วนค่ามิดพอยท์คำนวณได้จาก (min + max)/2 โดย P_0 (5+37)/2 = 21, P_1 (6+40)/2 = 23, P_2 (7+39)/2 = 23 และ P_3 (8+33)/2 = 20.5 ตามลำดับ จากรูปที่ 3.5 (ก) หลังจากได้ค่ามิดพอยท์แล้ว เรียงลำดับตัวแทนชุดข้อมูลจากน้อยไปมากตามคอลัมน์ค่ามิดพอยท์ ได้คู่ของหน่วยประมวลผลในรอบแรก P_1 คู่กับ P_2 และ P_0 คู่กับ P_3 ($P_1 \rightarrow P_2$, $P_0 \rightarrow P_3$) และในรอบสุดท้าย P_0 คู่กับ P_1 และ P_2 คู่กับ P_3 ($P_0 \rightarrow P_1$, $P_2 \rightarrow P_3$) (จากตัวอย่างเดียวกับ 3.1 ใช้เพียง 2 รอบเท่านั้น) รูปที่ 3.5 (ข) แสดงข้อมูลและการติดต่อสื่อสารในแต่ละหน่วยประมวลผลจากที่คำนวณได้ในตารางกระจายข้อมูลแบบมีประสิทธิภาพด้วยค่ามิดพอยท์



รูปที่ 3.5 (ก) แสดงการสร้าง B-Table ด้วยค่ามิดพอยท์ (ข)ตัวอย่างข้อมูลในแต่ละหน่วยประมวลผล

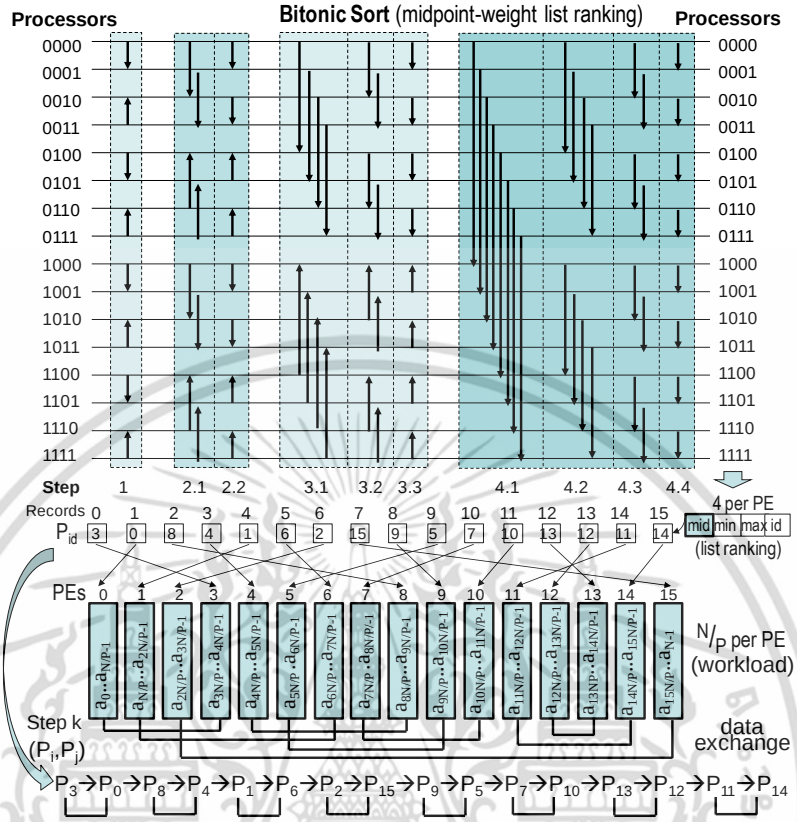
จากตัวอย่างเดียวกันการเรียงลำดับแบบดีซีพีเอส ที่ใช้ตารางการกระจายข้อมูลแบบมีประสิทธิภาพด้วยค่ามิดพอยท์ มีจำนวนรอบในการเรียงลำดับข้อมูลคือ $(\log_2 P+1) = (\log_2 4+1) = 3$ รอบ (กรณีที่แย่ที่สุด) หากพิจารณาจากชุดข้อมูลใช้เพียง 2 รอบในการทำงาน ดังนั้นการเลือกคู่หน่วยประมวลผลโดยพิจารณาตามลักษณะของข้อมูลจริงส่งผลให้จำนวนรอบในการทำงานลดลง

อย่างไรก็ตามการเรียงลำดับข้อมูลแบบดีซีอีเอส และแบบดีซีพีเอส ที่ใช้ตารางกระจายข้อมูลแบบมีประสิทธิภาพ ให้ค่าความซับซ้อนด้านเวลา O(P) ผู้วิจัยได้นำเสนอการเรียงลำดับข้อมูลแบบโอบีเอส (OBS: Optimized Bitonic Sort) ที่ใช้ตารางรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูงขึ้นเพื่อปรับปรุงประสิทธิภาพในการเรียงลำดับตัวแทนชุดข้อมูลภายในตารางดังกล่าวโดยประยุกต์ใช้การเรียงลำดับแบบขนานด้วยวิธีไบโทนิคสำหรับค่าของตัวแทนชุดข้อมูล P ค่า และใช้เทคนิคการติดต่อสื่อสารแบบ “Parallel Point-to-Point Communication” ส่งผลให้ค่าความซับซ้อนด้านเวลาคือ $O(\log_2 P)^2$ ซึ่งน้อยกว่าการกระจายข้อมูลแบบมีประสิทธิภาพ คือ O(P)

3.2 การติดต่อสื่อสารที่มีประสิทธิภาพสูงด้วยวิธีไดนามิค ด้วยการรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูงและการเรียงลำดับข้อมูลแบบโอบีเอส

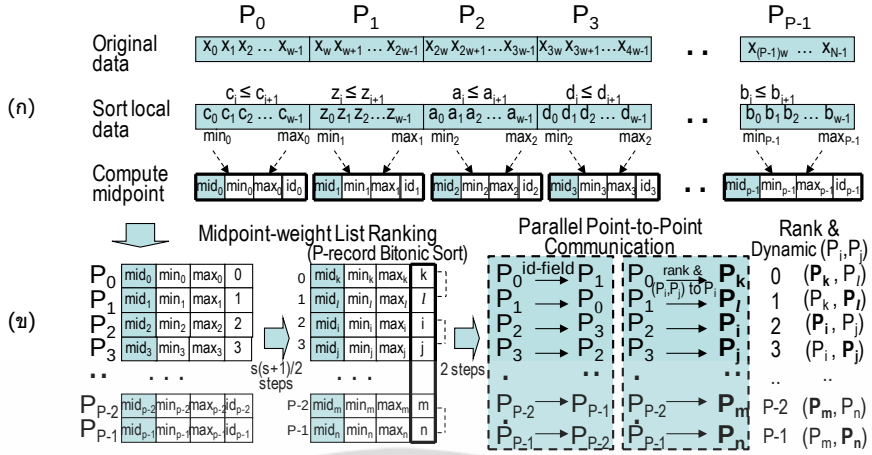
การติดต่อสื่อสารที่มีประสิทธิภาพสูงด้วยวิธีไดนามิค ที่ใช้การรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูงและการเรียงลำดับข้อมูลแบบโอบีเอส (More Efficient Point-to-Point Table: P-Table and Optimized Bitonic Sort: OBS) [26] เป็นการเลือกคู่หน่วยประมวลผลที่มีประสิทธิภาพสูงกว่าการกระจายข้อมูลแบบมีประสิทธิภาพ ซึ่งให้ค่าความซับซ้อนด้านเวลาคือ $O(\log_2 P)^2$ เมื่อ P คือ จำนวนหน่วยประมวลผล โดยนำวิธีการเรียงลำดับแบบขนานด้วยวิธีไบโทนิคมาประยุกต์ใช้ในการเรียงลำดับตัวแทนชุดข้อมูลด้วยค่ามิดพอยท์ หลังเรียงลำดับตัวแทนชุดข้อมูลแล้ว ทำการติดต่อสื่อสารเพื่อเลือกคู่ให้แต่ละหน่วยประมวลผล โดยประยุกต์ใช้เทคนิคการติดต่อสื่อสารแบบ “Parallel Point-to-Point Communication” ด้วยค่าความซับซ้อนด้านเวลาเพียง O(1) เพื่อใช้ในการ

ติดต่อสื่อสารต่อไป โดยผู้วิจัยได้ประยุกต์ใช้การรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูงกับการเรียงลำดับข้อมูลแบบโอบีเอส ซึ่งแสดงในรูปที่ 3.6



รูปที่ 3.6 แสดงการประยุกต์ใช้วิธีการเรียงลำดับแบบไบโทนิค พร้อมด้วยค่ามิดพอยท์

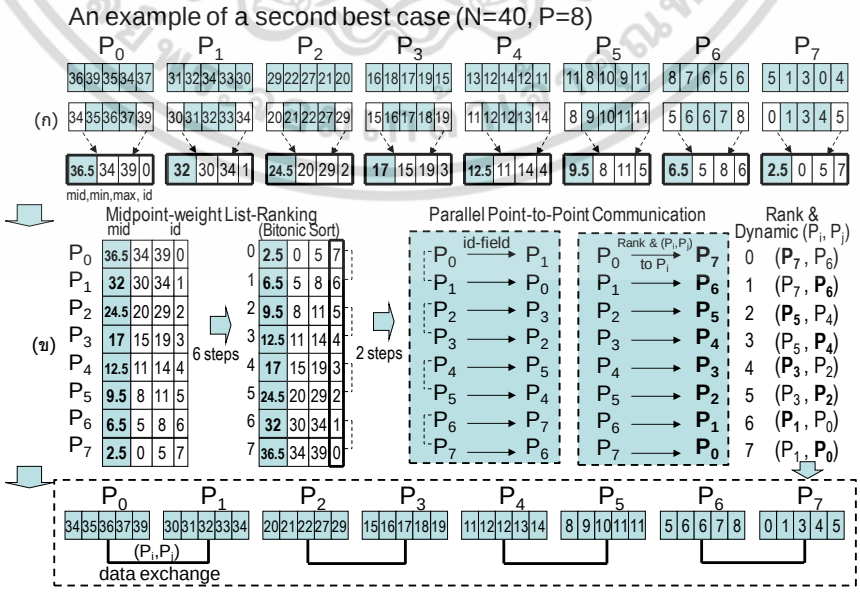
ในรูปที่ 3.7 (ก) แสดงตัวแทนชุดข้อมูลและการส่งข้อมูลค่ามิดพอยท์, ค่าน้อย, ค่ามาก และดัชนีของหน่วยประมวลผล ในแต่ละหน่วยประมวลผลพร้อมๆ กัน และรูปที่ 3.7 (ข) แสดงการเรียงลำดับตัวแทนชุดข้อมูลแบบขนานด้วยวิธีไบโทนิคภายในตารางการรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูง ซึ่งจำนวนรอบในการเรียงลำดับนั้นคำนวณได้จาก $(\log_2 P(\log_2 P+1))/2$ รอบ จากนั้นเลือกคู่ให้แต่ละหน่วยประมวลผลด้วยเทคนิคการติดต่อสื่อสารแบบ “Parallel Point-to-Point Communication” จากรูปที่ 3.7 (ข) ด้านขวา $P_0, P_1, \ldots, P_{p-1}$ ติดต่อหน่วยประมวลผลอื่นไปพร้อมๆ กันด้วยการเลือกคู่แบบคู่คี่ โดยหน่วยประมวลผลที่มีเลขดัชนีคู่บวกเพิ่มหนึ่งหน่วย ส่วนหน่วยประมวลผลที่มีเลขดัชนีคี่ลดลงหนึ่งหน่วยประมวลผลจะได้ว่า $(P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, \ldots, P_{p-1} \leftrightarrow P_{p-2})$ คู่ของหน่วยประมวลผลที่ติดต่อกันโดยตรงจากคอลัมน์ดัชนีของหน่วยประมวลผล



รูปที่ 3.7 ก) แสดงข้อมูลค่ามิดพอยท์, ค่าน้อย, ค่ามาก และดัชนี ข) P-Table โดยเรียงจากค่ามิดพอยท์

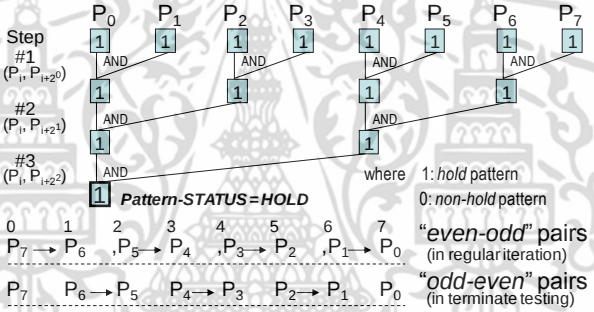
ตัวอย่างที่ 3.3 **แสดงการเรียงลำดับแบบโอบีเอส (OBS)**

สมมติให้ข้อมูลนำเข้า (N=40) คือ 34, 35, 36, 37, 39, 30, 31, 32, 33, 34, 20, 21, 22, 27, 29, 15, 16, 17, 18, 19, 11, 12, 12, 13, 14, 8, 9, 10, 11, 11, 5, 6, 6, 7, 8, 0, 1, 3, 4, 5 ซึ่งจำนวนหน่วยประมวล (P=8) แบ่งข้อมูลให้แต่ละหน่วยประมวลผล (N/P=5) ดังแสดงในรูปที่ 3.8 (ก) เรียงลำดับข้อมูลย่อยในแต่ละหน่วยประมวลผลด้วยการเรียงลำดับแบบเร็ว จากนั้นส่งข้อมูลค่าน้อย, ค่ามาก, ดัชนีหน่วยประมวลผล และค่ามิดพอยท์ หน่วยประมวลผลที่ติดต่อด้วยไปพร้อมๆ กัน ส่วนรูปที่ 3.8 (ข) แสดงการเรียงลำดับแบบขนานด้วยวิธีไบโทนิคโดยเรียงตามคอลัมน์ค่ามิดพอยท์ ซึ่งมีจำนวนรอบในการทำงานคือ $(\log_2 P(\log_2 P+1))/2$ รอบ คำนวณได้ดังนี้ $(3(3+1))/2 = 6$ รอบ จากนั้นเลือกคู่หน่วยประมวลผลในแต่ละหน่วยด้วยเทคนิคการติดต่อสื่อสารแบบ “Parallel Point-to-Point Communication” 2 ครั้ง โดยครั้งแรกจัดคู่ (P_i, P_j) จาก id-field ที่อยู่ในตำแหน่งคู่คี่ และครั้งสองติดต่อไปยัง P_i เพื่อให้ทราบค่า rank และคู่ (P_i, P_j)



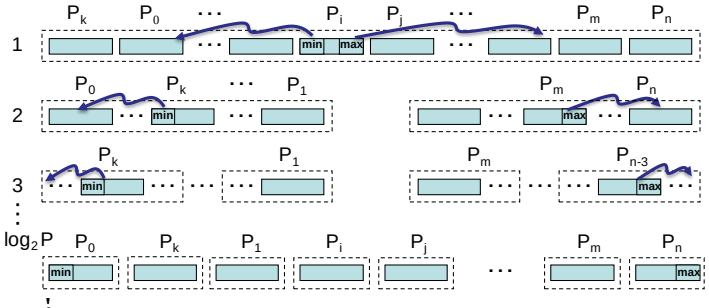
รูปที่ 3.8 แสดงตัวอย่างการสร้างตารางการรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูงขึ้น (P-Table)

หลังจากทุกหน่วยประมวลผลเรียงลำดับข้อมูลเสร็จสิ้นสมบูรณ์แล้ว ขั้นสุดท้ายก่อนจบกระบวนการเรียงลำดับคือการตรวจสอบสถานะของหน่วยประมวลผลว่าเรียงลำดับได้อย่างถูกต้องครบถ้วน ซึ่งเป็นขั้นตอนที่สำคัญและจำเป็นสำหรับการติดต่อสื่อสารแบบไดนามิค เพื่อยืนยันว่าข้อมูลได้ถูกจัดเรียงอย่างถูกต้องเหมาะสมแล้ว จากรูปที่ 3.9 แสดงตัวอย่างการตรวจสอบสถานะเพื่อจบกระบวนการเรียงลำดับข้อมูลสำหรับวิธีการติดต่อสื่อสารที่มีประสิทธิภาพด้วยวิธีไดนามิค โดยทั้ง 8 หน่วยประมวลผลตรวจสอบสถานะ หากข้อมูลเรียงได้อย่างถูกต้องสมบูรณ์สถานะของรูปแบบเท่ากับคงที่ (pattern-STATUS=HOLD:1) หากต้องเรียงลำดับต่อสถานะของรูปแบบเท่ากับไม่คงที่ (pattern-STATUS=non-HOLD:0) จากรูปแสดงให้เห็นว่าการติดต่อสื่อสารระหว่างหน่วยประมวลผลในรอบการทำงานทั่วไปเป็นลักษณะแบบคู่คี่ที่เรียงตามค่ามิดพอยท์ หรือ ($P_0 \rightarrow P_1$, $P_2 \rightarrow P_3$, $P_4 \rightarrow P_5$, $P_6 \rightarrow P_7$) สำหรับการตรวจสอบสถานะเพื่อสิ้นสุดกระบวนการเรียงลำดับข้อมูลการติดต่อสื่อสารเป็นแบบคี่คู่ หรือ ($P_1 \rightarrow P_2$, $P_3 \rightarrow P_4$, $P_5 \rightarrow P_6$) เมื่อตรวจสอบสถานะเสร็จสิ้นทุกๆ หน่วยประมวลผลติดต่อถึงกันครบทุกหน่วยดังนี้ ($P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow P_6 \rightarrow P_7$)



รูปที่ 3.9 แสดงการตรวจสอบสถานะของ Pattern และ List ด้วยฟังก์ชัน Parallel AND

สำหรับงานวิจัยที่ได้นำเสนอในบทนี้เน้นเรื่องประสิทธิภาพของการติดต่อสื่อสารแบบไดนามิค ดังตารางที่ 3.1 แสดงการเปรียบเทียบจำนวนรอบการทำงานของการติดต่อสื่อสารแบบไดนามิค ทั้ง 3 แบบ คือ แบบดีซีอีเอส, แบบดีซีพีเอส และแบบโอบีเอส โดยการเรียงลำดับสองแบบหลังได้นำเทคนิค “Divide-and-Conquer” มาประยุกต์ใช้ควบคู่กับการเลือกคู่หน่วยประมวลผลด้วยการเรียงตามค่ามิดพอยท์ ดังรูปที่ 3.10 แสดงเทคนิคการทำงานของ “Divide-and-Conquer” ส่งผลให้กรณีที่ช้าที่สุด (เช่น ในกรณีที่มีค่าต่างกันมากๆ min และ max แฝงอยู่ในกลุ่มเดียวกันในขั้นแรก) จะมีรอบการทำงานเพียง $\log_2 P$ รอบ (เพื่อทำการย้ายค่า min และค่า max ไปไว้ในกลุ่มที่เหมาะสม)



รูปที่ 3.10 แสดงเทคนิคการทำงานของ “Divide-and-Conquer”

ตารางที่ 3.1 แสดงการเปรียบเทียบจำนวนรอบการทำงานของการติดต่อสื่อสารแบบไดนามิค (Dynamic) ทั้ง 3 แบบ

ชนิดของการเรียงลำดับ	การติดต่อสื่อสาร	ดีที่สุด	แย่ที่สุด	Update List
การเรียงแบบดีซีอีเอส (DCES)	ไดนามิค	1	S(S+1)/2	$2(P-1),P=2^S$
การเรียงแบบดีซีพีเอส (DCPS)	ไดนามิค	1	S	$2(P-1),P=2^S$
การเรียงแบบโอบีเอส (OBS)	**ไดนามิค**	1	S	S(S+1)/2

สำหรับตารางที่ 3.2 แสดงการเปรียบเทียบค่าความซับซ้อนด้านเวลาและพื้นที่ (Time & Space Complexity) สำหรับการติดต่อสื่อสารแบบไดนามิค โดยการเรียงลำดับข้อมูลแบบดีซีอีเอส และแบบดีซีพีเอส ใช้ตารางการกระจายข้อมูลแบบมีประสิทธิภาพ (B-Table) ส่งผลดีกว่าแบบการกระจายข้อมูลแบบดั้งเดิม (All-to-All Broadcast) ในส่วนของการกระจายและการเรียงลำดับเฉพาะตัวแทนชุดข้อมูล ส่วนการเรียงลำดับแบบโอบีเอส ใช้การเรียงลำดับตัวแทนชุดข้อมูลแบบไดนามิคและใช้การรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูงขึ้น (P-Table) ซึ่งให้ผลดีที่สุด โดยไม่ต้องมีการกระจายข้อมูลเหมือนสองแบบแรก การเรียงลำดับข้อมูลดังกล่าวใช้เวลาเพียง $O(\log_2 P)^2$ เนื่องจากประยุกต์ใช้การเรียงลำดับแบบขนานด้วยวิธีไบโทนิคสำหรับเรียงค่ามิดพอยท์ ส่วนการค้นหาและพื้นที่ใช้งานลดลงด้วยเช่นกัน

ตารางที่ 3.2 แสดงการเปรียบเทียบความซับซ้อนด้านเวลาและพื้นที่ (Time & Space Complexity) สำหรับการติดต่อสื่อสารแบบไดนามิค (Dynamic Communication)

ชนิดของการเรียงลำดับ	การกระจาย	การเรียงลำดับ	การค้นหา	พื้นที่ใช้งาน
การกระจายข้อมูลแบบดั้งเดิม All-to-All Broadcast และเรียงค่า	$O(P \log_2 P)$	$O(P \log_2 P)$	$O(\log_2 P)$	P x P
การกระจายข้อมูลแบบมีประสิทธิภาพ Efficient Broadcast หรือ B-Table [DCES, DCPS]	O(P)	O(P)	$O(\log_2 P)$	P x P
การรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูงขึ้น Efficient Point-to-Point **หรือ** P-Table [OBS]	-	$O(\log_2 P)^2$	O(1)	P

บทที่ 4

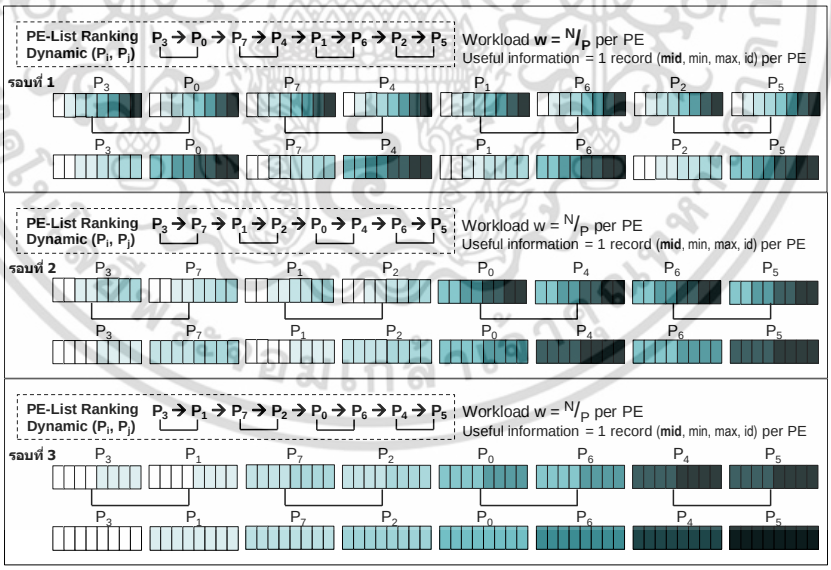
การแลกเปลี่ยนข้อมูลและการรวมข้อมูลที่มีประสิทธิภาพสูง

การเรียงลำดับข้อมูลแบบขนาน (Parallel Sorting) เวลาส่วนใหญ่ใช้ในการแลกเปลี่ยนข้อมูลและการรวมข้อมูลที่แลกเปลี่ยนเข้าด้วยกัน ดังนั้นงานวิจัยในบทนี้ได้นำเสนอการเพิ่มประสิทธิภาพด้านการจัดการชุดข้อมูลแบ่งเป็น 2 ส่วน คือ

1) การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูง (Efficient Data Exchanging)
2) การรวมผสานข้อมูลที่มีประสิทธิภาพสูง (Efficient Data Merging)

4.1 การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูง (Efficient Data Exchanging)

การแลกเปลี่ยนชุดข้อมูลขนาด N/P ระหว่างหน่วยประมวลผลแต่ละคู่ (P_i, P_j) นั้น เกิดขึ้นตลอดเวลาสำหรับการเรียงลำดับแบบขนาน จากรูปที่ 4.1 แสดงภาพรวมของการแลกเปลี่ยนข้อมูลขนาดใหญ่ทั้งหมด (N) ในแต่ละรอบการทำงาน โดยในงานวิจัยที่ได้นำเสนอจะจัดคู่ที่เหมาะสมให้ (P_i, P_j) ด้วยเทคนิค PE-list ranking ด้วยค่ามิดพอยท์ ดังกล่าวในบทที่ 3 เช่น เมื่อ P=8 จะสามารถเรียงค่าเสร็จภายใน $\log_2$ P=3 รอบ ซึ่งมีประสิทธิภาพดีกว่าวิธีปกติที่ต้องใช้จำนวนรอบเท่ากับ $\log_2$ P($\log_2$ P+1)/2 หรือ $\log_2$ 8($\log_2$ 8+1)/2=6 รอบ



รูปที่ 4.1 แสดงภาพรวมของการแลกเปลี่ยนข้อมูลขนาดใหญ่ในแต่ละรอบการทำงาน

เนื่องจากข้อมูลที่มีในแต่ละคู่ของหน่วยประมวลผล (P_i, P_j) ยังไม่ได้ถูกจัดเรียงเข้าด้วยกันอย่างเหมาะสม ดังนั้นสำหรับแต่ละคู่ของหน่วยประมวลผลจะต้องมีการค้นหาข้อมูลเฉพาะที่จำเป็นจำนวนน้อยที่สุด (Partial Data) เพื่อส่งให้หน่วยประมวลผลอื่นที่เป็นคู่กันนำไปใช้ในการรวมผสาน

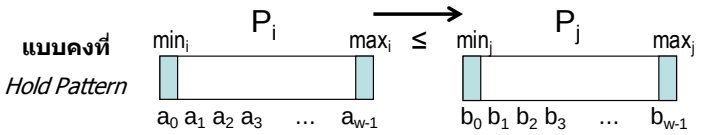
ข้อมูล (Merging Result) ที่อยู่ในคู่ (P_i, P_j) จึงเป็นเรื่องจำเป็น ซึ่งวิธีในการค้นหาข้อมูลเหล่านี้ผู้วิจัยได้นำเสนอเพิ่มเติมจากวิธีซีอีบีเอส [14] ดังที่ได้กล่าวไว้ในบทที่ 2 (3 แบบ) เป็น 4 แบบ ดังนี้

1) การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบคงที่ (Hold Pattern) เป็นการแลกเปลี่ยนข้อมูลแบบที่ดีที่สุด [14] เนื่องจากข้อมูลได้เรียงไว้อย่างเหมาะสมแล้ว จึงไม่จำเป็นต้องแลกเปลี่ยนข้อมูล ซึ่งให้ค่าความซับซ้อนด้านเวลา (Time Complexity) คือ O(1)
2) การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบสลับที่เป็นคงที่ (Swap-to-Hold Pattern) ที่ได้นำเสนอนี้เป็นการแลกเปลี่ยนข้อมูลเฉพาะดัชนีของหน่วยประมวลผล เช่นจาก $P_i \rightarrow P_j$ เป็น $P_j \rightarrow P_i$ ในตารางรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูง (P-Table) หรือตารางการกระจายข้อมูลแบบมีประสิทธิภาพ (B-Table) เท่านั้น ไม่มีการแลกเปลี่ยนข้อมูลจริง ส่งผลให้การแลกเปลี่ยนแบบสลับที่เป็นคงที่ แตกต่างจากวิธีการเดิม (CEBS) [14] ที่กล่าวไว้ในบทที่ 2 โดยให้ค่าความซับซ้อนด้านเวลา คือ O(1) จากเดิมคือ O(N/P) ซึ่งถือว่าเป็นแบบที่ดีที่สุดอีกวิธีหนึ่ง
3) การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบบางส่วน-1 (Partial-1 Pattern) เป็นการแลกเปลี่ยนข้อมูลเพียงบางส่วนที่ใช้ค่าน้อยและค่ามาก ในการตรวจสอบเพื่อพิจารณาว่าเป็นการแลกเปลี่ยนแบบบางส่วน-1 ซึ่งให้ค่าความซับซ้อนด้านเวลา คือ O(1)
4) การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบบางส่วน-2 (Partial-2 Pattern) เป็นการแลกเปลี่ยนข้อมูลเพียงบางส่วนที่ใช้หลักการค้นหาแบบไบนารี ซึ่งจะได้ข้อมูลที่จำเป็นเพื่อใช้ในการแลกเปลี่ยนเท่านั้น ให้ค่าความซับซ้อนด้านเวลา คือ $O(\log_2 w)$ โดย $w=(N/P)/2$ เมื่อ N คือจำนวนข้อมูล และ P คือ จำนวนหน่วยประมวลผล

การเพิ่มประสิทธิภาพในการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลมีผู้ให้ความสนใจและศึกษาค้นคว้าได้กล่าวแล้วในบทที่ 2 ซึ่งได้นำเสนอไว้ 3 แบบคือ 1) แบบคงที่ (Hold Pattern) 2) แบบสลับที่ (Swap Pattern) และ 3) แบบเฉพาะส่วน (Partial Pattern) ซึ่งทั้ง 3 แบบมีข้อแตกต่างกับวิธีการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงที่ได้นำเสนอในงานวิจัยนี้คือ

4.1.1 การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบคงที่ (Hold Pattern)

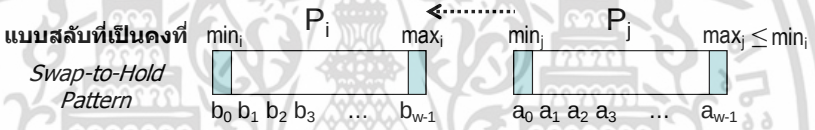
การแลกเปลี่ยนข้อมูลแบบคงที่นี้เป็นแบบที่ดีที่สุด เนื่องจากข้อมูลได้เรียงไว้อย่างเหมาะสมแล้ว กล่าวคือไม่มีการแลกเปลี่ยนข้อมูลเกิดขึ้นระหว่างหน่วยประมวลผลนั้นเอง ส่งผลให้เวลาที่ใช้การเรียงลำดับชุดข้อมูล และเวลาที่ใช้ในการประมวลผลชุดข้อมูลลดลงหากมีข้อมูลลักษณะนี้อยู่ในชุดข้อมูลที่ต้องการเรียงลำดับ ดังรูปที่ 4.2 แสดงรูปแบบข้อมูลทั่วไปในการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบคงที่ โดยที่ค่า $a_{w-1} \leq b_0$ หรือกล่าวอีกนัยหนึ่งคือค่ามากของหน่วยประมวลผล P_i (max_i) มีค่าน้อยกว่า ค่าน้อยของหน่วยประมวลผล P_j (min_j)



รูปที่ 4.2 แสดงรูปแบบข้อมูลทั่วไปในการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบคงที่

4.1.2 การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบสลับที่เป็นคงที่ (Swap-to-Hold Pattern)

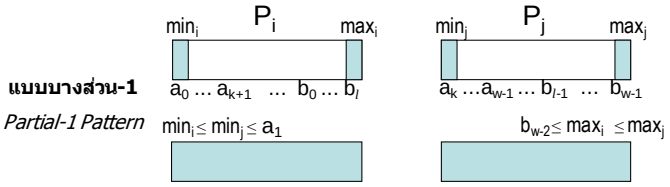
การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบสลับที่เป็นคงที่ ซึ่งนำเสนอในงานวิจัยนี้แตกต่างจากงานวิจัยเดิม (CEBS) [14] ที่ได้กล่าวไว้ในบทที่ 2 เนื่องจากวิธีที่ได้นำเสนอนี้เป็นการแลกเปลี่ยนข้อมูลเฉพาะดัชนีของหน่วยประมวลผลจาก $P_i \rightarrow P_j$ เป็น $P_j \rightarrow P_i$ ที่อยู่ในตารางรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูง หรืออยู่ในตารางการกระจายข้อมูลแบบมีประสิทธิภาพเท่านั้น ไม่มีการแลกเปลี่ยนข้อมูลจริง ส่งผลให้การแลกเปลี่ยนข้อมูลแบบสลับที่เป็นคงที่นี้มีคุณสมบัติเดียวกับการแลกเปลี่ยนข้อมูลแบบคงที่ ดังรูปที่ 4.3 แสดงข้อมูลทั่วไปในการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบสลับที่ โดยที่ค่า $a_{w-1} \leq b_0$ หรือกล่าวอีกนัยหนึ่งคือค่ามากของหน่วยประมวลผล P_j (max_j) มีค่าน้อยกว่า ค่าน้อยของหน่วยประมวลผล P_i (min_i)



รูปที่ 4.3 แสดงข้อมูลทั่วไปในการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบสลับที่เป็นคงที่

4.1.3 การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบบางส่วน-1 (Partial-1 Pattern)

โดยทั่วไปแล้วขนาดของข้อมูลที่ใช้ในการเรียงลำดับข้อมูลในแต่ละหน่วยประมวลผลจะมีขนาดเท่าๆ กันคือ w=N/P เมื่อ N คือ จำนวนข้อมูล และ P คือ จำนวนหน่วยประมวลผล ดังนั้นการแลกเปลี่ยนข้อมูลจึงมีขนาดเท่ากับ w แต่สำหรับการแลกเปลี่ยนข้อมูลเพียงบางส่วนจะคำนวณหาชุดข้อมูลย่อยเฉพาะที่จำเป็นในการเรียงลำดับข้อมูลภายในคู่ (P_i, P_j) เท่านั้น โดยส่วนใหญ่แล้วมีขนาดน้อยกว่า w ซึ่งการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบบางส่วน-1 คำนวณจากค่าน้อยและค่ามากคล้ายๆ ในหัวข้อก่อนด้วยเวลา O(1) เพื่อให้ได้ชุดข้อมูลย่อยเฉพาะที่จำเป็นเพื่อส่งให้กับหน่วยประมวลผลที่ติดต่อด้วย ในรูปที่ 4.4 แสดงข้อมูลทั่วไปในการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบบางส่วน-1 โดยตรวจสอบเงื่อนไขจากค่าของ $min_i \leq min_j \leq a_1$ และ $b_{w-2} \leq max_i \leq max_j$ หรือกล่าวอีกนัยหนึ่งคือค่ามากของหน่วยประมวลผล P_j (max_j) มีค่าน้อยกว่า ค่ามากของหน่วยประมวลผล P_i (max_i) และค่าน้อยของหน่วยประมวลผล P_j (min_j) มีค่ามากกว่า ค่าน้อยของหน่วยประมวลผล P_i (min_i) ถ้าเป็นไปตามเงื่อนไขข้างต้น ชุดข้อมูลย่อยที่คำนวณได้มีขนาดน้อยกว่า w=N/P

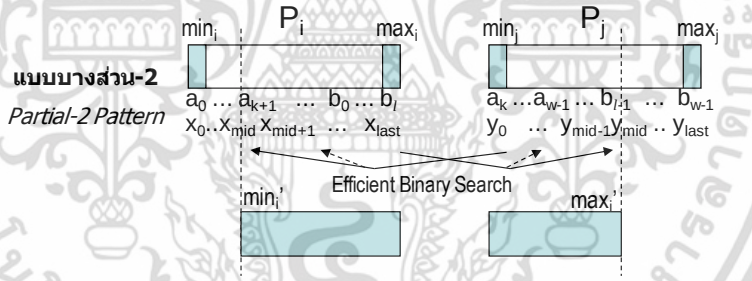


รูปที่ 4.4 แสดงรูปแบบข้อมูลทั่วไปในการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบบางส่วน-1

<u>หมายเหตุ</u> ถ้าชุดข้อมูลย่อยมีปริมาณ > w/2 วิธีนี้ถูกปรับให้มีประสิทธิภาพสูงขึ้นด้วยการใช้เทคนิค Inverted Data เพื่อส่งข้อมูลขนาด ≤ w/2 ดังนั้นลำดับของ (P_i, P_j) จะเปลี่ยนจาก $P_i \rightarrow P_j$ เป็น $P_j \rightarrow P_i$

4.1.4 การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบบางส่วน-2 (Partial-2 Pattern)

จากการตรวจสอบเงื่อนไขการแลกเปลี่ยนข้อมูลแบบบางส่วน-1 หากไม่เป็นไปตามเงื่อนไขให้ตรวจสอบด้วยเงื่อนไขการแลกเปลี่ยนข้อมูลแบบบางส่วน-2 ซึ่งคำนวณจากค่าน้อยและค่ามาก จากนั้นใช้หลักการค้นหาแบบไบนารีที่เพิ่มประสิทธิภาพ (Efficient Binary Search) ด้วยเวลา $O(\log_2 w)$ ที่นำข้อดีของ Binary Search สำหรับค่าตรงกลาง $O(\log_2 w)$ และข้อดีของ Sequence Search สำหรับค่าตรงปลาย O(1) ดังแสดงในรูปที่ 4.5 เพื่อให้ได้ชุดข้อมูลย่อยเฉพาะที่จำเป็นเพื่อรับ/ส่งข้อมูลให้หน่วยประมวลผลที่ติดต่อด้วย โดยมีเงื่อนไขในการตรวจสอบ แบ่งเป็น 2 กรณี สำหรับ P_i กรณีแรก และ P_j กรณีที่สอง ดังนี้



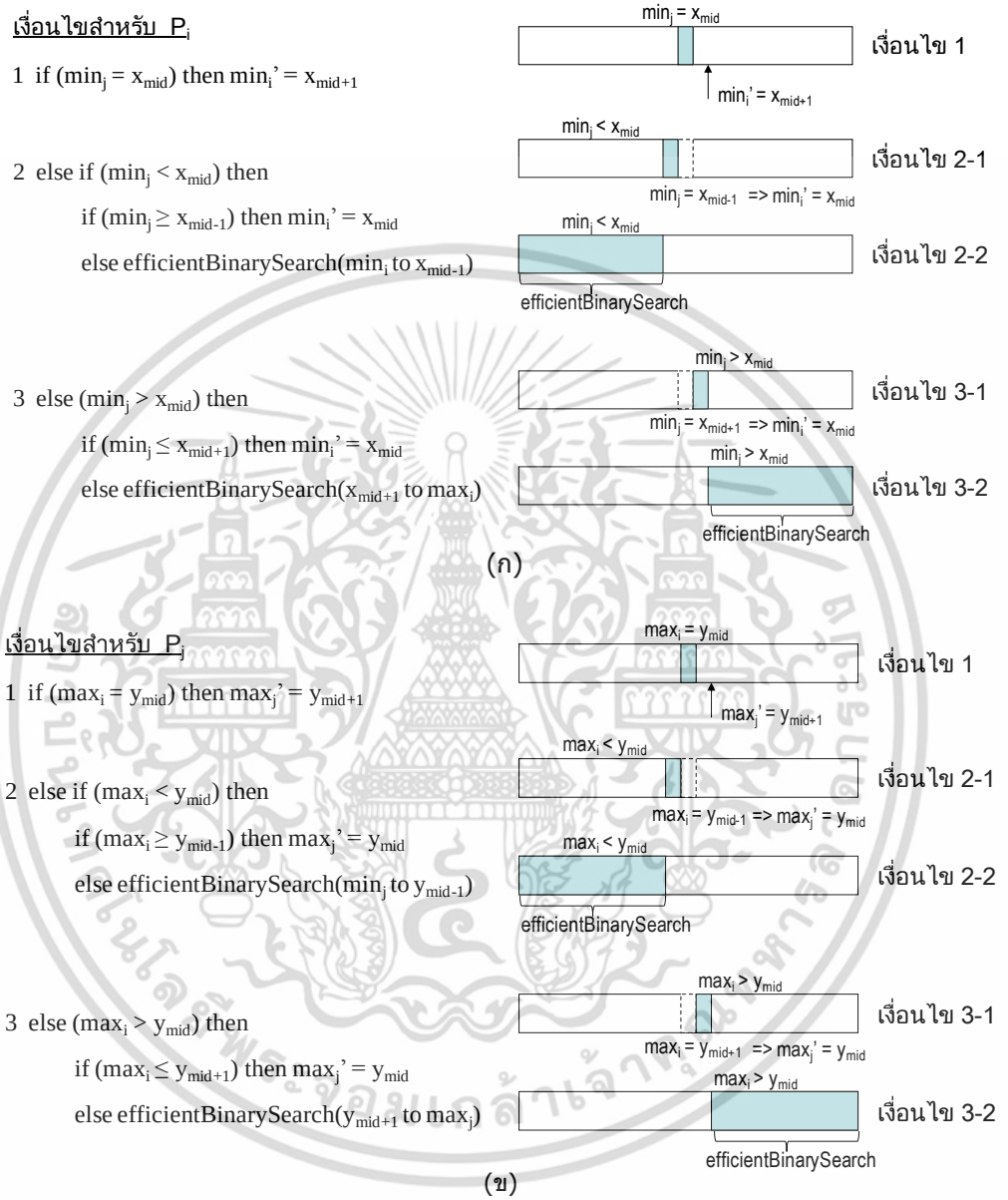
รูปที่ 4.5 แสดงรูปแบบข้อมูลทั่วไปในการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบบางส่วน-2

รูปที่ 4.6(ก) แสดงหลักการและตัวอย่างข้อมูลทั่วไปในการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบบางส่วน-2 สำหรับแต่ละคู่ของหน่วยประมวลผล (P_i, P_j)

<u>กรณีที่ 1</u> สำหรับ P_i (รูปที่ 46(ข)) จะประยุกต์ใช้เทคนิค Binary Search ที่ต้องตรวจสอบค่า min_j กับค่ากลาง x_{mid} ด้วยเงื่อนไข 3 เงื่อนไข คือ 1. $min_j = x_{mid}$ หรือ 2. $min_j < x_{mid}$ หรือ 3. $min_j > x_{mid}$ สำหรับเงื่อนไขในกรณีแรกคือ ถ้า $min_j = x_{mid}$ จะได้ค่า $min_i' = x_{mid+1}$ สำหรับเงื่อนไขในกรณีสอง ถ้า $min_j < x_{mid}$ จะเพิ่มการตรวจสอบเงื่อนไขย่อยด้วยการค้นหาจากค่าที่ติดกับ x_{mid} ดังนี้ ถ้า $min_j \geq x_{mid-1}$ จะได้ค่า $min_i' = x_{mid}$ แต่ถ้าไม่เป็นไปตามเงื่อนไขย่อยดังกล่าว จึงทำการคำนวณหาชุดข้อมูลย่อยในรอบต่อไปด้วย Binary Search ในช่วงข้อมูลที่ลดลงครึ่งหนึ่งคือ จากค่า min_i ถึง x_{mid-1} และในเงื่อนไขกรณีที่สามเมื่อ $min_j > x_{mid}$ ซึ่งจะเพิ่มการตรวจสอบเงื่อนไขย่อยด้วยการค้นหาจากค่าที่ติดกับ x_{mid} ดังนี้ ถ้า $min_j \leq x_{mid-1}$ จะได้ค่า $min_s' = x_{mid}$ แต่ถ้าไม่เป็นไปตามเงื่อนไขย่อยนี้ จึงทำการ

คำนวณหาชุดข้อมูลย่อยในรอบต่อไปด้วย Binary Search ในช่วงข้อมูลที่ลดลงครึ่งหนึ่งคือ จากค่า x_{mid-1} ถึง max_i

<u>กรณีที่ 2</u> สำหรับ P_j จะดำเนินการได้ในทำนองเดียวกันกับ P_i ดังแสดงในรูปที่ 46(ข) ในกรณี P_j



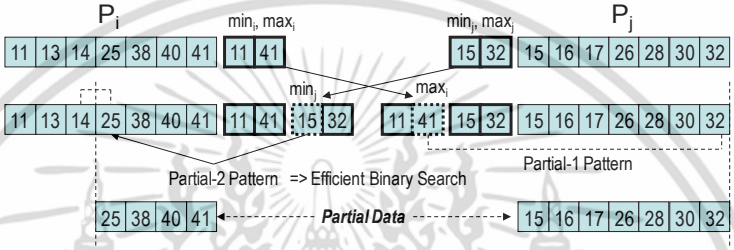
รูปที่ 4.6 แสดงการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงแบบบางส่วน-2 และเงื่อนไข (ก) สำหรับ P_i และ (ข) สำหรับ P_j

<u>ตัวอย่างที่ 4.1</u> **แสดงตัวอย่างการแลกเปลี่ยนข้อมูลแบบบางส่วน-1 และบางส่วน-2**

สมมติให้ข้อมูลนำเข้าคือ P_i เก็บค่า {<u>11</u>, 13, 14, 25, 38, 40, <u>41</u>} และ P_j เก็บค่า {<u>15</u>, 16, 17, 26, 28, 30, <u>32</u>} โดยส่งค่าน้อย (min_i) = 11 และค่ามาก (max_i) = 41 ในหน่วยประมวลผล P_i ให้กับ P_j และหน่วยประมวลผล P_j ส่งค่าน้อย (min_j) = 15 และค่ามาก (max_j) = 32 ให้กับหน่วยประมวลผล P_i

ในรูปที่ 4.7 หน่วยประมวลผล P_i ตรวจสอบเงื่อนไขว่าเป็นแบบบางส่วน-1 หรือไม่ โดยตรวจสอบว่า max_i (41) < max_j (32) และ min_i (11) > min_j (15) ไม่เป็นไปตามเงื่อนไข ตรวจสอบต่อว่าเป็นแบบบางส่วน-2 ด้วยเทคนิค Efficient Binary Search ดังนี้ ในรอบแรก $min_j(15) < x_{mid}(25)$ ดังนั้นตรวจสอบเงื่อนไขย่อย $min_j(15) \geq x_{mid-1}(14)$ ซึ่งจะได้ค่า $min'_i = x_{mid}$ และไม่ต้องทำรอบต่อไป ดังนั้นชุดข้อมูลย่อยแบบบางส่วน-2 ในหน่วยประมวลผล P_i คือ {25, 38, 40, 41}

ในขณะเดียวกันหน่วยประมวลผล P_j ตรวจสอบเงื่อนไขว่าเป็นแบบบางส่วน-1 หรือไม่ โดยตรวจสอบดังนี้ max_i (41) > max_j (32) และ min_i (11) < min_j (15) เป็นไปตามเงื่อนไขดังนั้นชุดข้อมูลย่อยเป็นแบบบางส่วน-1 ในหน่วยประมวลผล P_j คือ {15, 16, 17, 26, 28, 30, 32}



รูปที่ 4.7 ตัวอย่างการแลกเปลี่ยนข้อมูลแบบบางส่วน-1 และแบบบางส่วน-2

หลังจากหน่วยประมวลผลทั้งสองตรวจสอบเงื่อนไขเรียบร้อยแล้ว จะได้ชุดข้อมูลย่อย (Partial Data) หากเป็นงานวิจัยที่ผ่านๆ มา วิธี (CEBS) [14] แลกเปลี่ยนข้อมูลชุดดังกล่าว ระหว่างหน่วยประมวลผลที่ติดต่อด้วยทันที แต่ในงานวิจัยนี้ได้นำเสนอขั้นตอนการตรวจสอบข้อมูลเพิ่มขึ้นอีกขั้นหนึ่งเพื่อให้ได้มาซึ่งชุดข้อมูลย่อยที่จำเป็นใช้จริงเท่านั้น โดยวิธีการดังการนี้เรียกว่า “การคำนวณด้วยค่ามัธยฐาน” (Median Computing) ที่มีความแม่นยำสูงกว่าวิธี (LBM) [12] ดังแสดงในรูปที่ 4.8 นำชุดข้อมูลย่อย (Partial Data) ที่ได้จากตัวอย่างข้างต้น มาหาค่ามัธยฐาน(Median: med) โดยแบ่งประเภทของค่ากลาง ได้ดังนี้

med_i1 คือ ค่ามัธยฐานของชุดข้อมูลย่อย (Median of Partial Data) ในหน่วยประมวลผล P_i

med_i2 คือ ค่ามัธยฐานของชุดข้อมูลนำเข้า (Median of Data Set) ในหน่วยประมวลผล P_i

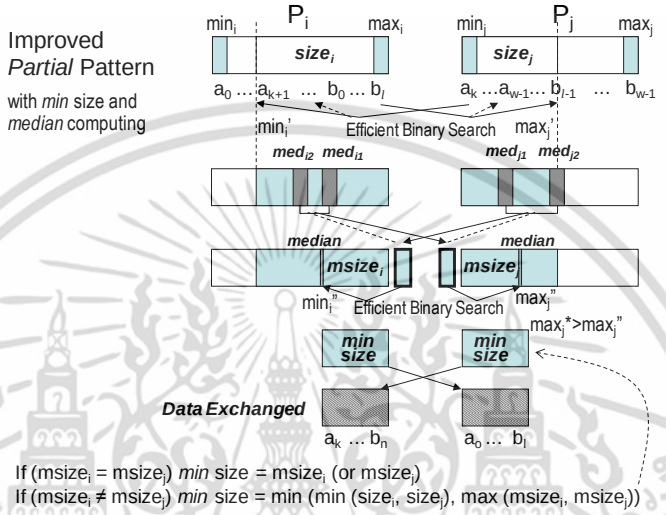
med_j1 คือ ค่ามัธยฐานของชุดข้อมูลย่อย (Median of Partial Data) ในหน่วยประมวลผล P_j

med_j2 คือ ค่ามัธยฐานของชุดข้อมูลนำเข้า (Median of Data Set) ในหน่วยประมวลผล P_j

นำค่ามัธยฐาน (med) ของ med_i1, med_i2, med_j1, med_j2 ที่ได้มาคำนวณหาชุดข้อมูลย่อยที่จำเป็น (Minimum Data Exchanged) เพื่อใช้ในการเรียงลำดับข้อมูล โดยเงื่อนไขในการตรวจสอบใช้วิธีเดียวกับการนำค่าน้อยและค่ามาก มาตรวจสอบนั้นเอง หลังจากได้ชุดข้อมูลย่อยที่จำเป็นแล้วส่งขนาดของชุดข้อมูลย่อยที่จำเป็น ให้กับหน่วยประมวลผลที่ติดต่อด้วย เพื่อตรวจสอบขนาดของข้อมูล หากมีขนาดเท่ากันให้หน่วยประมวลผลทั้งสองแลกเปลี่ยนข้อมูลได้เลย ถ้าขนาดไม่เท่ากันแต่ละหน่วยประมวลผลพิจารณาค่าทั้งสองเพื่อใช้ในการแลกเปลี่ยนข้อมูลต่อไป โดยขนาดของข้อมูล ≤ w/2 (w=N/P) หากมากกว่าจะใช้เทคนิค Inverted Data ซึ่งได้กล่าวถึงรายละเอียดต่อไป

ผู้วิจัยได้ประยุกต์ใช้การคำนวณด้วยค่ามัธยฐานกับการเรียงลำดับแบบดีซีพีเอสและแบบโอบีเอส ส่วนการเรียงลำดับแบบดีซีอีเอส ใช้การคำนวณค่ามัธยฐาน (Median) เพียง 2 ค่า คือ ค่า med_i2 คือ ค่ามัธยฐานของชุดข้อมูลนำเข้า (Median of Data Set) ในหน่วยประมวลผล P_i และค่า med_j2 คือ ค่ามัธยฐานของชุดข้อมูลนำเข้า (Median of Data Set) ในหน่วยประมวลผล P_j

<u>หมายเหตุ</u> หากชุดข้อมูลนำเข้า (Data Set) มีจำนวนเท่ากับเลขคู่ ให้นำค่าที่อยู่ตรงกลางทั้งสองค่ามาบวกกันแล้วหารสองเพื่อให้ได้ค่ามัธยฐานที่ใกล้เคียงที่สุด

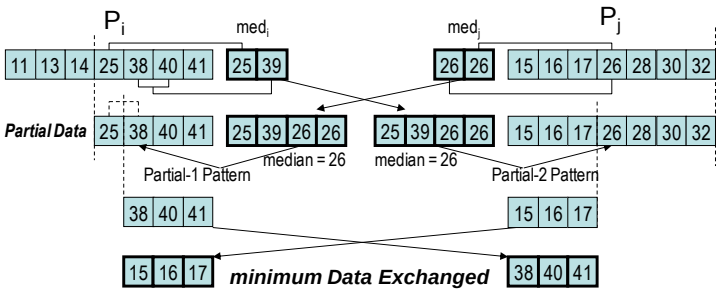


รูปที่ 4.8 แสดงการประยุกต์ใช้การคำนวณด้วยค่ามัธยฐาน(Median Computing) ใน Partial Pattern

จากตัวอย่างที่ 4.1 หลังจากได้ชุดข้อมูลย่อยแล้ว นำชุดข้อมูลย่อยดังกล่าวมาคำนวณหาชุดข้อมูลย่อยที่จำเป็นจากวิธีการคำนวณด้วยค่ามัธยฐานซึ่งได้อธิบายรายละเอียดในข้างต้นแล้ว

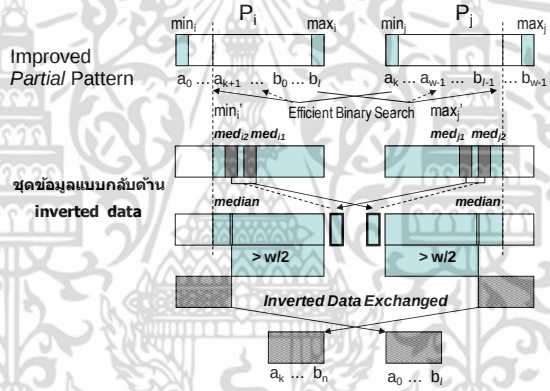
<u>ตัวอย่างที่ 4.2</u> **แสดงตัวอย่างการคำนวณชุดข้อมูลย่อยที่จำเป็น ด้วยวิธีคำนวณด้วยค่ามัธยฐาน**

จากชุดข้อมูลตัวอย่างเดียวกับตัวอย่างที่ 4.1 ในรูปที่ 4.9 หน่วยประมวลผล P_i มีชุดข้อมูลย่อย {25, 38, 40, 41} และหน่วยประมวลผล P_j มีชุดข้อมูลย่อย {15, 16, 17, 26, 28, 30, 32} คำนวณหาค่ามัธยฐานในหน่วยประมวลผล P_i ได้ดังนี้ med_i1 (38+40)/2 = 39, med_i2 = 25 และคำนวณหาค่ามัธยฐานในหน่วยประมวลผล P_j คือ med_j1 = 26 และ med_j2 = 26 ส่งค่ามัธยฐาน (med) ที่คำนวณได้ให้หน่วยประมวลผลที่ติดต่อด้วย ดังแสดงในรูป นำค่ามัธยฐานที่คำนวณได้ใน (P_i, P_j) มาเรียงลำดับได้ดังนี้ {25, 26, 26, 39} คำนวณค่ามัธยฐานจากชุดข้อมูลนี้ (26+26)/2 = 26 นำค่ามัธยฐานนี้มาตรวจสอบตามเงื่อนไขของการแลกเปลี่ยนข้อมูลแบบบางส่วน-1 และแบบบางส่วน-2 ที่ได้กล่าวไว้ข้างต้น แลกเปลี่ยนชุดข้อมูลย่อยที่จำเป็น ในหน่วยประมวลผล P_i {38, 40, 41} และหน่วยประมวลผล P_j {15, 16, 17} จากนั้นหน่วยประมวลผลทั้งสองเข้าสู่กระบวนการรวมข้อมูลต่อไป



รูปที่ 4.9 แสดงตัวอย่างการประยุกต์ใช้การคำนวณด้วยค่ามัธยฐานใน Partial Pattern

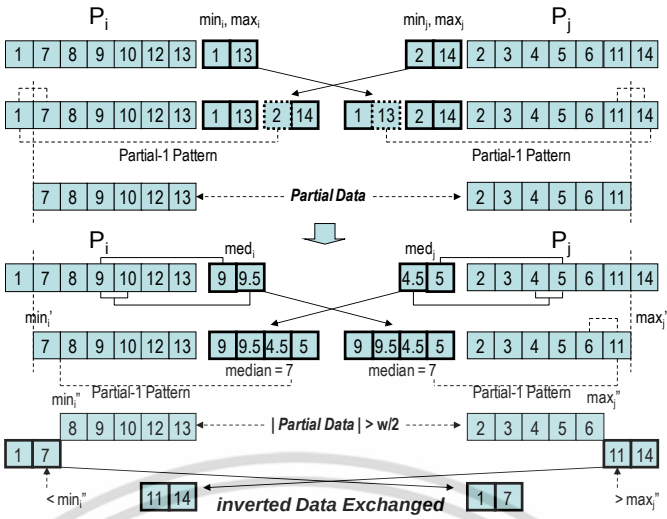
นอกเหนือจากตัวอย่างโดยทั่วไปแล้วจุดเด่นของงานวิจัยอีกจุดหนึ่งซึ่งเป็นผลดีต่อการจัดคู่ (P_i, P_j) แบบไดนามิค คือ สามารถใช้ "ชุดข้อมูลแบบกลับด้าน" (Inverted Data) เพื่อส่งข้อมูลน้อยกว่าครึ่งเสมอกล่าวคือ เมื่อคำนวณหาชุดข้อมูลย่อยที่จำเป็นเรียบร้อยแล้ว ตรวจสอบว่าขนาดของชุดข้อมูลนี้ มากกว่า w/2 โดย w=N/P (N คือ จำนวนข้อมูล, P คือ จำนวนหน่วยประมวลผล) จะเปลี่ยนเป็นส่งชุดข้อมูลย่อยฝั่งตรงข้ามให้กับหน่วยประมวลผลที่ติดต่อด้วย ดังแสดงในรูปที่ 4.10



รูปที่ 4.10 แสดงการประยุกต์วิธีการคำนวณด้วยค่ามัธยฐานแบบกลับด้าน (Inverted Data)

<u>ตัวอย่างที่ 4.3</u> **แสดงตัวอย่างการคำนวณข้อมูลด้วยค่ามัธยฐานแบบกลับด้าน (Inverted Data)**

สมมติให้ข้อมูลนำเข้าในหน่วยประมวลผล P_i{1, 7, 8, 9, 10, 12, 13} และหน่วยประมวลผล P_j {2, 3, 4, 5, 6, 11, 14} ตรวจสอบเงื่อนไขต่างๆ พบว่าข้อมูลทั้งสองชุดนี้เป็นแบบบางส่วน-1 หลังจากได้ชุดข้อมูลย่อยแล้ว คำนวณหาค่ากลางตามที่ได้อธิบายแล้วข้างต้น จากนั้นตรวจสอบเงื่อนไขการแลกเปลี่ยนข้อมูลแบบบางส่วน-1 และแบบบางส่วน-2 จะได้ชุดข้อมูลย่อยที่จำเป็น ในหน่วยประมวลผล P_i{8, 9, 10, 12, 13} และหน่วยประมวลผล P_j {2, 3, 4, 5, 6} ซึ่งข้อมูลทั้งสองชุดนี้มากกว่า w/2 (7/2=3.5) ดังนั้นทำการกลับข้อมูลที่ต้องแลกเปลี่ยนเป็นฝั่งตรงข้าม ดังรูปที่ 4.11 ในหน่วยประมวลผล P_i ส่งชุดข้อมูลแบบกลับด้านคือ {1, 7} ให้หน่วยประมวลผล P_j ในขณะเดียวกัน P_j ส่งชุดข้อมูลแบบกลับด้านคือ {11, 14} ให้หน่วยประมวลผล P_i เช่นกัน



รูปที่ 4.11 แสดงตัวอย่างการประยุกต์วิธีใช้การคำนวณข้อมูลด้วยค่ามัธยฐานแบบกลับด้าน

สำหรับการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูงทั้ง 4 แบบ ที่ได้นำเสนอในงานวิจัยนี้ โดยใน 3 แบบหลัง คือ แบบสลับที่เป็นคงที่ แบบบางส่วน-1 และแบบบางส่วน-2 ผู้วิจัยได้ปรับปรุงประสิทธิภาพเพิ่มขึ้นจากคำอธิบายจะเห็นได้อย่างชัดเจน ส่วนแบบคงที่ ใช้หลักการเดียวกับวิธีการเดิม (CEBS) [14] ที่มีผู้นำเสนอไว้แล้ว

นอกจากการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพแล้ว การรวมชุดข้อมูลที่มีประสิทธิภาพ ก็เป็นเรื่องสำคัญและจำเป็น ซึ่งช่วยลดเวลาที่ใช้ในการประมวลผลลง อีกทั้งช่วยส่งเสริมให้เวลาที่ใช้ในการเรียงลำดับข้อมูลโดยรวมลดลงอีกด้วย ซึ่งได้อธิบายรายละเอียดไว้ในหัวข้อที่ 4.2

4.2 การรวมข้อมูลที่มีประสิทธิภาพสูง (Efficient Data Merging)

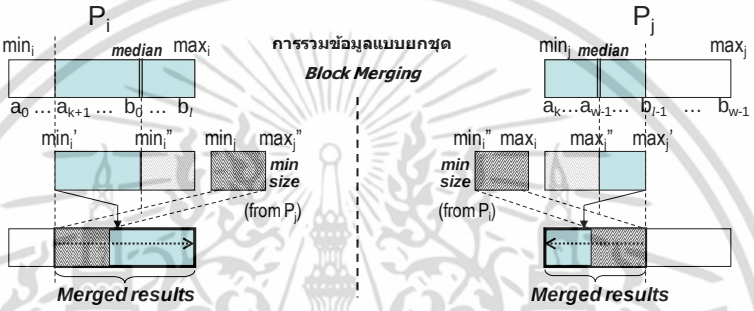
การรวมข้อมูลเข้าด้วยกันระหว่างหน่วยประมวลผลทั้งสอง ถือเป็นกระบวนการสุดท้ายก่อนกระบวนการตรวจสอบสถานะเพื่อสิ้นสุดการทำงานสำหรับการเรียงลำดับข้อมูลแบบขนาน งานวิจัยในอดีตที่ผ่านมาใช้วิธีการรวมชุดข้อมูลแบบเปรียบเทียบทีละค่า ในงานวิจัยนี้นำเสนอการรวมข้อมูลที่มีประสิทธิภาพสูง โดยแบ่งการรวมข้อมูลออกเป็น 2 แบบ คือ

1) การรวมข้อมูลแบบบล็อก (Block Merging) เป็นการตรวจสอบข้อมูลเพื่อรวมชุดข้อมูลย่อยที่จำเป็นในชุดข้อมูลนำเข้าได้ทันที
2) การรวมข้อมูลแบบหนึ่งต่อหนึ่ง (1-1 Merging) เป็นการรวมชุดข้อมูลย่อยในชุดข้อมูลนำเข้าที่ละค่าจนครบจำนวน

ในการรวมข้อมูลระหว่างหน่วยประมวลผลเข้าด้วยกันเป็นกระบวนการหนึ่งที่สำคัญหลังจากรับ/ส่งข้อมูลระหว่างหน่วยประมวลผลที่ติดต่อด้วย ซึ่งผู้วิจัยได้ประยุกต์ใช้การรวมข้อมูลที่มีประสิทธิภาพสูงกับการเรียงลำดับแบบดีซีพีเอส และแบบโอบีเอส ส่วนการเรียงลำดับแบบดีซีอีเอส ใช้การรวมข้อมูลแบบเปรียบเทียบทีละค่า ดังนี้

4.2.1 การรวมข้อมูลแบบบล็อก (Block Merging)

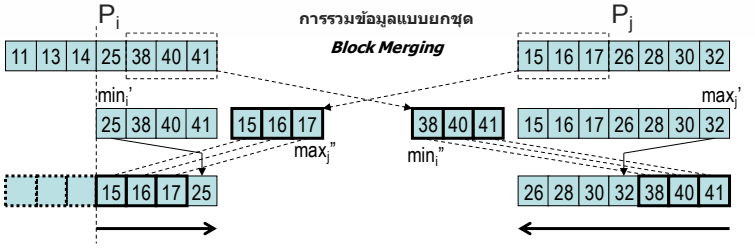
การรวมข้อมูลแบบบล็อก เป็นวิธีการที่ได้นำเสนอขึ้นใหม่ในงานวิจัยนี้ ด้วยการตรวจสอบเงื่อนไขดังแสดงในรูปที่ 4.12 หน่วยประมวลผล P_i และ P_j แลกเปลี่ยนชุดข้อมูลย่อยที่จำเป็น โดยหน่วยประมวลผล P_i ตรวจสอบเงื่อนไขจากชุดข้อมูลย่อยคือ max_j" $\leq$ min_j' ถ้าเป็นไปตามเงื่อนไขดังกล่าวให้รวมชุดข้อมูลย่อยลงในชุดข้อมูลนำเข้าดังแสดงในรูปที่ 4.11 ด้านซ้าย ในขณะเดียวกันหน่วยประมวลผล P_j ตรวจสอบเงื่อนไขจากชุดข้อมูลย่อยคือ min_j" $\geq$ max_j' ถ้าเป็นไปตามเงื่อนไขดังกล่าวให้รวมชุดข้อมูลย่อยลงในชุดข้อมูลนำเข้าดังแสดงในรูปที่ 4.12 ด้านขวา หากไม่เป็นไปตามเงื่อนไขดังกล่าวให้ตรวจสอบการรวมข้อมูลแบบหนึ่งต่อหนึ่ง ซึ่งอธิบายในหัวข้อถัดไป



รูปที่ 4.12 แสดงการประยุกต์วิธีการรวมข้อมูลแบบบล็อก (Block Merging)

ตัวอย่างที่ 4.4 **แสดงตัวอย่างการรวมข้อมูลแบบบล็อก**

จากชุดข้อมูลตัวอย่างเดียวกับตัวอย่างที่ 4.1 และ 4.2 หลังจากได้ชุดข้อมูลย่อยที่จำเป็นแล้ว หน่วยประมวลผลแลกเปลี่ยนข้อมูลกัน จากนั้นเข้าสู่กระบวนการรวมข้อมูล ดังแสดงในรูปที่ 4.13 หน่วยประมวลผล P_i ส่ง {38, 40, 41} ให้หน่วยประมวลผล P_j ในขณะเดียวกัน P_j ส่ง {15, 16, 17} ให้หน่วยประมวลผล P_i เช่นกัน จากข้อมูลที่ได้รับ P_i ตรวจสอบเงื่อนไขการรวมแบบบล็อก คือ max_j" (17) $\leq$ min_j' (25) เป็นไปตามเงื่อนไข ชุดข้อมูลย่อยที่จำเป็นที่ได้จากหน่วยประมวลผล P_j รวมชุดข้อมูลในตำแหน่งก่อนค่า min_j' และในขณะเดียวกันหน่วยประมวลผล P_j ตรวจสอบเงื่อนไขการรวมแบบบล็อกเช่นกันคือ min_i" (38) $\geq$ max_j'(32) หากเป็นไปตามเงื่อนไขนำชุดย่อยข้อมูลที่จำเป็นที่ได้จากหน่วยประมวลผล P_i รวมกับชุดข้อมูลนำเข้าได้ทันที

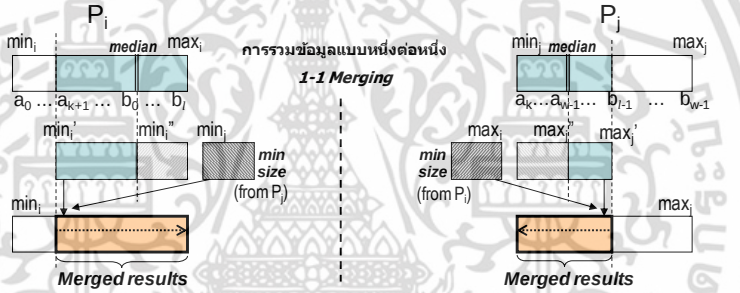


รูปที่ 4.13 แสดงตัวอย่างการประยุกต์ใช้วิธีการรวมข้อมูลแบบบล็อก

4.2.2 การรวมข้อมูลแบบหนึ่งต่อหนึ่ง (1-1 Merging)

การรวมข้อมูลแบบหนึ่งต่อหนึ่งนี้ถูกใช้งานอย่างแพร่หลายในอดีตที่ผ่านมา จากวิธีการดังกล่าวผู้วิจัยได้นำมาพัฒนาต่อยอดให้การรวมข้อมูลแบบหนึ่งต่อหนึ่งมีประสิทธิภาพดียิ่งขึ้น กล่าวคือ ตำแหน่งเริ่มต้นการรวมข้อมูลถ้าเรียงจากน้อยไปมาก เริ่มจากตำแหน่งแรกของชุดข้อมูลนำเข้า (Data Set) ส่วนการเรียงจากมากไปน้อยเริ่มจากตำแหน่งสุดท้ายของชุดข้อมูลนำเข้า จากนั้นเปรียบเทียบข้อมูลที่ละค่าจนครบจำนวน ส่วนวิธีการที่ผู้วิจัยได้นำเสนอคือ ตำแหน่งเริ่มต้นการรวมข้อมูลแบบหนึ่งต่อหนึ่งจากน้อยไปมาก เริ่มจากตำแหน่งแรกของชุดข้อมูลย่อย และการเรียงจากมากไปน้อยเริ่มจากตำแหน่งสุดท้ายของชุดข้อมูลย่อย

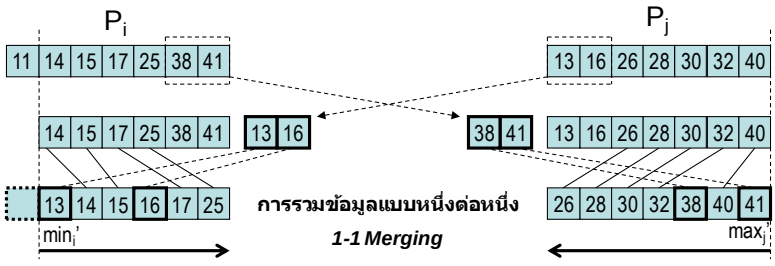
รูปที่ 4.14 แสดงตัวอย่างการประยุกต์ใช้วิธีรวมข้อมูลแบบหนึ่งต่อหนึ่ง โดยหน่วยประมวลผล P_i เปรียบเทียบชุดข้อมูลที่ละค่าโดยเริ่มจากตำแหน่ง min_i' หากเป็นวิธีการทั่วไปจะเริ่มจาก min_i ดังรูปที่ 4.14 ด้านซ้าย ในขณะเดียวกันหน่วยประมวลผล P_j เปรียบเทียบชุดข้อมูลที่ละค่าโดยเริ่มจากตำแหน่ง max_j' หากเป็นวิธีทั่วไป เริ่มจาก max_j ดังรูป 4.14 ด้านขวา



รูปที่ 4.14 แสดงการประยุกต์วิธีการรวมข้อมูลแบบหนึ่งต่อหนึ่ง (1-1 Merging)

ตัวอย่างที่ 4.5 **แสดงตัวอย่างการรวมข้อมูลแบบหนึ่งต่อหนึ่ง (1-1 Merging)**

สมมติให้ข้อมูลนำเข้าของหน่วยประมวลผล P_i {11, 14, 15, 17, 25, 38, 41} และหน่วยประมวลผล P_j {13, 16, 26, 28, 30, 32, 40} โดยมีชุดข้อมูลย่อยใน P_i {38, 41} และใน P_j {13, 16} แลกเปลี่ยนข้อมูลดังกล่าวระหว่างหน่วยประมวลผล จากนั้นแต่ละหน่วยประมวลผลเปรียบเทียบข้อมูลทีละค่า โดยหน่วยประมวลผล P_i เริ่มจากตำแหน่งของ min_i'(13) หากเป็นวิธีการทั่วไปจะเริ่มจากตำแหน่ง min_i(11) ในขณะเดียวกันหน่วยประมวลผล P_j เริ่มจากตำแหน่ง max_j'(40) ซึ่งเหมือนกับวิธีการทั่วไป ดังรูปที่ 4.15



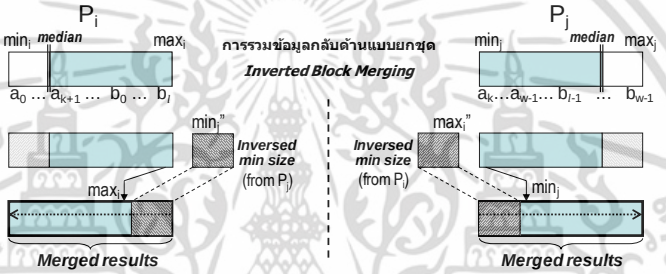
รูปที่ 4.15 แสดงตัวอย่างการประยุกต์ใช้วิธีการรวมข้อมูลแบบหนึ่งต่อหนึ่ง

เนื่องจากการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลมีการนำเสนอวิธีการแลกเปลี่ยนข้อมูลแบบกลับด้าน (Inverted Data Exchanging) ดังนั้นการรวมข้อมูลที่มีประสิทธิภาพสูงนี้รองรับข้อมูลในลักษณะนี้เช่นกัน โดยแบ่งการรวมข้อมูลแบบกลับด้านเป็น 2 แบบคือ

1) การรวมข้อมูลกลับด้านแบบบล็อก (Inverted Block Merging)

2) การรวมข้อมูลกลับด้านแบบหนึ่งต่อหนึ่ง (Inverted 1-1 Merging)

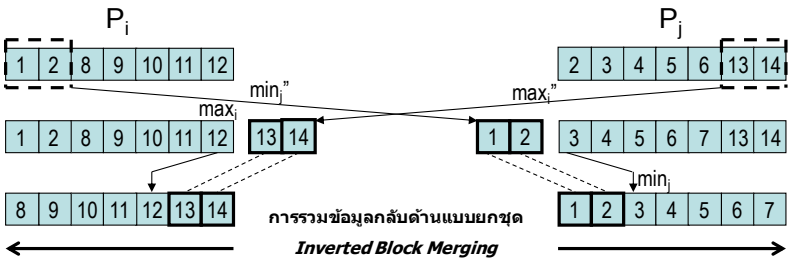
รูปที่ 4.16 แสดงการรวมข้อมูลกลับด้านแบบบล็อก โดยหน่วยประมวลผล P_i รับชุดข้อมูลย่อยที่จำเป็นจากหน่วยประมวลผล P_j จากนั้นตรวจสอบเงื่อนไข $max_i \leq min_j$" ซึ่งเป็นไปตามเงื่อนไขรวมข้อมูลย่อยชุดดังกล่าวในชุดข้อมูลนำเข้า ในขณะเดียวกันหน่วยประมวลผล P_j รับชุดข้อมูลย่อยที่จำเป็นจากหน่วยประมวลผล P_i ตรวจสอบเงื่อนไข max_i" $\leq min_j$ เป็นไปตามเงื่อนไขรวมข้อมูลย่อยชุดดังกล่าวในชุดข้อมูลนำเข้า



รูปที่ 4.16 แสดงการประยุกต์วิธีการรวมข้อมูลกลับด้านแบบบล็อก (Inverted Block Merging)

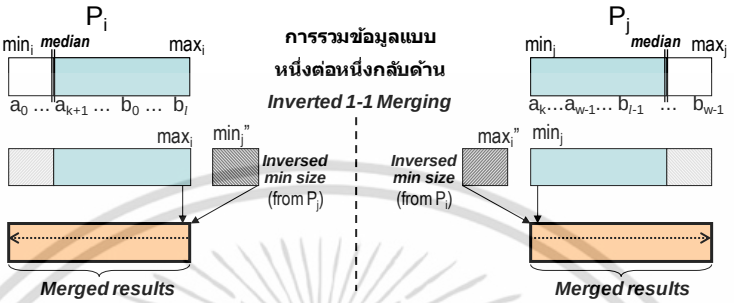
ตัวอย่างที่ 4.6 **แสดงการรวมข้อมูลกลับด้านแบบบล็อก**

สมมติให้ข้อมูลนำเข้าในหน่วยประมวลผล P_i {1, 2, 8, 9, 10, 11, 12} และหน่วยประมวลผล P_j {2, 3, 4, 5, 6, 13, 14} ชุดข้อมูลย่อยที่จำเป็นในหน่วยประมวลผล P_i {1, 2} และ P_j {13, 14} แลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผล จากนั้น P_i ตรวจสอบเงื่อนไข max_i (12) $\leq min_j$" (13) เป็นไปตามเงื่อนไขนำชุดข้อมูลย่อยที่ได้รับจากหน่วยประมวลผล P_j รวมกับชุดข้อมูลนำเข้า และในขณะเดียวกันหน่วยประมวลผล P_j ตรวจสอบเงื่อนไข max_i"(2) $\leq min_j$ (3) เป็นไปตามเงื่อนไขนำชุดข้อมูลย่อยที่ได้รับจากหน่วยประมวลผล P_i รวมในชุดข้อมูลนำเข้า ดังแสดงในรูปที่ 4.17 โดยนำค่า {13, 14} ต่อท้ายชุดข้อมูลนำเข้าในหน่วยประมวลผล P_i และนำค่า {1, 2} รวมในชุดข้อมูลนำเข้า สำหรับหน่วยประมวลผล P_j



รูปที่ 4.17 แสดงตัวอย่างการประยุกต์ใช้วิธีการรวมข้อมูลกลับด้านแบบบล็อก

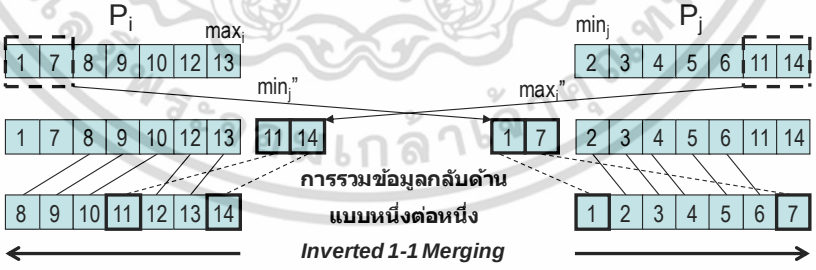
การรวมข้อมูลกลับด้านแบบหนึ่งต่อหนึ่ง หน่วยประมวลผล P_i เปรียบเทียบชุดข้อมูลที่ละค่าโดยเริ่มจากตำแหน่ง max_i เปรียบเทียบกับชุดข้อมูลย่อยที่จำเป็นเริ่มจาก min_j" ดังรูปที่ 4.18 ด้านซ้าย ในขณะเดียวกันหน่วยประมวลผล P_j เปรียบเทียบชุดข้อมูลที่ละค่าโดยเริ่มจากตำแหน่ง min_j เปรียบเทียบกับชุดข้อมูลย่อยที่จำเป็นเริ่มจาก max_i" ดังรูป 4.18 ด้านขวา จากนั้นหน่วยประมวลผลทั้งสองเปรียบเทียบข้อมูลทีละค่า



รูปที่ 4.18 แสดงการประยุกต์วิธีการรวมข้อมูลกลับด้านแบบหนึ่งต่อหนึ่ง (Inverted 1-1 Merging)

<u>ตัวอย่างที่ 4.7</u> **แสดงการรวมข้อมูลกลับด้านแบบหนึ่งต่อหนึ่ง**

สมมติให้ข้อมูลนำเข้าในหน่วยประมวลผล P_i {1, 7, 8, 9, 10, 12, 13} และหน่วยประมวลผล P_j {2, 3, 4, 5, 6, 11, 13} ชุดข้อมูลย่อยที่จำเป็นในหน่วยประมวลผล P_i {1, 7} และ P_j {11, 13} แลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผล จากนั้น P_i ตรวจสอบเงื่อนไข max_i (13) $\leq$ min_j" (11) ไม่เป็นไปตามเงื่อนไขดังนั้นเปรียบเทียบข้อมูลทีละค่าภายในหน่วยประมวลผล P_i ดังแสดงในรูป 4.19 ด้านซ้าย ในขณะเดียวกันหน่วยประมวลผล P_j ตรวจสอบเงื่อนไข max_i"(7) $\leq$ min_j (2) ไม่เป็นไปตามเงื่อนไขดังนั้นเปรียบเทียบข้อมูลทีละค่าภายในหน่วยประมวลผล P_j ดังแสดงในรูปที่ 4.19 ด้านขวา



รูปที่ 4.19 แสดงตัวอย่างการประยุกต์ใช้วิธีการรวมข้อมูลกลับด้านแบบหนึ่งต่อหนึ่ง

สำหรับเนื้อหาที่ได้นำเสนอในบทนี้เน้นเรื่องการแลกเปลี่ยนข้อมูลและการรวมข้อมูลที่มีประสิทธิภาพสูงซึ่งเป็นงานที่ผู้วิจัยได้นำเสนอไว้ กล่าวโดยสรุปมีประเด็นหลักดังนี้

1) การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูง เน้นการเพิ่มประสิทธิภาพในการแลกเปลี่ยนข้อมูลเป็นหลัก การนำค่ากลางมาช่วยในการคำนวณเพื่อให้ได้มาซึ่งชุดข้อมูลย่อยที่จำเป็น (จำนวนน้อยที่สุด) สำหรับการเรียงลำดับเท่านั้น นอกจากนี้ได้นำเสนอการแลกเปลี่ยนข้อมูลแบบกลับด้าน

ส่งผลให้ขนาดข้อมูลที่ใช้ในการแลกเปลี่ยนลดลงอย่างน้อย 50% เพราะลดขนาดข้อมูลที่แลกเปลี่ยนจาก ≤ w เป็น ≤ w/2

2) การรวมข้อมูลที่มีประสิทธิภาพสูง เน้นการเพิ่มประสิทธิภาพในการรวมข้อมูลโดยวิธีที่ได้นำเสนอขึ้นใหม่คือ การรวมข้อมูลแบบบล็อก ส่วนการรวมข้อมูลแบบหนึ่งต่อหนึ่ง มีการพัฒนาต่อยอดให้มีประสิทธิภาพเพิ่มขึ้นเช่นกัน

โดยประเด็นการวิจัยทั้ง 2 นี้ สรุปเป็นตารางเปรียบเทียบค่าความซับซ้อนด้านเวลาของการค้นหาช่วงของข้อมูลที่เหมาะสมสำหรับการแลกเปลี่ยนข้อมูลในแต่ละรูปแบบ ดังแสดงในตารางที่ 4.1 โดยนำการเรียงลำดับข้อมูลแบบดีซีอีเอส, แบบดีซีพีเอส และแบบโอบีเอส ที่ผู้วิจัยได้นำเสนอมาใช้ในการเปรียบเทียบ

ตารางที่ 4.1 แสดงการเปรียบเทียบค่าความซับซ้อนด้านเวลาของการค้นหาสำหรับการแลกเปลี่ยนข้อมูลในแต่ละรูปแบบ (Time Complexity of Searching Pattern)

ชนิดของการเรียงลำดับ	การค้นหา แบบคงที่	การค้นหา สลับที่เป็นคงที่	การค้นหา บางส่วน-1	การค้นหา บางส่วน-2
การเรียงแบบดีซีอีเอส (DCES)	O(1)	O(1)	$O(\log_2 N/P)$	$O(\log_2 N/P)$
การเรียงแบบดีซีพีเอส (DCPS)	O(1)	O(1)	$O(\log_2 N/P)$	$O(\log_2 N/P)$
การเรียงแบบโอบีเอส (OBS)	**O(1)**	**O(1)**	**O(1)**	**$O(\log_2 N/P)$**

บทที่ 5
การพิสูจน์ความถูกต้องและการวิเคราะห์ความซับซ้อนด้านเวลา

การนำเสนองานวิจัยทางวิชาการไม่ว่าเป็นการพัฒนาต่อยอดจากงานวิจัยเดิม หรือการคิดค้นวิธีการใหม่ๆ กระบวนการสำคัญที่ผู้วิจัยต้องดำเนินการเพื่อให้ผลงานวิจัยนั้นเป็นที่ยอมรับคือการพิสูจน์ความถูกต้อง (Proof of Correctness) [30] จะด้วยวิธีการพิสูจน์แบบใดนั้นขึ้นอยู่กับขั้นตอนวิธีที่นำเสนอในงานวิจัย นอกจากนี้การวิเคราะห์ความซับซ้อนด้านเวลา (Time Complexity Analyst) [29] ก็เป็นอีกปัจจัยหนึ่งช่วยส่งเสริมให้งานวิจัยที่นำเสนอมีความน่าเชื่อถือเพิ่มขึ้น และสามารถเปรียบเทียบกับวิธีการที่มีอยู่เดิมในงานวิจัยที่เกี่ยวข้องได้

สำหรับงานวิจัยที่ได้นำเสนอในวิทยานิพนธ์ฉบับนี้ มีหัวข้อในการพิสูจน์ความถูกต้องดังนี้

1) การพิสูจน์ความถูกต้องของการติดต่อสื่อสารแบบไดนามิค (Correctness of Dynamic Communication) โดยแบ่งเป็น 2 กรณี คือ 1) การพิสูจน์ความถูกต้องกรณีที่ดีที่สุด (Correctness of Best Case) และ 2) การพิสูจน์ความถูกต้องกรณีที่แย่ที่สุด (Correctness of Worst Case)
2) การพิสูจน์ความถูกต้องของการแลกเปลี่ยนข้อมูล (Correctness of Data Exchanging) แบ่งตามรูปแบบของการแลกเปลี่ยนข้อมูลเพื่อพิสูจน์ความถูกต้องได้เป็น 6 รูปแบบ คือ 1) การพิสูจน์ความถูกต้องแบบคงที่ (Correctness of Hold Pattern) 2) การพิสูจน์ความถูกต้องแบบสลับที่เป็นคงที่ (Correctness of Swap-to-Hold Pattern) 3) การพิสูจน์ความถูกต้องแบบบางส่วน-1 (Correctness of Partial-1 Pattern) 4) การพิสูจน์ความถูกต้องแบบบางส่วน-2 (Correctness of Partial-2 Pattern) 5) การพิสูจน์ความถูกต้องแบบคำนวณค่ามัธยฐาน (Correctness of Median Computing) และ 6) การพิสูจน์ความถูกต้องของข้อมูลแบบกลับด้าน (Correctness of Inverted Data Exchanging)

นอกจากการพิสูจน์ข้างต้นแล้วการวิเคราะห์ค่าความซับซ้อนด้านเวลา ก็เป็นอีกปัจจัยหนึ่งที่ช่วยส่งเสริมให้งานวิจัยมีคุณภาพมากขึ้น ผู้วิจัยได้แสดงการวิเคราะห์ค่าความซับซ้อนด้านเวลาแบ่งเป็น 3 ส่วน คือ 1) วิเคราะห์ความซับซ้อนด้านเวลาในฟังก์ชันการติดต่อสื่อสาร (Complexity of Communication Time) 2) วิเคราะห์ความซับซ้อนด้านเวลาในฟังก์ชันการแลกเปลี่ยนข้อมูล (Complexity of Data Exchanging) และ 3) วิเคราะห์ความซับซ้อนด้านเวลาทั้งหมดในการเรียงลำดับข้อมูลแบบโอบีเอส (Complexity of OBS) ซึ่งเป็นวิธีที่ดีที่สุดที่ได้นำเสนอในงานวิจัยนี้

5.1 การพิสูจน์ความถูกต้อง (Proof of Correctness)

จากที่ได้กล่าวมาแล้วข้างต้นการพิสูจน์ความถูกต้องในงานวิจัยนี้แบ่งเป็น 2 ฟังก์ชันหลักๆ คือ 1) การพิสูจน์ความถูกต้องของการติดต่อสื่อสารแบบไดนามิค 2) การพิสูจน์ความถูกต้องของการแลกเปลี่ยนข้อมูล ดังนี้

5.1.1 ความถูกต้องของการติดต่อสื่อสารแบบไดนามิค

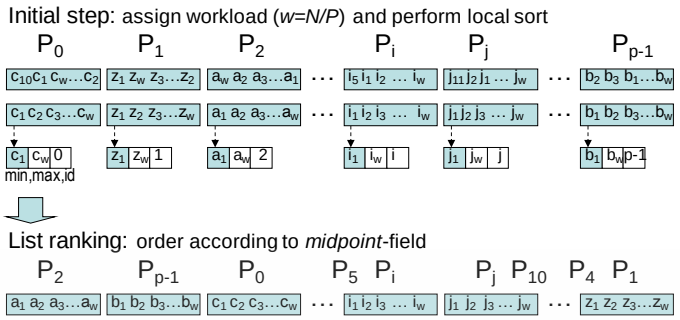
การพิสูจน์ความถูกต้องของการติดต่อสื่อสารแบบไดนามิค (Correctness of Dynamic Communication) นี้ เพื่อแสดงให้เห็นว่าจำนวนรอบในการติดต่อสื่อสารแบบไดนามิคนี้ขึ้นอยู่กับลักษณะของข้อมูลจริงที่ต้องการเรียงลำดับ เช่น ข้อมูลแบบสุ่ม (Random Data), ข้อมูลแบบเอนเอียง (Skew Data), ข้อมูลทั่วไป (Normal Data) เป็นต้น ซึ่งแบ่งการพิสูจน์ได้เป็น 2 กรณีคือ 1) การพิสูจน์ความถูกต้องกรณีที่ดีที่สุด และ 2) การพิสูจน์ความถูกต้องกรณีที่แย่ที่สุด มีรายละเอียดดังนี้

5.1.1.1 ความถูกต้องกรณีที่ดีที่สุด (Correctness of Best Case)

สำหรับการเรียงลำดับในกรณีที่ดีที่สุด แบ่งข้อมูลให้หน่วยประมวลผลขนาดเท่าๆ กัน หลังจากนั้นเรียงลำดับข้อมูลภายในแต่ละหน่วยประมวลผล และตรวจสอบการติดต่อสื่อสารด้วยรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูง (P-Table) ตัวอย่างการเรียงลำดับกรณีที่ดีที่สุด เช่น

- ข้อมูลนำเข้ามีการจัดเรียงไว้อย่างเหมาะสมแล้ว
- ข้อมูลนำเข้าย่อยมีการจัดเรียงในแต่ละหน่วยประมวลผลแล้ว เพียงแต่ลำดับของกลุ่มข้อมูลไม่ได้อยู่ในหน่วยประมวลผลที่เหมาะสม ตัวอย่างเช่นในรูปที่ 5.1 หลังจากทำงานในรอบแรกพบว่าข้อมูลย่อยถูกจัดเรียงไว้อย่างเหมาะสมแล้วในแต่ละหน่วยประมวลผล จากนั้นอัพเดทข้อมูลในตารางรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูง (P-Table) ขั้นตอนสุดท้ายก่อนจบกระบวนการทำงานตรวจสอบสถานะของหน่วยประมวลผล P_2 คู่กับ P_{p-1} , … , P_i คู่กับ P_j, … , P_4 คู่กับ P_1 ($P_2 \rightarrow P_{p-1}$, $P_i \rightarrow P_j$,…, $P_4 \rightarrow P_1$) มีสถานะของรูปแบบเป็นคงที่ (pattern-STATUS=HOLD) จากนั้นตรวจสอบสถานะของหน่วยประมวลผล P_{p-1} คู่กับ P_0, P_5 คู่กับ P_i, P_j คู่กับ P_{10}, … , และ P_k คู่กับ P_4 ($P_{p-1} \rightarrow P_0$, $P_5 \rightarrow P_i$, $P_{10} \rightarrow P_j$,…,$P_4 \rightarrow P_k$) โดยตรวจสอบสถานะเป็นคงที่สำหรับสถานะของลิสต์ (list-STATUS=HOLD) กล่าวคือไม่มีการแลกเปลี่ยนชุดข้อมูลระหว่างหน่วยประมวลผล จากตัวอย่างนี้เห็นได้ชัดเจนว่าการติดต่อสื่อสารแบบไดนามิคทำงานเพียง 1 รอบ สามารถจบกระบวนการทำงานได้ เมื่อเปรียบเทียบกับการติดต่อสื่อสารแบบสแตทิคต้องทำงานถึง $(\log_2 P(\log_2 P+1))/2$ รอบ เพื่อ

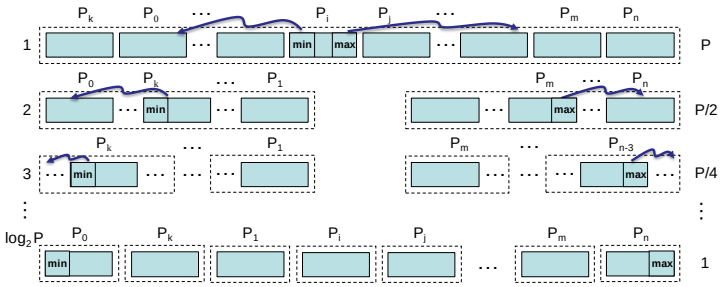
สลับข้อมูลให้อยู่ในหน่วยประมวลผลที่กำหนดไว้ โดยในกรณีสแตทิคนี้หน่วยประมวลผลที่มีเลขดัชนีน้อยเก็บชุดข้อมูลค่าน้อย ส่วนเลขดัชนีมากเก็บชุดข้อมูลค่ามาก



รูปที่ 5.1 แสดงตัวอย่างข้อมูลทั่วไปที่การเรียงด้วยค่ามิดพอยท์ เพียง 1 รอบ

5.1.1.2 ความถูกต้องกรณีที่แย่ที่สุด (Correctness of Worst Case)

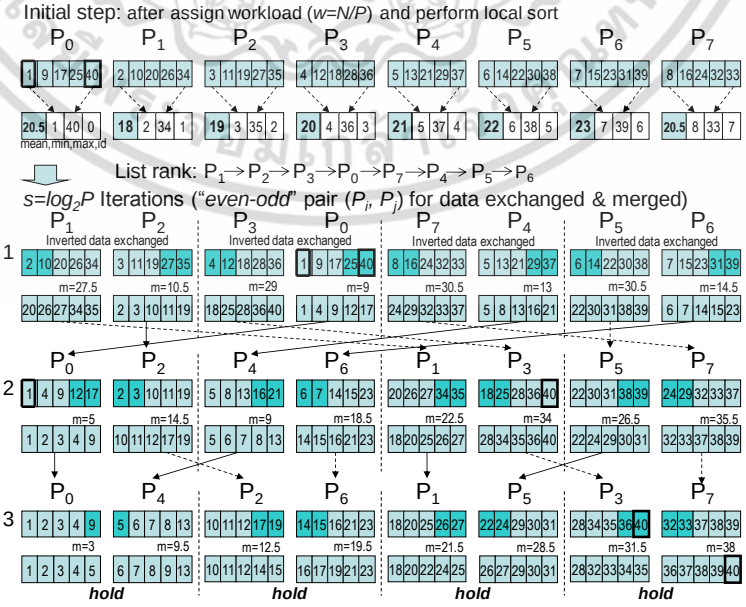
โดยทั่วไปข้อมูลส่วนใหญ่เป็นแบบสุ่ม ซึ่งอาจมีจำนวนรอบในการทำงานเพียง 1 หรือ 2 หรือ 3 หรือ, …, หรือ $\log_2 P$ รอบ (กรณีที่แย่ที่สุด) แต่สำหรับการเรียงชุดข้อมูลแบบเอนเอียงที่มีค่านอกกลุ่มแฝงอยู่ อาจต้องใช้ $\log_2 P$ รอบในการทำงาน หากการเรียงลำดับตัวแทนข้อมูลจากน้อยไปมากตามค่ามิดพอยท์ (แต่ถ้าใช้ค่าน้อยเป็นตัวแทนชุดข้อมูลกรณีดังกล่าวต้องใช้ $\log_2 P(\log_2 P +1)/2$ รอบ) โดยค่ามิดพอยท์นี้ช่วยให้สามารถนำค่านอกกลุ่มที่แฝงอยู่ออกจากกลุ่มดังกล่าวไปยังกลุ่มที่เหมาะสมได้ด้วยเทคนิค “Divide-and-Conquer” สำหรับกระบวนการเลือกคู่หน่วยประมวลผลตัวอย่างเช่น จากรูปที่ 5.2 แสดงการเรียงลำดับข้อมูลแบบเอนเอียงที่มีค่านอกกลุ่มแฝงอยู่กล่าวคือ แบ่งข้อมูลนำเข้าให้แต่ละหน่วยประมวลผลจำนวนเท่าๆ กัน จากนั้นเรียงลำดับข้อมูลภายในแต่ละหน่วยประมวลผล โดยข้อมูลชุดนี้มีลักษณะพิเศษคือ ข้อมูลในกลุ่มมีค่าส่วนใหญ่เป็นค่าน้อยแต่มีค่ามากแฝงตัวอยู่ ดังนั้นการเรียงลำดับตัวแทนชุดข้อมูลด้วยค่ามิดพอยท์ และประยุกต์ใช้เทคนิค “Divide-and-Conquer” โดยการทำงานในรอบแรกข้อมูลกลุ่มที่มีค่านอกกลุ่มแฝงอยู่จะถูกเรียงอยู่ตรงกลางๆ ภายในตารางการรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูง และในรอบการทำงานต่อๆ ไป ค่าดังกล่าวจะถูกย้ายไปอยู่ในกลุ่มย่อยที่ต้องการแลกเปลี่ยนข้อมูลระหว่างกันภายในขอบเขตของกลุ่มใหญ่ที่ต้องติดต่อสื่อสารตามลำดับดังนี้ P, P/2, P/4, P/8, …, 4, 2, 1 ซึ่งมีจำนวนรอบสูงสุดคือ $\log_2 P$ รอบ เป็นประเด็นหลักของการเรียงลำดับแบบขนานที่มีประสิทธิภาพสูงด้วยวิธีการติดต่อสื่อสารแบบไดนามิค ที่มีจำนวนรอบในการทำงานสูงสุด $\log_2 P$ รอบ โดยรอบที่ $\log_2 P+1$ ใช้ตรวจสอบสถานะเพื่อจบกระบวนการทำงาน ตามที่ได้กล่าวมาแล้วในบทที่ 3 รูปที่ 3.9



รูปที่ 5.2 แสดงตัวอย่างการเรียงลำดับข้อมูลแบบเอนเอียงที่มีค่านอกกลุ่มแฝงอยู่

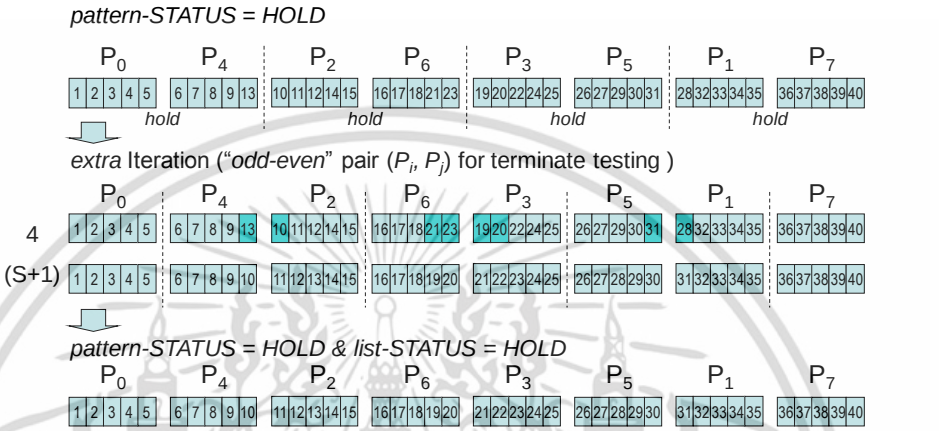
ตัวอย่างที่ 5.1 **แสดงตัวอย่างการพิสูจน์ความถูกต้องกรณีที่แย่ที่สุด**

สมมติให้ข้อมูลนำเข้าขนาด N=40 และหน่วยประมวลผลจำนวน P=8 แบ่งข้อมูลให้แต่ละหน่วยประมวลผลเท่าๆ กัน w=N(40)/P(8)=5 จะได้ว่า P_0 เก็บค่า {1, 9, 17, 25, 40}, P_1 เก็บค่า {2, 10, 20, 26, 34}, P_2 เก็บค่า {3, 11, 19, 27, 35}, P_3 เก็บค่า {4, 12, 18, 28, 36}, P_4 เก็บค่า {5, 13, 21, 29, 37}, P_5 เก็บค่า {6, 12, 22, 30, 38}, P_6 เก็บค่า {7, 15, 23, 31, 39} และ P_7 เก็บค่า {8, 16, 24, 32, 33} ดังแสดงในรูปที่ 5.3 จากข้อมูลเริ่มต้นพบกว่าลักษณะข้อมูลเป็นแบบเอนเอียงที่มีค่านอกกลุ่มแฝงอยู่ คือ 40 อยู่ในหน่วยประมวลผล P_0 ซึ่งควรอยู่ในหน่วยประมวลผล P_7 ข้อมูลลักษณะเช่นนี้ต้องใช้ $(\log_2 P+1) = 3+1 = 4$ รอบ ในการเรียงลำดับข้อมูลโดยแต่ละรอบการทำงานจะพิจารณาจากค่ามิดพอยท์ที่เรียงจากน้อยไปมากในรอบแรกลำดับของ list คือ $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_0 \rightarrow P_7 \rightarrow P_4 \rightarrow P_5 \rightarrow P_6$ ดังนั้นการจัดคู่ (P_i, P_j) ตาม list ranking คือ (P_1, P_2), (P_3, P_0), (P_7, P_4), (P_5, P_6) ในรอบ 2 ลำดับของ list คือ $P_0 \rightarrow P_2 \rightarrow P_4 \rightarrow P_6 \rightarrow P_1 \rightarrow P_3 \rightarrow P_5 \rightarrow P_7$ และจัด (P_i, P_j) ตาม list ranking คือ (P_0, P_2), (P_4, P_6), (P_1, P_3), (P_5, P_7) ส่วนในรอบ 3 ลำดับของ list คือ $P_0 \rightarrow P_4 \rightarrow P_2 \rightarrow P_6 \rightarrow P_1 \rightarrow P_5 \rightarrow P_3 \rightarrow P_7$ และจัด (P_i, P_j) ตาม list ranking คือ (P_0, P_4), (P_2, P_6), (P_1, P_5), (P_3, P_7) จากการประยุกต์ใช้เทคนิค "Divide-and-Conquer" ส่งผลให้ข้อมูลดังกล่าวจัดกลุ่มได้อย่างเหมาะสมภายใน $\log_2 P+1$ รอบ ดังรูปที่ 5.3-5.4



รูปที่ 5.3 แสดงตัวอย่างการพิสูจน์ความถูกต้องกรณีที่แย่ที่สุด (Worst Case)

จากรูปที่ 5.4 เมื่อทำงานรอบที่ 3 ตรวจสอบสถานะรูปแบบเป็นคงที่แล้ว ในรอบสุดท้าย ($\log_2$ P+1=4) ตรวจสอบสถานะของลิสต์เป็นแบบคงที่ ในกรณีที่ 2 ของทฤษฎีบทที่ 1 คือ มีการแลกเปลี่ยนข้อมูลในรอบการทำงานนี้ ดังนั้นขั้นตอนนี้แลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลอีกครั้งใน (P_i, P_j) ที่อยู่ตำแหน่ง odd-even ใน P-Table เพื่อเรียงลำดับข้อมูลก่อนสิ้นสุดกระบวนการทำงานจาก P-Table ตำแหน่ง 0-7 ($P_0 \rightarrow P_4 \rightarrow P_2 \rightarrow P_6 \rightarrow P_3 \rightarrow P_5 \rightarrow P_1 \rightarrow P_7$) โดยคู่ odd-even ของหน่วยประมวลผลคือ ($P_4 \rightarrow P_2$, $P_6 \rightarrow P_3$, $P_5 \rightarrow P_1$)



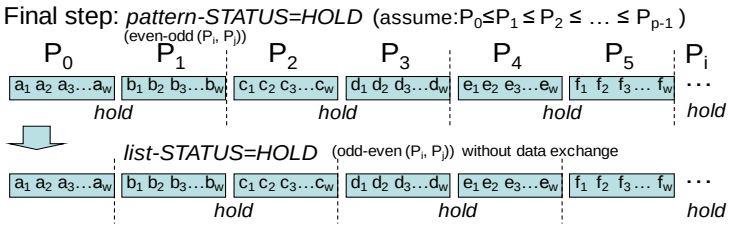
รูปที่ 5.4 แสดงตัวอย่างการตรวจสอบสถานะ pattern-STATUS & list-STATUS

สำหรับกระบวนการเรียงลำดับข้อมูลแบบขนานด้วยวิธีการติดต่อสื่อสารแบบไดนามิคที่นำเสนอจำเป็นต้องมีการพิสูจน์ความถูกต้องสำหรับเงื่อนไขเพื่อหยุดการทำงานในรอบที่เหมาะสม ซึ่งพิสูจน์ในทฤษฎีบท 1 คือ

ทฤษฎีบท 1: ถ้าสถานะของรูปแบบเป็นคงที่ (pattern-STATUS=HOLD) และสถานะของลิสต์เป็นคงที่ (list-STATUS=HOLD) ต่อเนื่องกันแล้ว แสดงว่าข้อมูลได้ถูกจัดเรียงไว้อย่างเหมาะสม

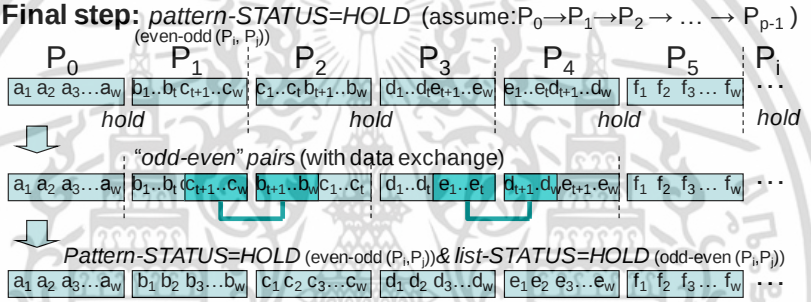
พิสูจน์ ในรอบการทำงานที่ $\log_2$ P สถานะของรูปแบบเป็นคงที่ (pattern-STATUS=HOLD) ให้ตรวจสอบสถานะของลิสต์เป็นคงที่ (list-STATUS=HOLD) เพื่อจบกระบวนการทำงาน ซึ่งแบ่งสถานะของลิสต์ ได้เป็น 2 กรณี

กรณีที่ 1 ไม่มีการแลกเปลี่ยนข้อมูล (Without Data Exchanging) เมื่อสถานะของลิสต์เท่ากับคงที่ (list-STATUS=HOLD) แสดงการพิสูจน์ในรูปที่ 5.5 ในรอบที่ $\log_2$ P สถานะของรูปแบบเป็นคงที่ (pattern-STATUS=HOLD) โดยการจับคู่ตำแหน่งของหน่วยประมวลผล (P_i, P_j) ในตาราง P-Table เป็นลักษณะแบบคู่คี่ (Even-Odd Pairs) กล่าวคือถ้าสมมติค่าใน P-Table คือ ($P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow \ldots \rightarrow P_i$) ดังนั้นการจัดคู่คือ ($P_0$, P_1), (P_2, P_3), (P_4, P_5), …, (P_i ,P_j) จากนั้นตรวจสอบต่อว่าสถานะของลิสต์ในรอบที่ $\log_2$ P+1 ซึ่งจับคู่หน่วยประมวลผลในตำแหน่งคี่คู่ (odd-even) กล่าวคือ (P_1, P_2), (P_3, P_4), … เพื่อแสดงให้เห็นว่าเมื่อทำการจับคู่หน่วยประมวลผลครบทุกๆ หน่วยประมวลผลแล้ว ($P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow, \ldots, \rightarrow P_i \rightarrow, \ldots$) ข้อมูลจะถูกจัดเรียงได้อย่างถูกต้องเหมาะสม



รูปที่ 5.5 แสดงตัวอย่างการตรวจสอบสถานะของลิสต์ กรณีที่ 1

<u>กรณีที่ 2</u> มีการแลกเปลี่ยนข้อมูล (With Data Exchanging) เมื่อสถานะของลิสต์เท่ากับคงที่ ดังแสดงการพิสูจน์ในรูปที่ 5.6 โดยในรอบการทำงานสุดท้าย ($\log_2 P+1$) การจับคู่หน่วยประมวลผลในตำแหน่งคี่คู่ กล่าวคือ ถ้า P-Table คือ $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow \ldots$ จะได้ $(P_1, P_2), (P_3, P_4), (\ldots)$ มีการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลอีกครั้ง ก่อนสิ้นสุดกระบวนการเรียงลำดับข้อมูล เมื่อพบว่าสถานะทั้งสองมีสถานะเป็นคงที่



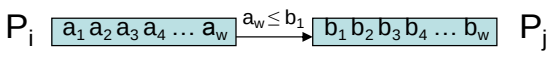
รูปที่ 5.6 แสดงตัวอย่างการตรวจสอบสถานะของลิสต์ กรณีที่ 2

5.1.2 ความถูกต้องของการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูง

การพิสูจน์การแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูง (Correctness of Data Exchanging) ที่ได้นำเสนอในงานวิจัยนี้มี 4 รูปแบบคือ แบบคงที่, แบบสลับที่เป็นคงที่, แบบบางส่วน-1 และแบบบางส่วน-2 รวมถึงการพิสูจน์ความถูกต้องสำหรับการคำนวณด้วยค่ามัธยฐาน (Median Computing) และการพิสูจน์ความถูกต้องสำหรับข้อมูลแบบกลับด้าน (Inverted Data) ว่ามีกระบวนการทำงานที่ถูกต้อง

5.1.2.1 ความถูกต้องแบบคงที่ (Correctness of Hold Pattern)

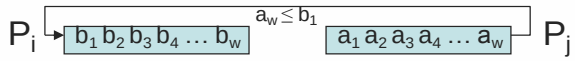
สำหรับรูปแบบการแลกเปลี่ยนข้อมูลแบบคงที่นี้เห็นได้ชัดเจนว่า เพียงตรวจสอบเงื่อนไขค่า $\min_i(a_w) \le \min_j(b_1)$ แล้ว ชุดข้อมูลในหน่วยประมวลผล P_i น้อยกว่าชุดข้อมูลหน่วยประมวลผล P_j ดังนั้นข้อมูลเรียงไว้อย่างเหมาะสมแล้ว ในรูปที่ 5.7



รูปที่ 5.7 แสดงตัวอย่างทั่วไปสำหรับการพิสูจน์ความถูกต้องแบบคงที่

5.1.2.2 ความถูกต้องแบบสลับที่เป็นคงที่ (Correctness of Swap-to-Hold Pattern)

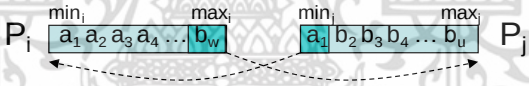
สำหรับการแลกเปลี่ยนข้อมูลแบบสลับที่นี้ เป็นการเปลี่ยนแปลงเพียงดัชนีของหน่วยประมวลผลจาก $P_i \rightarrow P_j$ เป็น $P_j \rightarrow P_i$ โดยไม่ต้องมีการแลกเปลี่ยนข้อมูลจริง เพียงตรวจสอบเงื่อนไขค่า $\max_j(a_w) \leq \min_i(b_w)$ แล้ว ชุดข้อมูลในหน่วยประมวลผล P_i เก็บค่ามากกว่าชุดข้อมูลในหน่วยประมวลผล P_j ดังนั้นเปลี่ยนแปลงเพียงค่าดัชนีของหน่วยประมวลผลในตาราง P-Table เท่านั้น ดังแสดงในรูปที่ 5.8



รูปที่ 5.8 แสดงตัวอย่างทั่วไปสำหรับการพิสูจน์ความถูกต้องแบบสลับที่เป็นคงที่

5.1.2.3 ความถูกต้องแบบบางส่วน-1 (Correctness of Partial-1 Pattern)

สำหรับการแลกเปลี่ยนข้อมูลแบบบางส่วน-1 เป็นการตรวจสอบเพื่อหาชุดข้อมูลย่อยเฉพาะที่จำเป็นที่คำนวณได้ภายในหนึ่งรอบ ตามเงื่อนไขในแต่ละหน่วยประมวลผล (P_i, P_j) ดังนี้ ใน P_i ถ้าค่า $\min_i(=a_1) \leq \min_j \leq$ สมาชิกต่อจาก $\min_i(=a_2)$ ส่วนใน P_j ถ้าค่า $\max_j \leq \max_i$ สมาชิกก่อน $\max_j \leq \max_i$ ดังนั้นชุดข้อมูลนำเข้าทั้งชุดคือชุดข้อมูลย่อยเฉพาะที่ตรวจสอบได้ ดังรูปที่ 5.9 และตัวอย่างที่ 4.1 ในบทที่ 4

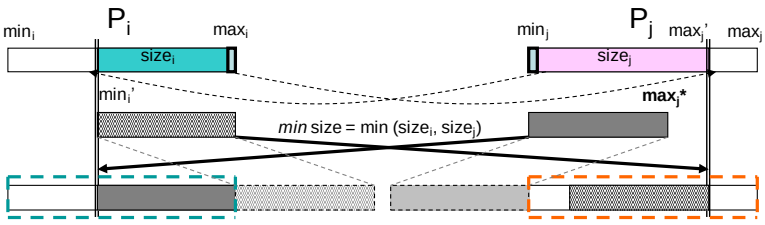


รูปที่ 5.9 แสดงตัวอย่างทั่วไปสำหรับการพิสูจน์ความถูกต้องแบบบางส่วน-1

5.1.2.4 ความถูกต้องแบบบางส่วน-2 (Correctness of Partial-2 Pattern)

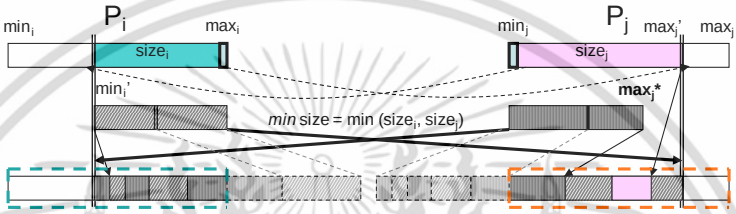
สำหรับการแลกเปลี่ยนข้อมูลแบบบางส่วน-2 เป็นการตรวจสอบเพื่อหาชุดข้อมูลย่อยโดยประยุกต์การค้นหาแบบไบนารี (Binary Search) เงื่อนไขการตรวจสอบและตัวอย่างดูได้จากบทที่ 4 ในหัวข้อ 4.1.4 ส่วนสำคัญของการพิสูจน์ความถูกต้องในรูปแบบนี้คือ ชุดข้อมูลย่อยที่หาได้ครอบคลุมเพียงพอหรือไม่ แบ่งได้เป็น 2 กรณีคือ

- กรณีการรวมชุดข้อมูลย่อยแบบบล็อก (Block Merging) เพื่อแสดงให้เห็นว่าชุดข้อมูลย่อยที่ได้ครอบคลุมเพียงพอสำหรับการเรียงลำดับข้อมูล เมื่อขนาดของข้อมูลที่แลกเปลี่ยนไม่เท่ากัน (สมมติว่า $\text{size}_i < \text{size}_j$) จากรูปที่ 5.10 ชุดข้อมูลย่อยของหน่วยประมวลผล P_i คือค่า $\min_i$' ถึงค่า $\max_i$ ส่วนในหน่วยประมวลผล P_j คือค่า $\min_j$ ถึงค่า $\max_i$' ซึ่งมีขนาดข้อมูลใหญ่กว่า ตรวจสอบข้อมูลใน P_j ชุดข้อมูลย่อยที่ต้องส่งคือ $\min_j$ ถึง $\max_i$* จากรูปแสดงอย่างชัดเจนว่าขนาดของข้อมูลที่จำเป็นต้องแลกเปลี่ยนเพื่อรวม คือ $\text{minimum}(\text{size}_i, \text{size}_j)$ จากนั้นรวมชุดข้อมูลย่อยทั้งสองกับชุดข้อมูลนำเข้าในแต่ละหน่วยประมวลผล



รูปที่ 5.10 แสดงตัวอย่างทั่วไปสำหรับการพิสูจน์ความถูกต้องแบบบางส่วน-2 กรณีที่ 1

- กรณีการรวมชุดข้อมูลย่อยแบบหนึ่งต่อหนึ่ง (1-1 Merging) เช่นเดียวกับกรณีที่ 1 แตกต่างตรงการรวมชุดข้อมูลย่อยกับชุดข้อมูลนำเข้ากระทำได้ทีละค่าเท่านั้น ดังแสดงในรูปที่ 5.11

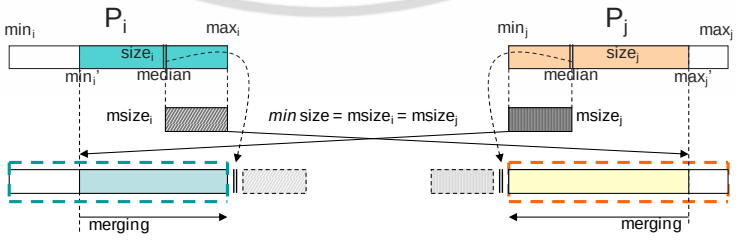


รูปที่ 5.11 แสดงตัวอย่างทั่วไปสำหรับการพิสูจน์ความถูกต้องแบบบางส่วน-2 กรณีที่ 2

5.1.2.5 ความถูกต้องของการคำนวณด้วยค่ามัธยฐาน (Correctness of Median Computing)

การได้มาซึ่งชุดข้อมูลย่อยที่จำเป็นจริงๆ นั้น สามารถคำนวณได้จากค่ามัธยฐาน ซึ่งได้แสดงรายละเอียดไว้ในบทที่ 4 ตัวอย่างที่ 4.2 การคำนวณด้วยค่ามัธยฐานเป็นค่าประมาณที่ใช้ค่ามัธยฐานทั้ง 4 ค่า ซึ่งประกอบไปด้วย ค่ามัธยฐานของชุดข้อมูลย่อยของ P_i และ P_j คือ (med_{i1}, med_{j1}) และค่ามัธยฐานของชุดข้อมูลนำเข้าของ P_i และ P_j คือ (med_{i2}, med_{j2}) เพื่อความถูกต้องแม่นยำครอบคลุมชุดข้อมูลย่อยที่จำเป็นในการเรียงลำดับ แบ่งได้เป็น 2 กรณี คือ

- กรณีขนาดข้อมูลเท่ากัน ($msize_i$=$msize_j$) ชุดข้อมูลย่อยทั้งสองมีขนาดเท่ากันเมื่อคำนวณด้วยค่ามัธยฐาน ซึ่งสามารถทำการรวมข้อมูลได้เลย เพราะค่ามัธยฐานในกรณีนี้อาจจะเป็นค่าที่แท้จริงหลังจากรวมข้อมูลแล้วดังรูปที่ 5.12 คือ median = (max_i+min_j)/2



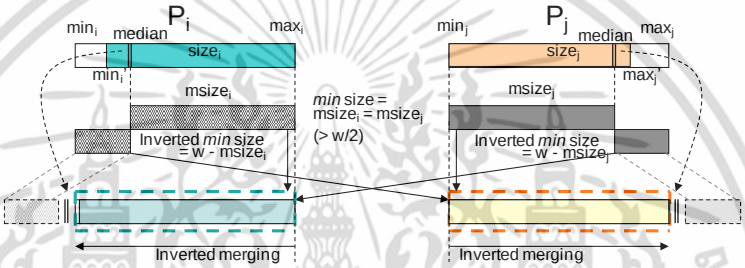
รูปที่ 5.12 แสดงตัวอย่างทั่วไปสำหรับการพิสูจน์ความถูกต้องคำนวณด้วยมัธยฐาน

- กรณีขนาดของข้อมูลไม่เท่ากัน ($msize_i$≠$msize_j$) หากชุดข้อมูลย่อยมีขนาดไม่เท่ากัน ถ้า $msize_i$<$msize_j$ แล้ว ค่ามัธยฐานที่คำนวณได้จะมีค่ามากกว่าค่ามัธยฐานจริง หรือถ้า $msize_i$>$msize_j$

แล้ว ค่ามัธยฐานที่คำนวณได้จะมีค่าน้อยกว่าค่ามัธยฐานจริง ส่งผลให้ชุดข้อมูลย่อยมีขนาดไม่เท่ากัน ดังนั้นเพื่อให้ครอบคลุมค่าที่ถูกต้องจึงส่งข้อมูลขนาดใหญ่ แต่ไม่เกินค่า minimum ($size_i$, $size_j$) ในหัวข้อ 5.1.2.4 หรือส่งข้อมูลขนาดน้อยกว่าระหว่าง max($msize_j$, $msize_i$) และ min($size_i$, $size_j$)

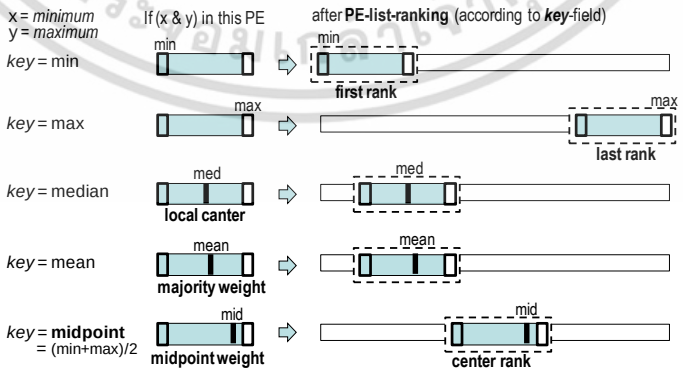
5.1.2.6 ความถูกต้องสำหรับข้อมูลแบบกลับด้าน (Correctness of Inverted Data Exchanging)

สำหรับการแลกเปลี่ยนข้อมูลลักษณะนี้เห็นได้อย่างชัดเจนว่าขนาดข้อมูลที่แลกเปลี่ยนลดลง หากข้อมูลมีขนาดใหญ่กว่า w/2 เมื่อ w=N/P ให้กลับด้านข้อมูลที่ต้องแลกเปลี่ยนเพราะหลังจากนั้นลำดับใน P-Table จะเปลี่ยนจาก $P_i \rightarrow P_j$ เป็น $P_j \rightarrow P_i$ (โดยมีรายละเอียดของการพิสูจน์แบบเดียวกับกรณีปกติในหัวข้อก่อนๆ) การกระทำในลักษณะนี้ส่งผลโดยตรงกับขนาดข้อมูลที่ลดลง ดังรูปที่ 5.13



รูปที่ 5.13 แสดงตัวอย่างทั่วไปในการพิสูจน์ความถูกต้องสำหรับข้อมูลแบบกลับด้าน

หมายเหตุ สำหรับงานวิจัยนี้คีย์สำคัญคือการเลือกคู่หน่วยประมวลได้เหมาะสมถูกที่ถูกเวลา โดยที่ในการเรียงกลุ่มของข้อมูลย่อย การได้มาซึ่งความเหมาะสมนี้ขึ้นอยู่กับคีย์ตัวแทนที่เลือกใช้ ดังนั้นเพื่อเป็นการพิสูจน์ให้เห็นว่าค่ามิดพอยท์เป็นค่าของคีย์ตัวแทนที่เหมาะสมสำหรับนำมาใช้ในงานวิจัยนี้ รูปที่ 5.14 แสดงผลกระทบของคีย์ที่สามารถเลือกใช้จำนวน 5 คีย์ และตารางที่ 5.1 แสดงเวลาที่ใช้ในการหาคีย์และรอบในการทำงานของคีย์ตัวแทนแต่ละชนิดสำหรับข้อมูลปกติ, ข้อมูลแบบเอนเอียง และข้อมูลแบบสุ่ม



รูปที่ 5.14 แสดงผลกระทบของคีย์ตัวแทนที่เลือกทั้ง 5 คีย์ (min, max, median, mean และ midpoint)

ตารางที่ 5.1 แสดงรอบในการทำงานสำหรับข้อมูลปกติ, ข้อมูลแบบเอนเอียง และข้อมูลแบบสุ่ม เมื่อเลือกใช้คีย์ตัวแทนแต่ละชนิด

ชนิดของคีย์ที่ใช้	เวลาของการคำนวณคีย์ตัวแทน	ข้อมูลปกติ แสดงผลลัพธ์แบบที่ดีที่สุด	ข้อมูลแบบเอนเอียง แสดงผลลัพธ์แบบที่แย่ที่สุด	ข้อมูลแบบสุ่ม แสดงผลลัพธ์แบบเฉลี่ย
ค่าน้อย (min key) [9]	O(1)	1	$\log_2 P(\log_2 P+1)/2$	1 ถึง $\log_2 P(\log_2 P+1)/2$
ค่ามาก (max key)	O(1)	1	$\log_2 P(\log_2 P+1)/2$	1 ถึง $\log_2 P(\log_2 P+1)/2$
ค่ามัธยฐาน (median key)	O(1)	1	$\log_2 P(\log_2 P+1)/2$	1 ถึง $\log_2 P(\log_2 P+1)/2$
ค่าเฉลี่ย (mean key)	O(N/P)	1	$\log_2 P(\log_2 P+1)/2$	1 ถึง $\log_2 P(\log_2 P+1)/2$
ค่ามิดพอยท์ (midpoint key)	O(1)	1	$\log_2 P$	1 ถึง $\log_2 P$

5.2 การวิเคราะห์ความซับซ้อนด้านเวลา (Time Complexity Analysis)

การวิเคราะห์ความซับซ้อนด้านเวลา (Time Complexity) สำหรับงานวิจัยที่ได้นำเสนอนี้เป็นอีกปัจจัยหนึ่งที่สนับสนุนให้งานวิจัยน่าสนใจ เพราะความซับซ้อนด้านเวลาบ่งชี้ถึงศักยภาพและประสิทธิภาพในการทำงานโดยเฉพาะลดรอบการแลกเปลี่ยนข้อมูล งานวิจัยนี้แบ่งการวิเคราะห์ค่าความซับซ้อนด้านเวลาเป็น 3 ส่วน คือ 1) วิเคราะห์ความซับซ้อนด้านเวลาในการติดต่อสื่อสาร 2) วิเคราะห์ความซับซ้อนด้านเวลาในการแลกเปลี่ยนข้อมูล และ 3) วิเคราะห์ความซับซ้อนด้านเวลาในการเรียงลำดับข้อมูลแบบโอบีเอส โดยมีรายละเอียดดังนี้

5.2.1 ความซับซ้อนด้านเวลาในการติดต่อสื่อสาร (Complexity of Communication Time)

จากการพิสูจน์ความถูกต้องในการติดต่อสื่อสารแบบไดนามิคระหว่าง 1 ถึง $\log_2 P$ รอบในหัวข้อที่ 5.1.1 แบ่งการวิเคราะห์ความซับซ้อนด้านเวลาเป็น 2 กรณี คือ

กรณีที่ดีที่สุด (Best Case) = O(1) × O(การประมวลผลในแต่ละรอบการทำงาน)

กรณีที่แย่ที่สุด (Worse Case) = $O(\log_2 P)$ × O(การประมวลผลในแต่ละรอบการทำงาน)

สำหรับฟังก์ชันหลักของเวลาการประมวลผลแต่ละรอบการทำงาน แจกแจงได้ดังตารางที่ 5.2

ตารางที่ 5.2 แสดงความซับซ้อนด้านเวลา (Time Complexity) ของการประมวลผลฟังก์ชันหลักในแต่ละรอบ

ฟังก์ชันหลักของการประมวลผล	ความซับซ้อนด้านเวลา Time Complexity
การอัพเดทตารางและการเลือกคู่หน่วยประมวลผลแบบไดนามิค	$O(\log_2 P)$
การแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผล (≤(N/P)/2)	O(N/P)
การรวมข้อมูลระหว่างหน่วยประมวลผล (≤(N/P)/2)	O(N/P)

หมายเหตุ เมื่อ N คือ จำนวนข้อมูล และ P คือ จำนวนหน่วยประมวลผล

5.2.2 ความซับซ้อนด้านเวลาในการแลกเปลี่ยนข้อมูล (Complexity of Data Exchanging)

การแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลตั้งแต่เริ่มกระบวนการทำงาน จนสิ้นสุดกระบวนการทำงานมีขั้นตอนและค่าความซับซ้อนด้านเวลาในแต่ละขั้นตอน ดังตารางที่ 5.3

ตารางที่ 5.3 แสดงค่าความซับซ้อนด้านเวลา (Time Complexity) ของฟังก์ชันย่อยๆ ในการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผล

ขั้นตอนการประมวลผล	ความซับซ้อนด้านเวลา Time Complexity
ส่งค่ามิดพอยท์ ค่าน้อย ค่ามาก และดัชนีหน่วยประมวลผล (mid, min, max, id)	O(1)
เรียงลำดับตัวแทนชุดข้อมูลในตารางด้วยค่ามิดพอยท์ โดยวิธีไบโทนิค	$O((\log_2 P)^2)$
ตรวจสอบสถานะเพื่อจบกระบวนการทำงาน (pattern & list-STATUS)	$O(\log_2 P)$
ตรวจสอบรูปแบบในการแลกเปลี่ยนข้อมูลแบบคงที่ (hold pattern)	O(1)
ตรวจสอบรูปแบบในการแลกเปลี่ยนข้อมูลแบบสลับที่ (swap pattern)	O(1)
ตรวจสอบรูปแบบในการแลกเปลี่ยนข้อมูลแบบบางส่วน-1 (partial-1 pattern)	O(1)
ตรวจสอบรูปแบบในการแลกเปลี่ยนข้อมูลแบบบางส่วน-2 (partial-2 pattern)	$O(\log_2 N/P)$
คำนวณค่ากลาง (median computing)	O(1)
แลกเปลี่ยนขนาดชุดข้อมูลย่อย (size, msize)	O(1)
แลกเปลี่ยนชุดข้อมูลย่อยที่จำเป็นเพื่อใช้เรียงลำดับข้อมูลแบบคงที่ (hold)	O(1)
หรือแลกเปลี่ยนชุดข้อมูลย่อยที่จำเป็นเพื่อใช้เรียงลำดับข้อมูลแบบสลับที่ (swap)	O(1)
หรือแลกเปลี่ยนชุดข้อมูลย่อยที่จำเป็นเพื่อใช้เรียงลำดับข้อมูลแบบบางส่วน(partial)	O(N/P)
การรวมชุดข้อมูลย่อยในชุดข้อมูลนำเข้าแบบบล็อก (block merge)	O(1)
การรวมชุดข้อมูลย่อยในชุดข้อมูลนำเข้าแบบหนึ่งต่อหนึ่ง (1-1 merge)	O(N/P)

5.2.3 ความซับซ้อนด้านเวลาในการเรียงลำดับข้อมูลแบบโอบีเอส

สำหรับการเรียงลำดับแบบโอบีเอสเวลาที่ใช้ในการประมวลผลทั้งหมด ประกอบด้วย เวลาในการเรียงลำดับ 2 ส่วนใหญ่ๆ คือ

การเรียงลำดับข้อมูลย่อย (N/P) ในแต่ละหน่วยประมวลผล (Local Sort) +
การเรียงลำดับแบบขนานในขั้นตอนวิธีแบบโอบีเอส (OBS)

ตารางที่ 5.4 แสดงความซับซ้อนด้านเวลาของวิธีโอบีเอส (OBS) เปรียบเทียบกับวิธีอื่นๆ (Bitonic [2][9][15], DCES[25], DCPS) ในกรณีที่ดีที่สุด (Best Case) กรณีที่ดีที่สุดลำดับสอง (Second Best Case) กรณีที่แย่ที่สุด (Worst Case) และกรณีเฉลี่ย (Average Case) โดยทุกวิธีดังกล่าวใช้การเรียงลำดับแบบเร็ว (Quick Sort) ในการประมวลผลเบื้องต้นด้วยความซับซ้อนด้านเวลาเท่ากับ $O(N/P \log_2 N/P)$

ตารางที่ 5.4 แสดงการเปรียบเทียบการเรียงลำดับแบบโอบีเอส, แบบดีซีอีเอส และแบบสแตทิค เมื่อใช้การเรียงลำดับข้อมูลเริ่มต้นในแต่ละหน่วยประมวลผลแบบเร็ว (Quick Sort)

ขั้นตอนวิธี	แบบโอบีเอส Dynamic (OBS)	แบบโบโทนิค Static (Bitonic)	แบบดีซีอีเอส Dynamic (DECS)	แบบดีซีพีเอส Dynamic (DCPS)
แบบที่ดีที่สุด	$O(N/P \log_2 N/P) + O((\log_2 P)^2)$	$O(N/P \log_2 N/P) + O((\log_2 P)^2)$	$O(N/P \log_2 N/P) + O(P)$	$O(\log_2 N/P) + O(P)$
แบบที่ดีที่สุดลำดับที่สอง	$O(N/P \log_2 N/P) + O((\log_2 P)^2)$	$O(N/P \log_2 N/P) + O((\log_2 P)^2 \times N/P)$	$O(N/P \log_2 N/P) + O(P)$	$O(N/P \log_2 N/P) + O(P)$
แบบที่แย่ที่สุด	$O(N/P \log_2 N/P) + O(\log_2 P \times N/P)$	$O(N/P \log_2 N/P) + O((\log_2 P)^2 \times N/P)$	$O(N/P \log_2 N/P) + O((\log_2 P)^2 \times N/P)$	$O(N/P \log_2 N/P) + O((\log_2 P)^2 \times N/P)$
แบบเฉลี่ย	$O(N/P \log_2 N/P) + O(\log_2 P \times N/P)$	$O(N/P \log_2 N/P) + O((\log_2 P)^2 \times N/P)$	$O(N/P \log_2 N/P) + O((\log_2 P)^2 \times N/P)$	$O(N/P \log_2 N/P) + O((\log_2 P)^2 \times N/P)$

ส่วนตารางที่ 5.5 แสดงการเปรียบเทียบความซับซ้อนด้านเวลาของวิธีโอบีเอส (OBS) เทียบกับวิธีอื่นๆ เมื่อทุกวิธีดังกล่าวใช้การเรียงลำดับแบบเรดิกซ์ (Radix Sort) ในการประมวลผลเบื้องต้นด้วยความซับซ้อนด้านเวลา O(N/P)

ตารางที่ 5.5 แสดงการเปรียบเทียบการเรียงลำดับแบบโอบีเอส, แบบดีซีอีเอส และแบบสแตทิค เมื่อใช้การเรียงลำดับข้อมูลเริ่มต้นในแต่ละหน่วยประมวลผลแบบเรดิกซ์ (Radix Sort)

ขั้นตอนวิธี	แบบโอบีเอส Dynamic (OBS)	แบบโบโทนิค Static (Bitonic)	แบบดีซีอีเอส Dynamic (DECS)	แบบดีซีพีเอส Dynamic (DCPS)
แบบที่ดีที่สุด	$O(N/P)$ $+ O((\log_2 P)^2)$	$O(N/P)$ $+ O((\log_2 P)^2)$	$O(N/P)$ $+ O(P)$	$O(N/P)$ $+ O(P)$
แบบที่ดีที่สุด ลำดับที่สอง	$O(N/P)$ $+ O((\log_2 P)^2)$	$O(N/P)$ $+ O((\log_2 P)^2 \times N/P)$	$O(N/P)$ $+ O(P)$	$O(N/P)$ $+ O(P)$
แบบที่แย่ที่สุด	$O(N/P)$ $+ O(\log_2 P \times N/P)$	$O(N/P)$ $+ O((\log_2 P)^2 \times N/P)$	$O(N/P)$ $+ O((\log_2 P)^2 \times N/P)$	$O(N/P)$ $+ O((\log_2 P)^2 \times N/P)$
แบบเฉลี่ย	$O(N/P)$ $+ O(\log_2 P \times N/P)$	$O(N/P)$ $+ O((\log_2 P)^2 \times N/P)$	$O(N/P)$ $+ O((\log_2 P)^2 \times N/P)$	$O(N/P)$ $+ O((\log_2 P)^2 \times N/P)$

<u>หมายเหตุ</u> กรณีที่ P=N จะมีค่าความซับซ้อนด้านเวลาดีที่สุด $O((\log_2 N)^2)$ เช่นเดียวกับการเรียงลำดับข้อมูลแบบไบโทนิค (Bitonic Sort)

บทที่ 6

การทดลองและผลการทดลอง

การประมวลผลแบบขนาน (Parallel Processing) เป็นการประมวลผลสำหรับงานที่มีขนาดใหญ่โดยใช้หลายๆ หน่วยประมวลผลเพื่อลดเวลาในการประมวลผล โดยปกติการประมวลผลแบบขนานจะมีความซับซ้อนกว่าการประมวลผลแบบอนุกรม (Sequential Processing) เนื่องจากเวลาที่ใช้ในการประมวลผลแบบขนานทั้งหมด ประกอบด้วยเวลาที่ใช้ในการติดต่อสื่อสารระหว่างหน่วยประมวลผล และเวลาที่ใช้ในการประมวลผล ถึงแม้ว่าการประมวลผลแบบขนานจะมีความซับซ้อนกว่าการประมวลผลทั่วๆไปที่เป็นแบบอนุกรม แต่การประมวลผลแบบขนานมีบทบาทสำคัญต่อการประมวลผลสำหรับงานขนาดใหญ่ที่ใช้เวลานาน ดังนั้นวิธีการประมวลผลแบบขนานที่มีประสิทธิภาพจึงถูกเสนอขึ้น ทั้งขั้นตอนวิธีแบบขนานและการประยุกต์ใช้งานในหลายๆ ด้าน เช่น การเรียงลำดับแบบขนาน การคูณเมตริกซ์แบบขนาน เป็นต้น

งานวิจัยนี้เน้นเรื่องขั้นตอนวิธีการเรียงลำดับแบบขนาน ดังนั้นจึงขอกล่าวถึงการออกแบบและการเปรียบเทียบประสิทธิภาพของระบบ (System Performance) ในการเรียงลำดับแบบขนาน ซึ่งสามารถจำแนกได้เป็น 3 กลุ่มหลักๆ คือ

1) การออกแบบระบบคอมพิวเตอร์แบบขนานเพื่อรองรับการเรียงลำดับข้อมูลแบบขนาน ซึ่งการทำงานโดยทั่วไปนั้น จำนวนหน่วยประมวลผล (P) น้อยกว่าจำนวนข้อมูล (N) หรือ $P < N$ ดังนั้นการวัดประสิทธิภาพเพื่อเปรียบเทียบการทำงานต้องคำนึงถึงการแบ่งงานให้แต่ละหน่วยประมวลผลในปริมาณเท่าๆ กัน

2) การออกแบบขั้นตอนวิธีการเรียงลำดับแบบขนานบนระบบคอมพิวเตอร์แบบขนานที่มีอยู่หรือถูกสร้างขึ้นมาแล้วและสามารถประมวลผล หรือทำการทดลองได้จริง เช่น ระบบมัลติคอร์ (Multi-cores System), ระบบคลัสเตอร์ (Cluster System), คอมพิวเตอร์แบบกริด (Grid Computer) และระบบคอมพิวเตอร์แบบ SMP เป็นต้น เนื่องจากระบบดังกล่าวเป็นระบบที่มีใช้อย่างกว้างขวาง ดังนั้นการวัดประสิทธิภาพ ส่วนใหญ่จะวัดจากเวลาที่ใช้ในการประมวลผลจริง (Response Time) บนระบบต่างๆ ดังกล่าว

3) การเสนอขั้นตอนวิธีการเรียงลำดับแบบขนานแบบใหม่ ซึ่งเป็นการปรับปรุงจากแบบที่มีอยู่เดิมให้มีประสิทธิภาพมากขึ้น โดยส่วนใหญ่จะเป็นการเพิ่มประสิทธิภาพด้านเวลาที่ใช้ในการติดต่อสื่อสารระหว่างหน่วยประมวลผล

วิทยานิพนธ์นี้เสนอการออกแบบขั้นตอนวิธีการเรียงลำดับแบบขนานที่มีประสิทธิภาพสูง โดยวิธีต่างๆ ที่เสนอเป็นวิธีใหม่ที่ปรับปรุงประสิทธิภาพทั้งด้านเวลาที่ใช้ในการติดต่อสื่อสาร และเวลาที่ใช้ในการประมวลผล นอกจากนี้ลดการใช้เนื้อที่ในหน่วยความจำที่ไม่จำเป็น ดังแสดง

รายละเอียดในบทที่ 3 และ 4 ส่วนเนื้อหาในบทนี้ นำเสนอผลการทดลองจากการพัฒนาโปรแกรมการเรียงลำดับแบบขนานที่นำเสนอแบบแอลบีเอ็ม, แบบดีซีอีเอส, แบบดีซีพีเอส และแบบโอบีเอส โดยประมวลผลบนระบบมัลติคอร์เป็นระบบที่ได้รับความสนใจเป็นอย่างมากในปัจจุบัน เนื่องจากมีราคาไม่แพง และมีขั้นตอนการพัฒนาที่ไม่ยุ่งยากซับซ้อนมากจนเกินไป

6.1 เครื่องมือที่ใช้ในการทดลอง

ระบบมัลติคอร์ที่ใช้ในการทดลอง ทำงานบนระบบปฏิบัติการวินโดว์ 7 (Windows 7) ขณะทำการทดลองระบบมัลติคอร์ไม่มีภาระงานอื่นใดประมวลผลอยู่ในระบบ นอกจากการประมวลผลการเรียงลำดับข้อมูลแบบขนานเท่านั้น โดยมีส่วนประกอบพื้นฐานของระบบซึ่งประกอบด้วยส่วนประกอบด้านฮาร์ดแวร์ ระบบปฏิบัติการและซอฟท์แวร์ ที่ทำงานในระดับของแกน (Kernel) ของระบบปฏิบัติการ และสุดท้ายคือการเชื่อมต่อแต่ละคอร์เพื่อแลกเปลี่ยนข้อมูลผ่านทางเครือข่ายภายในความเร็วสูง

1) ส่วนประกอบด้านฮาร์ดแวร์

โครงสร้างทางฮาร์ดแวร์ของระบบที่ใช้ในการทดลอง คือระบบมัลติคอร์โดยขนาดของหน่วยประมวลผล หน่วยความจำ และส่วนประกอบอื่นๆ เหมือนกัน คือเครื่องคอมพิวเตอร์ส่วนบุคคล (Personal Computer: PC) ทั่วๆ ไป เพื่อแสดงให้เห็นถึงความง่ายในการประยุกต์ใช้งานสำหรับการเรียงลำดับแบบขนานที่ผู้วิจัยได้นำเสนอในวิทยานิพนธ์นี้ ซึ่งมีองค์ประกอบด้านฮาร์ดแวร์ที่จำเป็น ดังนี้

ก) แผงวงจรรวม (Mother Board) ยี่ห้อ Gigabyte รุ่น GA-970A-D3

ข) หน่วยประมวลผล (CPUs) ในงานวิจัยนี้ใช้เครื่องคอมพิวเตอร์แบบแปดคอร์ในเครื่องเดียวกัน (8-Cores) ยี่ห้อเอเอ็มดี (AMD-FX-8120) ความเร็วต่อรอบ 3.1 กิกะเฮิร์ต (GHz) มีแคช L1 ขนาด 128 กิโลไบต์ (Kbyte), แคช L2 และแคช L3 ขนาด 8 เมกะไบต์ (Mbyte)

ค) หน่วยความจำหลัก (RAM) ขนาด 8 (4x2) กิกะไบต์ (Gbyte)

ง) หน่วยความจำสำรอง (Disk Storage) ขนาด 1 เทราไบต์ (Tbyte)

จ) เชื่อมต่อผ่านเครือข่ายภายในแผงวงจรรวม 1600 เมกกะบิตต่อวินาที (Mbps)

2) ส่วนประกอบด้านซอฟต์แวร์

ระบบปฏิบัติการ (Operating System: OS) มีส่วนสำคัญอย่างมากต่อการทำงานของคอมพิวเตอร์เพราะในระบบมัลติคอร์ต้องสามารถทำงานเองได้โดยอิสระไม่ขึ้นกับเงื่อนไขการทำงานของคอร์อื่น ระบบปฏิบัติการ จึงมีหน้าที่หลักในการบริหารจัดคิวการทำงานให้แต่ละคอร์ทำงานได้อย่างอิสระต่อกัน และสามารถติดต่อสื่อสารกันได้หากต้องการ ระบบปฏิบัติการที่รองรับการทำงานแบบระบบมัลติคอร์ เช่น Windows, Linux, UNIX เป็นต้น[28]

โดยส่วนใหญ่ซอฟต์แวร์และชุดคำสั่งที่ถูกพัฒนาขึ้นมาเพื่อใช้ในระบบมัลติคอร์นี้จะทำงานเข้ากันได้กับระบบปฏิบัติการเกือบทุกชนิดเนื่องจากเน้นใช้งานง่ายและผู้ใช้ทั่วไปสามารถใช้งานได้

3) เอ็มพีไอซีเอชสอง (MPICH2)

เอ็มพีไอซีเอชสอง [6][8][18] เป็นการพัฒนาชุดคำสั่งตามมาตรฐานขึ้นมาใช้งานจริงมากที่สุดคือ มาตรฐานการส่งผ่านข้อความ (Message Passing Interface: MPI) ซึ่งสามารถพัฒนาโปรแกรมภาษาต่างๆ เช่น โปรแกรมภาษาซี โปรแกรมภาษาฟอร์แทน และโปรแกรมภาษาจาวา เป็นต้น ที่สนับสนุนการโปรแกรมแบบขนาน ซึ่งมีฟังก์ชันพื้นฐานต่างๆ รวมทั้งมีคำสั่งใช้งานเพื่อทำการคอมไพล์โปรแกรมที่เขียนขึ้น

6.2 ลักษณะข้อมูล การเลือกข้อมูลและเหตุผลการเลือก

ลักษณะข้อมูลที่ใช้ในการทดลองเป็นข้อมูลชนิดตัวเลข (Number) แบ่งข้อมูลเป็น 3 แบบ คือ 1) ข้อมูลแบบสุ่ม (Random Data) 2) ข้อมูลแบบเอนเอียงด้านซ้าย (Left Skewed-Data) และ 3) ข้อมูลแบบเอนเอียงด้านขวา (Right Skewed-Data) โดยข้อมูลแต่ละแบบอยู่ในช่วง 0-100,000,000 ซึ่งจำนวนข้อมูลที่ใช้ในการทดลองมีจำนวนตั้งแต่ 10-100 ล้านข้อมูล เนื่องจากข้อมูลจำนวนนี้แสดงให้เห็นถึงความแตกต่างในแต่ละวิธีที่นำเสนอได้อย่างชัดเจน ถ้าใช้ข้อมูลที่มีจำนวนน้อยกว่าที่ได้กล่าวมา เช่น 1 แสนข้อมูล ไม่สามารถแสดงให้เห็นถึงความแตกต่างในแต่ละวิธีได้เด่นชัดมากนัก

ผู้วิจัยทำการเลือกข้อมูลชนิดตัวเลข เนื่องจากมีความใกล้เคียงกับการใช้งานในชีวิตประจำวันอย่างยิ่ง เช่น การเรียงลำดับของเลขที่บัญชีธนาคาร เลขที่บัตรประจำตัวประชาชน เลขที่บัตรประกันสังคม การสร้างดัชนีแฟ้มข้อมูล เป็นต้น ซึ่งในงานวิจัยนี้ (การเรียงลำดับ) ข้อมูลจะถูกจัดเก็บอยู่ในคอมพิวเตอร์ที่มีประสิทธิภาพและมีความสามารถในการประมวลผลงาน

6.3 ผลการทดลอง

การวัดประสิทธิภาพของการเรียงลำดับข้อมูลนี้จะประเมินผลจากเวลาที่ใช้ในการเรียงลำดับข้อมูล (Response Time) อัตราการเพิ่มขึ้นของความเร็ว (Speedup) และประสิทธิภาพ (Efficiency) โดยในผลการทดลองนี้จะทำการเปรียบเทียบประสิทธิภาพของการเรียงลำดับข้อมูลที่ได้นำเสนอทั้ง 4 แบบ โดยแบบแรกคืองานวิจัยเดิมที่มีอยู่ในปัจจุบัน ส่วนสามแบบหลังเป็นงานวิจัยที่ได้นำเสนอในวิทยานิพนธ์นี้ คือ

1) การเรียงลำดับแบบแอลบีเอ็ม (LBM: Load-Balanced Merge Sort) [12]

2) การเรียงลำดับแบบดีซีอีเอส (DCES: Dynamic Communication Efficient Sort) [25]

3) การเรียงลำดับแบบดีซีพีเอส (DCPS: Dynamic Communication Parallel Sort)

4) การเรียงลำดับแบบโอบีเอส (OBS: Optimized Bitonic Sort) [26]

โดยผลการทดลองในงานวิจัยนี้ทำการเปรียบเทียบการเรียงลำดับทั้ง 4 แบบ ในหลายมิติรอบด้านเพื่อให้ครอบคลุมทุกมุมมอง ดังนี้

1) ผลการทดลองเปรียบเทียบกับเวลาในอุดมคติ

2) ผลการทดลองเปรียบเทียบเมื่อขนาดข้อมูลที่ใช้ทดลองต่างกัน

3) ผลการทดลองเปรียบเทียบเมื่อจำนวนหน่วยประมวลผลที่ใช้ทดลองต่างกัน

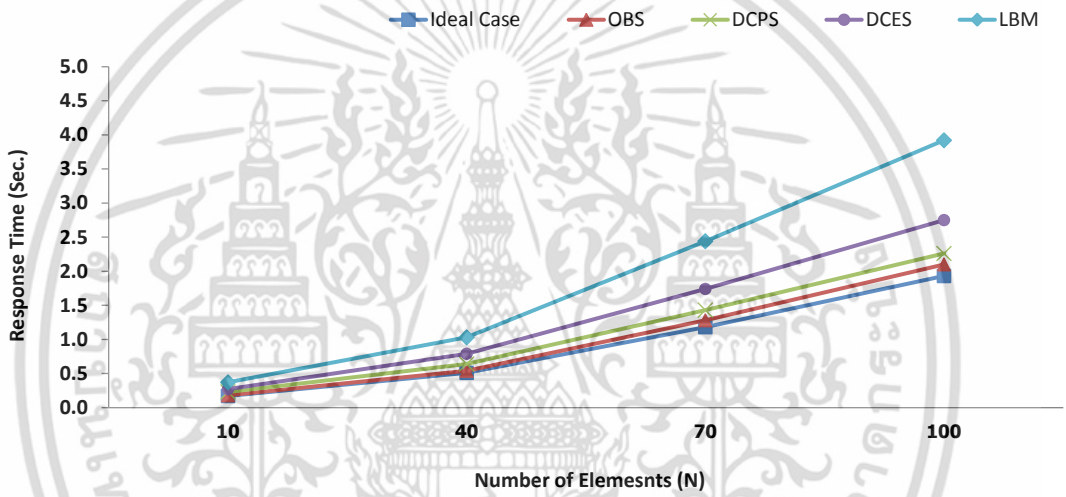
4) ผลการทดลองเปรียบเทียบจำนวนรอบในการเรียงลำดับข้อมูล

6.3.1 ผลการทดลองเปรียบเทียบกับเวลาในอุดมคติ

การทดลองเปรียบเทียบเวลาที่ประมวลผลจริงกับเวลาในอุดมคติ เป็นการวัดประสิทธิภาพของระบบที่ใช้ในการทดลองดังนี้ ตารางที่ 6.1 และรูปที่ 6.1 แสดงการเปรียบเทียบเวลาที่ใช้ในการเรียงลำดับข้อมูลแบบสุ่ม (Random Data) ทั้ง 4 แบบ ($T_P=T_S/P +T_C$) โดยในงานวิจัยนี้เน้นการเพิ่มประสิทธิภาพของเวลาที่ใช้ในการติดต่อสื่อสาร (Communication Time: T_C) และเวลาที่ใช้ในการประมวลผล (Computation Time: T_p) โดยเปรียบเทียบกับเวลาที่ใช้ในการเรียงลำดับข้อมูลในอุดมคติ (Response Time of Ideal Case) คำนวณโดยสมมติว่าเวลาที่ใช้ในการติดต่อสื่อสาร เพื่อแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลเท่ากับศูนย์ ($T_C=0$) ดังนั้น $T_P = T_S/P$ ตามสมการที่ 2.5 ส่วนตารางที่ 6.2 และรูปที่ 6.2 แสดงการเปรียบเทียบเวลาที่ใช้ในการเรียงลำดับข้อมูลแบบเอนเอียงด้านซ้าย (Left Skewed-Data) ทั้ง 4 แบบ เปรียบเทียบกับเวลาในอุดมคติ และในตารางที่ 6.3 และรูปที่ 6.3 แสดงการเปรียบเทียบเวลาที่ใช้ในการเรียงลำดับข้อมูลแบบเอนเอียงด้านขวา (Right Skewed-Data) ทั้ง 4 แบบ เปรียบเทียบกับเวลาในอุดมคติ

ตารางที่ 6.1 เวลาที่ใช้ในการเรียงลำดับข้อมูล (Response Time) ของ 4 แบบโดยเปรียบเทียบกับเวลาที่ใช้ในการเรียงลำดับข้อมูลในอุดมคติ (Response Time of Ideal Case) เมื่อข้อมูลเป็นแบบสุ่มมีขนาดตั้งแต่ 10, 40, 70, 100 ล้านข้อมูล 8 หน่วยประมวลผล

วิธีการ/จำนวนข้อมูล	10	40	70	100
เวลาในอุดมคติ	0.17	0.51	1.18	1.93
เวลาในการเรียงลำดับแบบ “LBM”	0.37	1.03	2.44	3.92
เวลาในการเรียงลำดับแบบ “DCES”	0.27	0.79	1.74	2.75
เวลาในการเรียงลำดับแบบ “DCPS”	0.23	0.64	1.43	2.26
เวลาในการเรียงลำดับแบบ “OBS”	0.18	0.54	1.28	2.10

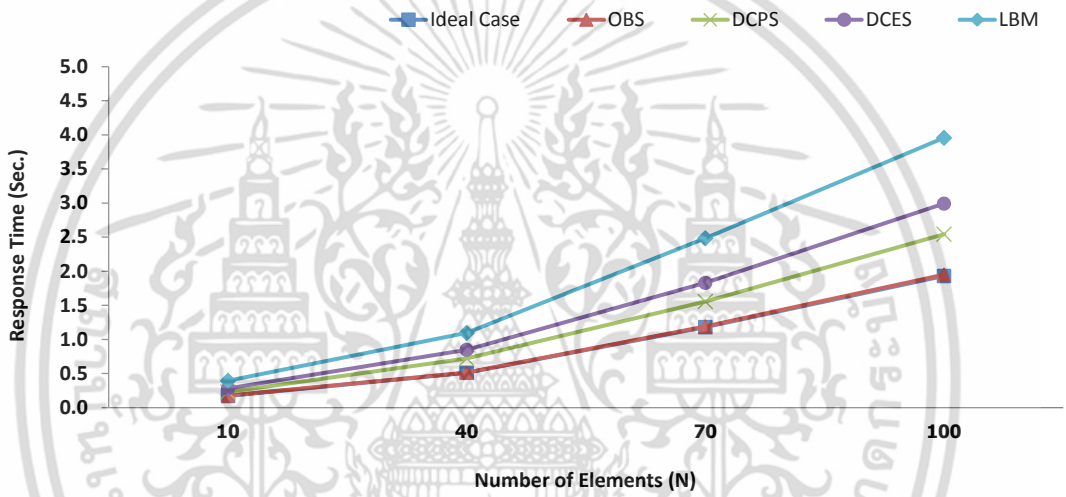


รูปที่ 6.1 เปรียบเทียบเวลาที่ใช้ในการเรียงลำดับข้อมูลแบบสุ่มทั้ง 4 แบบกับเวลาในอุดมคติ

จากรูปที่ 6.1 (กำหนดให้ค่าของข้อมูลเป็นแบบสุ่ม) สามารถอธิบายได้ว่าเมื่อจำนวนข้อมูลเพิ่มขึ้นจะทำให้เวลาที่ใช้ในการเรียงลำดับข้อมูลเพิ่มขึ้น โดยเวลาที่ใช้ในการเรียงลำดับข้อมูลแบบโอบีเอส (OBS) เมื่อจำนวนหน่วยประมวลผลเท่ากับ 8 หน่วยประมวลผล และเพิ่มจำนวนข้อมูลเป็น 10, 40, 70 และ 100 ล้านข้อมูล จะให้ผลดังนี้ 0.18, 0.54, 1.28 และ 2.10 วินาที ใกล้เคียงกับเวลาในอุดมคติ กล่าวคือ 0.17, 0.51, 1.18 และ 1.93 วินาที ซึ่งคิดเป็น 94%, 94%, 92% และ 92% ตามลำดับ ซึ่งเป็นวิธีที่ให้ผลดีที่สุด เพราะมีรอบการทำงานหลักเป็นแบบไดนามิคระหว่าง 1 ถึง $\log_2 P = 3$ รอบ ส่วนการเรียงลำดับแบบอื่นๆ เช่นแบบดีซีพีเอส (DCPS) และแบบดีซีอีเอส (DCES) ใช้เวลาในการเรียงลำดับข้อมูลใกล้เคียงกันระหว่าง 1 ถึง $(\log_2 P(\log_2 P+1))/2 = 6$ รอบ และแบบสุดท้ายคือแบบแอลบีเอ็ม (LBM) เป็นวิธีที่ใช้เวลาในการเรียงลำดับข้อมูลมากที่สุด เพราะมีรอบการทำงานหลักเป็นแบบสแตทิคเท่ากับ $(\log_2 P(\log_2 P+1))/2 = 6$ รอบ

ตารางที่ 6.2 เวลาที่ใช้ในการเรียงลำดับข้อมูล (Response Time) ของ 4 แบบโดยเปรียบเทียบกับเวลาที่ใช้ในการเรียงลำดับข้อมูลในอุดมคติ (Response Time of Ideal Case) เมื่อข้อมูลเป็นแบบเอนเอียงด้านซ้ายมีขนาดตั้งแต่ 10, 40, 70, 100 ล้านข้อมูล 8 หน่วยประมวลผล

วิธีการ/จำนวนข้อมูล	10	40	70	100
เวลาในอุดมคติ	0.17	0.51	1.18	1.93
เวลาในการเรียงลำดับแบบ “LBM”	0.39	1.09	2.48	3.96
เวลาในการเรียงลำดับแบบ “DCES”	0.28	0.85	1.83	2.99
เวลาในการเรียงลำดับแบบ “DCPS”	0.23	0.72	1.56	2.54
เวลาในการเรียงลำดับแบบ “OBS”	0.18	0.51	1.19	1.95

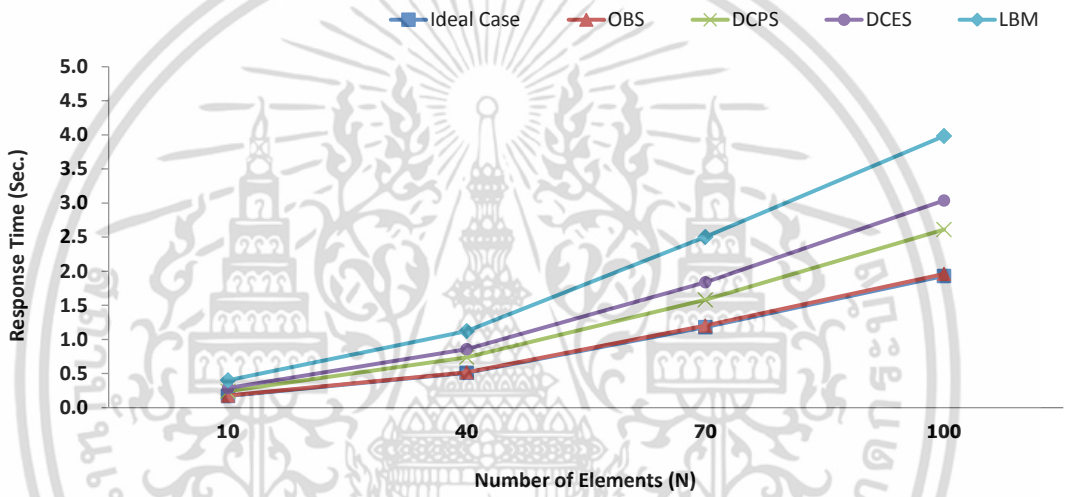


รูปที่ 6.2 เปรียบเทียบเวลาในการเรียงข้อมูลแบบเอนเอียงด้านซ้าย ทั้ง 4 แบบกับเวลาในอุดมคติ

จากรูปที่ 6.2 (กำหนดให้ข้อมูลเป็นแบบเอนเอียงด้านซ้าย กล่าวคือ ค่าส่วนใหญ่มีค่าน้อยๆ) เมื่อจำนวนหน่วยประมวลผลเท่ากับ 8 หน่วยประมวลผล และเพิ่มจำนวนข้อมูลเป็น 10, 40, 70 และ 100 ล้านข้อมูล โดยเวลาที่ใช้ในการเรียงลำดับข้อมูลแบบ โอบีเอส (OBS) ให้ผลดังนี้ 0.18, 0.51, 1.19 และ 1.95 วินาที ซึ่งใกล้เคียงกับเวลาในอุดมคติ กล่าวคือ 0.17, 0.51, 1.18 และ 1.93 วินาที ซึ่งคิดเป็น 98%, 99%, 99% และ 99% ตามลำดับ ซึ่งเป็นวิธีที่ให้ผลดีที่สุดเนื่องจากรอบในการทำงานอยู่ระหว่าง 1 ถึง $\log_2 P$ (3 รอบ) เท่านั้น ส่วนแบบอื่นๆ เช่น แบบดีซีพีเอส และแบบดีซีอีเอส ใช้เวลาในการเรียงลำดับข้อมูลใกล้เคียงกันคืออยู่ระหว่าง 1 ถึง $\log_2 P(\log_2 P+1)/2$ (6 รอบ) และแบบสุดท้ายคือแบบแอลบีเอ็ม เป็นวิธีที่ใช้เวลาในการเรียงลำดับข้อมูลมากที่สุดเนื่องจากมีรอบการทำงานแบบสแตทิคกล่าวคือ $\log_2 P(\log_2 P+1)/2$ (6 รอบ) เสมอ

ตารางที่ 6.3 เวลาที่ใช้ในการเรียงลำดับข้อมูล (Response Time) ของ 4 แบบโดยเปรียบเทียบกับเวลาที่ใช้ในการเรียงลำดับข้อมูลในอุดมคติ (Response Time of Ideal Case) เมื่อข้อมูลเป็นแบบเอนเอียงด้านขวามีขนาดตั้งแต่ 10, 40, 70, 100 ล้านข้อมูล 8 หน่วยประมวลผล

วิธีการ/จำนวนข้อมูล	10	40	70	100
เวลาในอุดมคติ	0.17	0.51	1.18	1.93
เวลาในการเรียงลำดับแบบ “LBM”	0.40	1.12	2.50	3.98
เวลาในการเรียงลำดับแบบ “DCES”	0.29	0.86	1.84	3.04
เวลาในการเรียงลำดับแบบ “DCPS”	0.24	0.74	1.58	2.61
เวลาในการเรียงลำดับแบบ “OBS”	0.18	0.52	1.20	1.96



รูปที่ 6.3 เปรียบเทียบเวลาในการเรียงข้อมูลแบบเอนเอียงด้านขวา ทั้ง 4 แบบกับเวลาในอุดมคติ

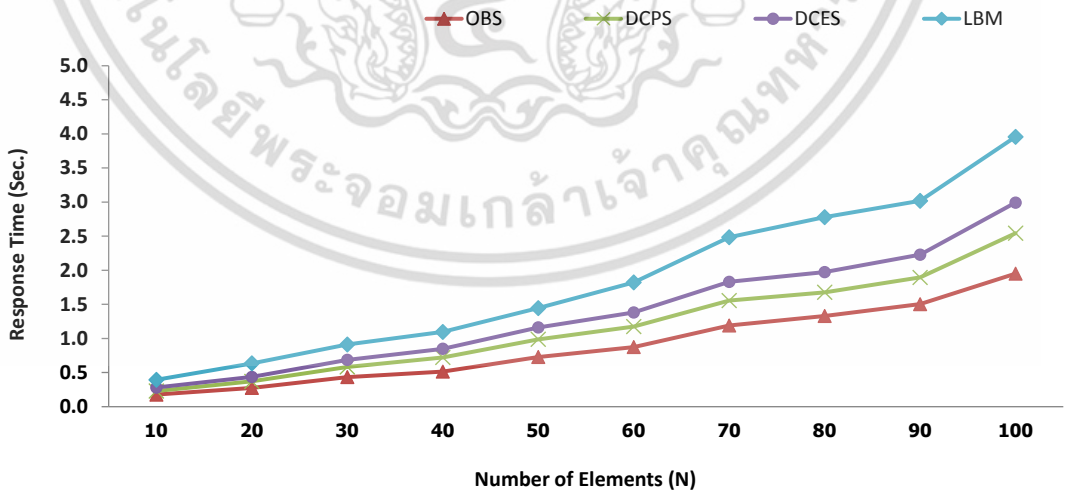
จากรูปที่ 6.3 (กำหนดให้ข้อมูลเป็นแบบเอนเอียงด้านขวา กล่าวคือ ค่าส่วนใหญ่มีค่ามากๆ) เมื่อจำนวนหน่วยประมวลผลเท่ากับ 8 หน่วยประมวลผล และเพิ่มจำนวนข้อมูลเป็น 10, 40, 70 และ 100 ล้านข้อมูล โดยเวลาที่ใช้ในการเรียงลำดับข้อมูลแบบ โอบีเอส (OBS) ให้ผลดังนี้ 0.18, 0.52, 1.20 และ 1.96 วินาที ใกล้เคียงกับเวลาในอุดมคติ กล่าวคือ 0.17, 0.51, 1.18 และ 1.93 วินาที ซึ่งคิดเป็น 97%, 99%, 98% และ 98% ตามลำดับ ซึ่งเป็นวิธีที่ให้ผลดีที่สุดเนื่องจากรอบในการทำงานอยู่ระหว่าง 1 ถึง $\log_2 P$ (3 รอบ) เท่านั้น ส่วนแบบอื่นๆ เช่น แบบดีซีพีเอส และแบบดีซีอีเอส ใช้เวลาในการเรียงลำดับข้อมูลใกล้เคียงกันคืออยู่ระหว่าง 1 ถึง $\log_2 P(\log_2 P+1)/2$ (6 รอบ) และแบบสุดท้ายคือแบบแอลบีเอ็ม เป็นวิธีที่ใช้เวลาในการเรียงลำดับข้อมูลมากที่สุดเนื่องจากมีรอบการทำงานแบบสแตทิคกล่าวคือ $\log_2 P(\log_2 P+1)/2$ (6 รอบ) เสมอ

6.3.2 ผลการทดลองเปรียบเทียบเมื่อขนาดข้อมูลที่ใช้ทดลองต่างกัน

การทดลองเปรียบเทียบการเรียงลำดับข้อมูลทั้ง 4 แบบที่ได้กล่าวมาในข้างต้น สำหรับข้อมูลที่มีขนาดต่างกัน โดยทดลองที่ขนาดของชุดข้อมูลตั้งแต่ 10-100 ล้านข้อมูล และกำหนดให้จำนวนหน่วยประมวลผลที่ใช้ในการคำนวณเท่ากับ 8 หน่วยประมวลผล ซึ่งแสดงผลการทดลองดังนี้ ตารางที่ 6.4 และรูปที่ 6.4 แสดงเวลาที่ใช้ในการเรียงลำดับข้อมูล (Response Time) ที่เป็นผลรวมของเวลาที่ใช้ในการติดต่อสื่อสารและเวลาที่ใช้ในการประมวลผล

ตารางที่ 6.4 เวลาที่ใช้ในการเรียงลำดับข้อมูลแบบเอนเอียงด้านซ้าย (Left-Skewed Data) ทั้ง 4 แบบ เมื่อใช้ 8 หน่วยประมวลผล และขนาดข้อมูลตั้งแต่ 10-100 ล้านชุดข้อมูล

จำนวนชุดข้อมูล/วิธีการ	LBM	DCES	DCPS	OBS
10×10^6	0.39	0.28	0.23	0.18
20×10^6	0.63	0.43	0.37	0.27
30×10^6	0.91	0.69	0.58	0.43
40×10^6	1.09	0.85	0.72	0.51
50×10^6	1.44	1.16	0.99	0.73
60×10^6	1.82	1.38	1.17	1.87
70×10^6	2.48	1.83	1.56	1.19
80×10^6	2.78	1.97	1.68	1.33
90×10^6	3.02	2.23	1.89	1.50
100×10^6	3.96	2.99	2.54	1.95



รูปที่ 6.4 เปรียบเทียบเวลาที่ใช้ในการเรียงลำดับแบบเอนเอียงด้านซ้าย ทั้ง 4 แบบ เมื่อ P = 8

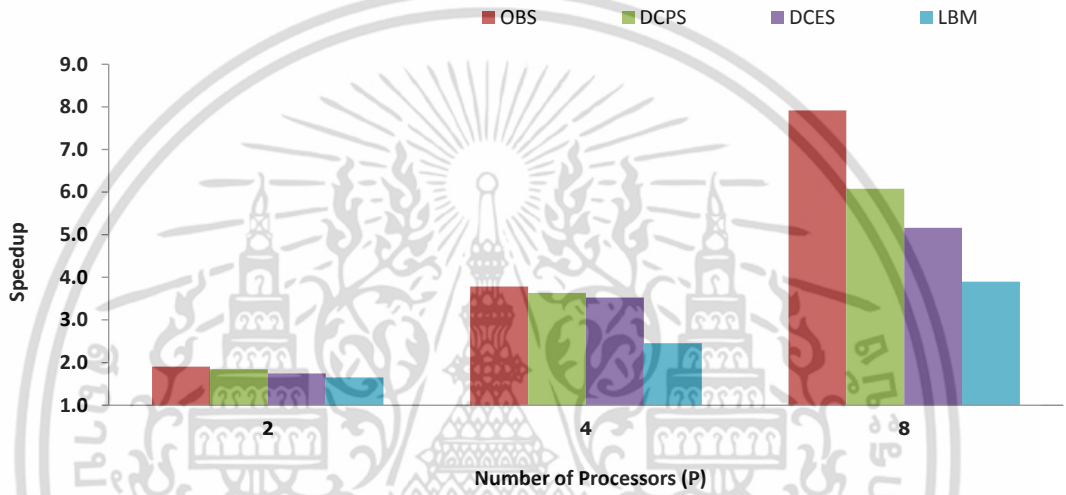
จากรูปที่ 6.4 แสดงเวลาที่ใช้ในการเรียงลำดับข้อมูลแบบเอนเอียงด้านซ้าย โดยเปรียบเทียบเวลาทั้ง 4 แบบ โดยการเรียงลำดับข้อมูลแบบโอบีเอสดีกว่าแบบแอลบีเอ็มประมาณ 50-53% ดีกว่าแบบดีซีอีเอสประมาณ 35-40% เนื่องจากจำนวนรอบในการทำงานของแบบแอลบีเอ็มเป็นแบบสแตทิคและขนาดของชุดข้อมูลที่ส่ง/รับในแต่ละรอบนั้น มีขนาดเท่ากับ w=N/P ในขณะที่ทั้งสองแบบรองลงมาคือแบบดีซีอีเอสและดีซีพีเอส มีจำนวนรอบในการทำงานระหว่าง 1 ถึง $\log_2 P(\log_2 P+1)/2$ (6 รอบ) และรับ/ส่งข้อมูลมีขนาดไม่เกิน w/2 ซึ่งในบางเวลาไม่จำเป็นต้องแลกเปลี่ยนข้อมูลทำให้เวลาในการติดต่อสื่อสารระหว่างหน่วยประมวลผลลดลง ส่งผลให้เวลาที่ใช้ในการประมวลผลทั้งหมดลดลงด้วย

6.3.3 ผลการเปรียบเทียบเมื่อเพิ่มจำนวนหน่วยประมวลผล

การเรียงลำดับข้อมูลของทั้งสี่แบบกำหนดให้ขนาดข้อมูล (N) เท่ากับ 100 ล้านข้อมูล และทำการเพิ่มจำนวนหน่วยประมวลผลตั้งแต่ 2, 4 และ 8 ตามลำดับ ในงานวิจัยนี้เน้นการเพิ่มประสิทธิภาพในการเรียงลำดับ ทั้งด้านการติดต่อสื่อสารแบบไดนามิค ด้านการแลกเปลี่ยนข้อมูล รวมไปถึงการรวมชุดข้อมูล โดยทุกๆ ด้านที่กล่าวมาส่งผลต่อเวลาในการทำงานลดลงอย่างมากเมื่อเพิ่มหน่วยประมวลผลเปรียบเทียบผลการทดลองทั้ง 4 แบบนี้ด้วยอัตราการเพิ่มขึ้นของความเร็วตามสมการที่ 2.7 ($S_P = T_S/T_P$) แสดงในตารางที่ 6.5 และรูปที่ 6.5 ส่วนตารางที่ 6.6 และรูปที่ 6.6 นำค่าของอัตราการเพิ่มขึ้นของความเร็วมาคำนวณหาประสิทธิภาพ ตามสมการที่ 2.8 ($E_P = S_P/P$)

ตารางที่ 6.5 อัตราการเพิ่มขึ้นของความเร็ว (Speedup) ทั้ง 4 แบบ เมื่อข้อมูลมีขนาด 100 ล้านข้อมูล และหน่วยประมวลผลตั้งแต่ 2, 4 และ 8 ตามลำดับ

วิธีการ/จำนวนหน่วยประมวลผล	2	4	8
อัตราการเพิ่มขึ้นของความเร็วแบบ “LBM”	1.65	2.45	3.90
อัตราการเพิ่มขึ้นของความเร็วแบบ “DCES”	1.74	3.52	5.16
อัตราการเพิ่มขึ้นของความเร็วแบบ “DCPS”	1.84	3.63	6.08
อัตราการเพิ่มขึ้นของความเร็วแบบ “OBS”	1.90	3.78	7.92

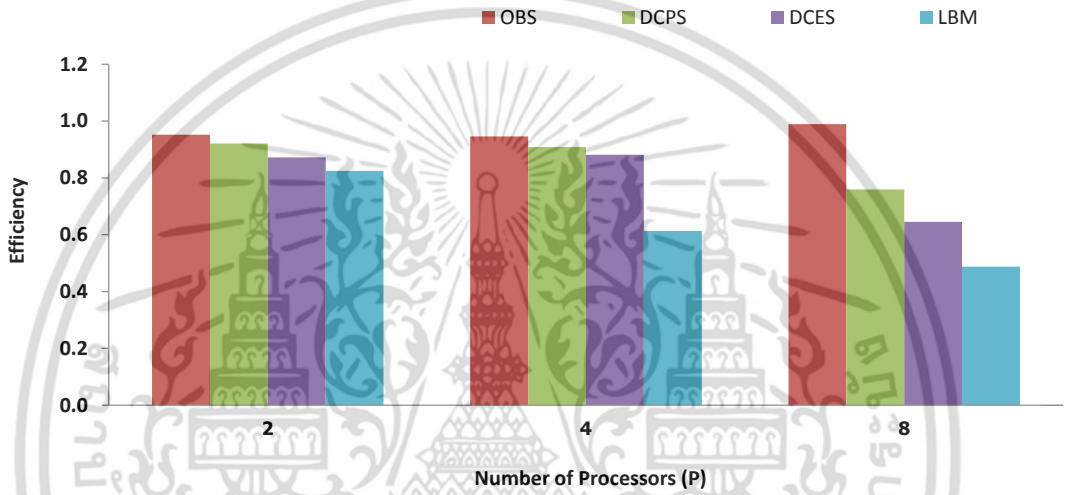


รูปที่ 6.5 เปรียบเทียบอัตราการเพิ่มขึ้นของความเร็วทั้ง 4 แบบ เมื่อ N = 100

จากรูปที่ 6.5 แสดงอัตราการเพิ่มขึ้นของความเร็ว ซึ่งการเรียงลำดับแบบดีซีพีเอส และแบบดีซีอีเอส จะมีอัตราการเพิ่มขึ้นของความเร็วให้ผลใกล้เคียงกันมาก ส่วนการเรียงลำดับแบบแอลบีเอ็มให้ผลของอัตราการเพิ่มขึ้นของความเร็ว 1.65, 2.45 และ 3.90 เมื่อใช้ 2, 4 และ 8 หน่วยประมวลผล ตามลำดับ เมื่อเปรียบเทียบกับแบบโอบีเอส ซึ่งให้ผลของอัตราการเพิ่มขึ้นของความเร็ว 1.90 เมื่อใช้ 2 หน่วยประมวลผล และอัตราการเพิ่มขึ้นของความเร็ว 3.78 เมื่อใช้ 4 หน่วยประมวลผล สุดท้ายคือเมื่อใช้ 8 หน่วยประมวลผล อัตราการเพิ่มขึ้นของความเร็ว 7.92 พบว่าการเรียงลำดับข้อมูลแบบโอบีเอส เป็นแบบที่ให้ผลดีที่สุด ในทุกๆ ผลการทดลอง

ตารางที่ 6.6 ประสิทธิภาพ (Efficiency) ทั้ง 4 แบบ เมื่อข้อมูลมีขนาด 100 ล้านข้อมูล และหน่วยประมวลผลตั้งแต่ 2, 4 และ 8 ตามลำดับ

วิธีการ/จำนวนหน่วยประมวลผล	2	4	8
ประสิทธิภาพแบบ “LBM”	0.82	0.61	0.49
ประสิทธิภาพแบบ “DCES”	0.87	0.88	0.65
ประสิทธิภาพแบบ “DCPS”	0.92	0.91	0.76
ประสิทธิภาพแบบ “OBS”	**0.95**	**0.95**	**0.99**



รูปที่ 6.6 เปรียบเทียบประสิทธิภาพทั้ง 4 แบบ เมื่อ N = 100

จากรูปที่ 6.6 แสดงผลการเปรียบเทียบประสิทธิภาพของการเรียงลำดับทั้ง 4 แบบ ซึ่งการเรียงลำดับแบบโอบีเอส ให้ประสิทธิภาพเข้าใกล้ 1 มากที่สุดคิดเป็น 0.95, 0.95 และ 0.99 เมื่อใช้หน่วยประมวลผลเท่ากับ 2, 4 และ 8 ตามลำดับ สำหรับข้อมูลขนาด 100 ล้านข้อมูล ส่วนแบบดีซีพีเอส และแบบดีซีอีเอส จะให้ผลใกล้เคียงกัน และแบบสุดท้ายคือแบบแอลบีเอ็ม ให้ผลน้อยที่สุดเมื่อเปรียบกับสามแบบแรก

6.3.4 ผลการทดลองเปรียบเทียบจำนวนรอบในการเรียงลำดับข้อมูล

ในส่วนของการทดลองนี้เป็นการเทียบเปรียบจำนวนรอบหลักในการเรียงลำดับข้อมูลแบบไดนามิค โดยทำการเปรียบเทียบการเรียงลำดับแบบโอบีเอส (1 ถึง $\log_2 P$) รอบ และแบบดีซีอีเอส (1 ถึง $(\log_2 P(\log_2 P+1))/2$ รอบ เมื่อ P=8 โดยแสดงข้อมูลดังตารางที่ 6.7

ตารางที่ 6.7 แสดงการเปรียบเทียบจำนวนรอบในการเรียงลำดับข้อมูลระหว่างการเรียงลำดับแบบดีซีอีเอส และแบบโอบีเอส โดยใช้ชุดข้อมูลในการทดลอง 10,000 ชุดข้อมูล

Problem sizes (N)	Input data (iteration)	BEST		Others		WORST	
		DCES (1)	**OBS** (1)	DCES (2-3)	**OBS** (2)	DCES (4-6)	**OBS** (3)
	Random	453	**847**	2196	**3604**	6851	**6049**
$10x10^6$	L-Skew	363	**1037**	1391	**2709**	7846	**6654**
	R-Skew	273	**1227**	953	**2347**	8174	**7026**
	Random	587	**913**	2168	**3632**	6545	**6155**
$20x10^6$	L-Skew	440	**1160**	1375	**3125**	7335	**6565**
	R-Skew	293	**1107**	1068	**2532**	7939	**7061**
	Random	501	**1099**	1913	**3387**	6586	**6514**
$30x10^6$	L-Skew	310	**1190**	1450	**3150**	7390	**6510**
	R-Skew	593	**1107**	1519	**2681**	7788	**6312**
	Random	468	**832**	1733	**3467**	6899	**6601**
$40\ x10^6$	L-Skew	262	**838**	1527	**3473**	7211	**6689**
	R-Skew	140	**560**	1362	**3538**	7498	**6902**
	Random	685	**1015**	1938	**3762**	6427	**6173**
$50\ x10^6$	L-Skew	192	**908**	1774	**3326**	7234	**6566**
	R-Skew	295	**805**	1820	**2780**	6985	**6315**
	Random	313	**987**	2545	**3255**	6692	**6208**
$60\ x10^6$	L-Skew	454	**946**	1606	**3194**	7240	**6560**
	R-Skew	370	**730**	1905	**3195**	7525	**6275**
	Random	259	**641**	2319	**3481**	7422	**5878**
$70\ x10^6$	L-Skew	315	**985**	1375	**2925**	7110	**7290**
	R-Skew	302	**898**	1629	**3171**	7369	**6631**
	Random	284	**516**	1217	**2683**	8299	**7001**
$80\ x10^6$	L-Skew	153	**547**	1492	**3308**	7255	**7245**
	R-Skew	148	**452**	1553	**3647**	7399	**6801**
	Random	237	**463**	1474	**2726**	7959	**7141**
$90\ x10^6$	L-Skew	178	**522**	1800	**3000**	7822	**6678**
	R-Skew	110	**490**	1605	**3495**	7385	**6915**
	Random	174	**326**	1962	**3238**	7314	**6986**
$100\ x10^6$	L-Skew	121	**479**	1755	**3445**	7474	**6726**
	R-Skew	267	**533**	1674	**3726**	7159	**6641**

สำหรับข้อมูลในตารางที่ 6.9 แสดงให้เห็นว่าการเรียงลำดับข้อมูลแบบโอบีเอสให้ผลดีกว่าแบบดีซีอีเอสในทุกๆ กรณี กล่าวคือ ใน 10,000 ชุดข้อมูล ปรากฏกรณีดีที่สุด (1 รอบ) มากกว่า และปรากฏกรณีเฉลี่ยมีจำนวนรอบน้อยกว่า

บทที่ 7

สรุปผลการทดลองและแนวทางการพัฒนางานวิจัย

7.1 สรุปผลและวิเคราะห์ผลการทดลอง

การเรียงลำดับแบบขนานที่ผู้วิจัยได้นำเสนอไว้ในบทที่ 3 และ 4 กรณีที่ P<N มีทั้งหมด 3 แบบคือ แบบที่ 1) ดีซีอีเอส (DCES) แบบที่ 2) ดีซีพีเอส (DCPS) และแบบที่ 3) โอบีเอส (OBS) ที่นำเสนอต่อยอดในแต่ละประเด็นสำคัญๆ ส่วนแบบแอลบีเอ็ม (LBM) เป็นงานวิจัยในอดีตที่มีผู้นำเสนอไว้ซึ่งกล่าวในบทที่ 2 โดยแต่ละแบบดังกล่าวมีวัตถุประสงค์และความยากง่ายในขั้นตอนวิธีในประเด็นที่แตกต่างกันดังนี้คือ การเรียงลำดับแบบที่ 1 ดีซีอีเอส เป็นแบบที่เน้นการเพิ่มประสิทธิภาพในการเรียงลำดับข้อมูลครบทุกด้าน แต่มีข้อจำกัดสำหรับข้อมูลบางลักษณะเช่นข้อมูลแบบเอนเอียง จึงได้นำเสนอแบบที่ 2 ดีซีพีเอส เพื่อจัดการกับข้อจำกัดดังกล่าวด้วยค่ามิดพอยท์ ส่วนแบบที่ 3 โอบีเอส เป็นแบบที่ดีที่สุดเพราะทุกๆ ขั้นตอนของการเรียงลำดับข้อมูลได้ถูกเพิ่มประสิทธิภาพอย่างสูงสุดแล้ว ทั้งขั้นตอนการติดต่อสื่อสารระหว่างหน่วยประมวลผล ขั้นตอนการแลกเปลี่ยนข้อมูล ขั้นตอนการรวมข้อมูล ส่งผลให้แบบโอบีเอส เป็นการเรียงลำดับแบบขนานที่ดีที่สุดในปัจจุบันนี้สำหรับกรณีที่ P<N

จากผลการทดลองที่ผ่านมาในบทที่ 6 ซึ่งเป็นการทดลองบนระบบคอมพิวเตอร์แบบมัลติคอร์ (Multi-Cores Computer) พบว่าการเรียงลำดับแบบขนานที่มีประสิทธิภาพสูงด้วยวิธีไดนามิคในแต่ละแบบมีการเพิ่มประสิทธิภาพในด้านต่างๆ ที่แตกต่างกันสอดคล้องกับทฤษฎีที่กล่าวมาแล้วในบทที่ 3 และ 4 ซึ่งสามารถแบ่งได้เป็น 2 ด้าน ดังนี้

1) การเพิ่มประสิทธิภาพโดยการลดเวลาที่ใช้ในการติดต่อสื่อสาร (Communication Time) ด้วยการติดต่อสื่อสารแบบไดนามิค และการแลกเปลี่ยนข้อมูลที่มีประสิทธิภาพสูง
2) การเพิ่มประสิทธิภาพโดยการลดเวลาที่ใช้ในการประมวลผลทั้งหมด (Computation Time) ด้วยการรวมข้อมูลที่มีประสิทธิภาพสูง

ตารางที่ 7.1 แสดงการเปรียบเทียบจำนวนรอบการทำงานของการติดต่อสื่อสารแบบไดนามิค (Dynamic) และแบบสแตทิค (Static) ทั้ง 4 แบบ

ชนิดของ การเรียงลำดับ	การติดต่อ สื่อสาร	ดีที่สุด Best	แย่ที่สุด Worst	Update List
แบบแอลบีเอ็ม	สแตทิค	$\log_2 P(\log_2 P+1)/2$	$\log_2 P(\log_2 P+1)/2xN/P$	-
แบบดีซีอีเอส	ไดนามิค	1	$\log_2 P(\log_2 P+1)/2xN/P$	2(P-1)
แบบดีซีพีเอส	ไดนามิค	1	$\log_2 PxN/P$	2(P-1)
แบบโอบีเอส	**ไดนามิค**	1	$\log_2 PxN/P$	$\log_2 P(\log_2 P+1)/2$

ในตารางที่ 7.1 อธิบายถึงจำนวนรอบในการเรียงลำดับข้อมูลด้วยวิธีการติดต่อสื่อสารแบบไดนามิค และการติดต่อสื่อสารแบบสแตทิค โดยแบบแรกแอลบีเอ็ม เป็นการติดต่อสื่อสารแบบสแตทิค กล่าวคือหากเริ่มกระบวนการเรียงลำดับแล้วต้องทำครบทุกรอบ ดังนั้นจำนวนรอบในการทำงานเท่ากับ $\log_2 P(\log_2 P+1)/2$ ทั้งกรณีที่ดีที่สุดและแย่ที่สุด ส่วนสามแบบหลังเป็นการติดต่อสื่อสารแบบไดนามิค คือ แบบดีซีอีเอส จากกระบวนการทำงานแบบไดนามิคสามารถจบกระบวนการทำงานได้ตลอดเวลาหากพบว่าข้อมูลได้ถูกจัดเรียงอย่างเหมาะสมแล้ว ดังนั้นกรณีที่ดีที่สุดคือ 1 รอบ ส่วนกรณีที่แย่ที่สุดคือ $\log_2 P(\log_2 P +1)/2$ และเวลาที่ใช้อัพเดทข้อมูลภายในตารางคือ 2(P-1) ส่วนแบบดีซีพีเอส และแบบโอบีเอส ใช้จำนวนรอบการทำงานเท่ากันคือ กรณีที่ดีที่สุด 1 รอบ ส่วนกรณีที่แย่ที่สุดคือ S รอบ แตกต่างกันตรงเวลาที่ใช้อัพเดทข้อมูลภายในตารางสำหรับแบบดีซีพีเอส คือ 2(P-1) ส่วนแบบโอบีเอส คือ $\log_2 P(\log_2 P+1)/2$ ซึ่งเป็นแบบที่ดีที่สุดที่ผู้วิจัยได้นำเสนอ

ตารางที่ 7.2 แสดงการเปรียบเทียบความซับซ้อนด้านเวลาและพื้นที่ (Time & Space Complexity) ของการติดต่อสื่อสารที่มีประสิทธิภาพสูงด้วยวิธีไดนามิค (Dynamic Communication)

ชนิดของการเรียงลำดับ	การกระจายและเรียงลำดับ	การค้นหา	พื้นที่ใช้งาน
การกระจายข้อมูลแบบดั้งเดิม All-to-All Broadcast	$O(P \log_2 P)$	$O(\log_2 P)$	P x P
การกระจายข้อมูลแบบมีประสิทธิภาพ Efficient Broadcast [DCES, DCPS]	O(P)	$O(\log_2 P)$	P x P
การรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูง Efficient Point-to-Point [OBS]	$O((\log_2 P)^2)$	O(1)	P

ตารางที่ 7.2 แสดงการเปรียบเทียบสำหรับการติดต่อสื่อสารระหว่างหน่วยประมวลผลด้วยวิธีไดนามิค โดยแบ่งเป็น 3 ประเภทคือ 1) การกระจายข้อมูลแบบดั้งเดิม 2) การกระจายข้อมูลแบบมีประสิทธิภาพ และ 3) การรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูง ซึ่งการเรียงลำดับแบบขนานที่ผู้วิจัยได้นำเสนอใช้ 2 แบบหลัง โดยการกระจายข้อมูลแบบมีประสิทธิภาพใช้กับการเรียงลำดับแบบดีซีอีเอส และการเรียงลำดับแบบดีซีพีเอส ส่วนการรับ/ส่งข้อมูลโดยตรงที่มีประสิทธิภาพสูงใช้กับการเรียงลำดับแบบโอบีเอส ซึ่งให้ค่าความซับซ้อนด้านเวลาดีที่สุดในทุกๆ ด้าน และใช้พื้นที่ในการทำงานน้อยที่สุดด้วย

ตารางที่ 7.3 แสดงการเปรียบเทียบความซับซ้อนด้านเวลาของการค้นหาสำหรับการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลในแต่ละรูปแบบ (Searching Pattern)

ชนิดของการเรียงลำดับ	การค้นหาแบบคงที่	การค้นหาสลับที่	การค้นหาบางส่วน-1	การค้นหาบางส่วน-2
การเรียงแบบแอลบีเอ็ม (LBM)	O(1)	O(1)	$O(\log_2 N/P)$	$O(\log_2 N/P)$
การเรียงแบบดีซีอีเอส (DCES)	O(1)	O(1)	$O(\log_2 N/P)$	$O(\log_2 N/P)$
การเรียงแบบดีซีพีเอส (DCPS)	O(1)	O(1)	$O(\log_2 N/P)$	$O(\log_2 N/P)$
การเรียงแบบเอชพีเอส (OBS)	O(1)	O(1)	O(1)	$O(\log_2 N/P)$

ตารางที่ 7.3 อธิบายการตรวจสอบรูปแบบการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผล ซึ่งผู้วิจัยได้นำเสนอไว้ 4 รูปแบบ (Pattern) ด้วยกันคือ 1) แบบคงที่ 2) แบบสลับที่ 3) แบบบางส่วน-1 และ 4) แบบบางส่วน-2 ซึ่งในแต่ละรูปแบบมีค่าความซับซ้อนด้านเวลาในการค้นหาดังตาราง ค่าความซับซ้อนด้านเวลาที่ดีที่สุดคือ O(1) ส่วนแบบที่แย่ที่สุดคือ O(N/P) ซึ่งเป็นการเรียงลำดับแบบแอลบีเอ็มซึ่งเป็นการค้นหาแบบบางส่วน

ตารางที่ 7.4 แสดงการเปรียบเทียบขนาดของข้อมูลที่ใช้ในการแลกเปลี่ยนสูงสุดในแต่ละรูปแบบ

ชนิดของการเรียงลำดับ	แบบคงที่	แบบสลับที่	แบบบางส่วน-1	แบบบางส่วน-2
การเรียงแบบแอลบีเอ็ม (LBM)	-	w=N/P	$\leq$ w	$\leq$ w
การเรียงแบบดีซีอีเอส (DCES)	-	-	$\leq$ w	$\leq$ w
การเรียงแบบดีซีพีเอส (DCPS)	-	-	$\leq$ w/2	$\leq$ w/2
การเรียงแบบเอชพีเอส (OBS)	-	-	$\leq$ w/2	$\leq$ w/2

ตารางที่ 7.4 เปรียบเทียบขนาดของข้อมูลที่ลดลงสำหรับรูปแบบในการแลกเปลี่ยนข้อมูลระหว่างหน่วยประมวลผลที่ผู้วิจัยได้นำเสนอ กล่าวคือ การเรียงลำดับแบบแอลบีเอ็ม หากข้อมูลอยู่ในรูปแบบสลับที่ ต้องมีการแลกเปลี่ยนข้อมูลขนาด w=N/P เมื่อ N คือ จำนวนข้อมูล และ P คือ จำนวนหน่วยประมวลผล ในขณะที่การเรียงลำดับสามแบบหลัง (DCES, DCPS, OBS) ไม่มีการแลกเปลี่ยนข้อมูลสำหรับรูปแบบสลับที่ ส่วนรูปแบบการแลกเปลี่ยนข้อมูลแบบบางส่วน-1 และบางส่วน-2 มีขนาดข้อมูลลดลง 50% เนื่องจากการเรียงลำดับข้อมูลเหล่านี้สามารถกลับข้อมูลที่ต้องการส่งได้หากตรวจสอบแล้วพบว่าข้อมูลมีขนาดมากกว่า w/2 เรียกว่าการรับ/ส่งข้อมูลแบบกลับด้าน ซึ่งแสดงรายละเอียดในบทที่ 4

ตารางที่ 7.5 แสดงการเปรียบเทียบความซับซ้อนด้านเวลาที่เน้นเฉพาะส่วนของการเรียงลำดับแบบขนาน

ชนิดของการเรียงลำดับ	ดีที่สุด	เฉลี่ย	แย่ที่สุด
การเรียงแบบแอลบีเอ็ม (LBM)	$O(wS^2)$	$O(wS^2)$	$O(wS^2)$
การเรียงแบบดีซีอีเอส (DCES)	O(P)	$O(wS^2)$	$O(wS^2)$
การเรียงแบบดีซีพีเอส (DCPS)	O(P)	O(wS)	O(wS)
การเรียงแบบโอพีเอส (OBS)	$O(S^2)$	O(wS)	O(wS)

ตารางที่ 7.5 สำหรับความซับซ้อนด้านเวลาซึ่งเน้นเฉพาะส่วนของการเรียงลำดับแบบขนาน (ไม่รวม Local Sort ที่อาจจะประยุกต์ Quick Sort หรือ Radix Sort) สามารถแบ่งได้เป็น 3 กรณี คือ กรณีที่ดีที่สุด (Best Case), กรณีเฉลี่ย (Average Case) และกรณีที่แย่ที่สุด (Worst Case) โดยการเรียงลำดับแบบโอบีเอส ให้ค่าความซับซ้อนด้านเวลาดังนี้ $O(S^2)$ สำหรับกรณีที่ดีที่สุด และ O(wS) สำหรับกรณีเฉลี่ยและแย่ที่สุด โดยที่ $w=N/P$ และ $S=\log_2 P$ ซึ่งเห็นว่าเป็นการเรียงลำดับข้อมูลแบบขนานที่ดีที่สุด

7.2 แนวทางการพัฒนางานวิจัย

- ด้านทฤษฎีนำเสนอขั้นตอนวิธีการใหม่สำหรับการเรียงลำดับข้อมูล ที่ง่ายและให้ผลดีกว่าแบบอื่นๆ ที่ได้นำเสนอไว้แล้ว เช่น วิธีที่มีความซับซ้อนด้านเวลาน้อยกว่า $O(\log_2 P)^2$
- ด้านประยุกต์นำขั้นตอนวิธีการที่มีประสิทธิภาพนี้ ไปประยุกต์ใช้ในด้านต่างๆ อาทิ การสร้างดัชนีข้อมูลเพื่อช่วยในการสืบค้น เพื่อลดเวลาที่ใช้ในการประมวลผลลง

เอกสารอ้างอิง

[1] Adler M., Byers J.W. and Karp R.M. "Parallel Sorting with Limited Bandwidth." **SIAM Journal Computer .** 2000, pp. 1997-2015.

[2] Batcher K.E. "Sorting networks and their applications." **Proceedings Spring Joint Computing Conference AFIPS.** Washington DC, 1968, pp.307-314.

[3] Bitton D., DeWitt D.J., Hsiao D.K. and Menon J. "A Taxonomy of Parallel Sorting." **ACM Computer Survey.** 1984, pp. 287-318.

[4] Brest J., Vreze A. and Zumer V. "A Sorting Algorithm on a PC Cluster." **Proceedings 2000 ACM Symposium on Applied Computing.** Como, Italy, 2000, pp. 710-715.

[5] Chhugani J. "Efficient implementation of sorting on multi-core SIMD CPU architecture.", **Journal Proceedings of the VLDB Endowment.** Vol.1, No. 2, 2008, pp.1313-1324.

[6] Dimitrov R. and Skjellum A. "Software Architecture and Performance Comparison of MPI/Pro and MPICH." **International Conference on Computational Science.** 2003, pp. 307-315.

[7] El-Nashar A.I. "Parallel Performance of MPI Sorting Algorithms on Dual-Core Processor Windows-Based Systems." **International Journal of Distributed and Parallel Systems (IJDPS)** Vol.2, No.3, 2011, pp. 1-14.

[8] Gropp W., Lusk E. and Skjellum A. **Using MPI: Portable Parallel Programming with the Massage Passing Interface.** Cambridge : MIT Press. 1994.

[9] Helman D.R. and JaJa J. "Sorting on Cluster of SMPs." **12th International Parallel Processing Symposium.** University of Maryland, College Park, MD, USA. 1997.

[10] Hwang K. **Advanced Computer Architecture: Parallelism, Scalability, Programmability.** New York : McGraw-Hill. 1993.

[11] Ionescu M.F. and Schauser K.E. "Optimizing Parallel Bitonic Sort." **Proceedings 11th Int'l Parallel Processing Symposium,** 1997, pp. 303-309.

[12] Jeon, M. Kim, D. "Parallel Merge Sort with Load Balancing", **International Journal of Parallel Programming**, Vol. 31, No. 1, 2003, pp. 21-33.

เอกสารอ้างอิง (ต่อ)

[13] Jim´enez-Gonz´alez, D. Navarro, J. and Larriba-Pey, J.-L. "The Effect of local sort on parallel sorting algorithms," **In proceedings of Euromicro Workshop on Parallel, Distributed and Network-based Processing**, Spain, January 2002, pp. 360-367.

[14] Kim Y.C., Jeon M., Kim D. and Sohn A. "Communication-Efficient Bitonic Sort on a Distributed Memory Parallel Computer." **Int' l Conference Parallel and Distributed Systems.** 2001, pp. 165-170.

[15] Kumar V. **Introduction to parallel computing : design and analysis of algorithms.** Redwood City, CA : Benjamin/Cummings. 1994.

[16] Lee J.D. and Batcher K.E. "Minimizing Communication in the Bitonic Sort." **IEEE Transaction on Parallel and Distributed Systems.** 2000, pp. 459-473.

[17] Leighton T. "Tight Bounds on the Complexity of Parallel Sorting." **IEEE Transaction on Computers.** 1985, pp. 344-354.

[18] Message Passing Interface Forum. "MPI: A message passing interface standard." **Int'l Journal of Supercomputer Applications.** 1994.

[19] Miller R. **Algorithms sequential and parallel : a unified approach.** Upper Saddle River, NJ : Prentice Hall. 2000.

[20] **"Multi-Cores."** [Online]. Availavle: http://www.intel.com/

[21] Nakatani T., Huang S.T., Arden B.W. and Tripathi S.K. "K-Way Bitonic Sort**." IEEE Transaction Computers.** 1989, pp. 283-288.

[22] Quinn J.M. **Parallel Programming in C with MPI and OpenMP,** International Edition 2003. Singapore : McGraw Hill. 2003.

[23] Quinn J.M. **Parallel computing : theory and practice.** 2nd ed. New York : McGraw-Hill. 1994.

[24] Thanakulwarapas, T. and Werapun, J. "Communication -space efficient parallel bitonic sorting on symmetric multiprocessors," **In proceedings of Advances in Computer Science and Technology**, Langkawi, Malaysia, April 2008, , pp. 74-78.

[25] Thanakulwarapas, T. and Werapun, J. "Dynamic Communication-Efficient Parallel Sorting on SMPs." **In proceedings of the 11th IEEE International Conference on High Performance Computing and Communications,** June 2009, pp. 132-138.

เอกสารอ้างอิง (ต่อ)

[26] Thanakulwarapas, T. and Werapun, J. “An Optimized Bitonic Sorting Strategy with Midpoint-based Dynamic Communication” **Journal of Parallel and Distributed Computing,** 2015.

[27] **“TILE64 Processor.”**[Online]. Available: http://www.tilera.com/

[28] ชิดชนก เหลือสินทรัพย์ **Analysis & Design of Algorithms** กรุงเทพมหานคร : ซีเอ็ดยูเคชั่น 2543

[29] จีรพร วีระพันธุ์ **การออกแบบและวิเคราะห์ขั้นตอนวิธีทางคอมพิวเตอร์** กรุงเทพมหานคร: จุฬาลงกรณ์มหาวิทยาลัย 2552

[30] ฉัฐไชย์ ลีนาวงศ์ **ตรรกะ...แห่งการพิสูจน์-How to Read and Do Proofs** กรุงเทพมหานคร: ท้อป 2547

[31] จุรีพร บุญนิยม **การเรียงลำดับแบบขนานด้วยวิธีไบโทนิคบนระบบพีซีคลัสเตอร์** วิทยานิพนธ์วิทยาศาสตรมหาบัณฑิต สาขาวิชาวิทยาการคอมพิวเตอร์ บัณฑิตวิทยลัย, สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง 2548

งานวิจัยที่ได้รับตีพิมพ์

<u>วารสารวิชาการ</u>

[1] Tipraporn Thanakulwarapas and Jeeraporn Werapun, “An Optimized Bitonic Sorting Strategy with Midpoint-based Dynamic Communication,” **Journal of Parallel and Distributed Computing,** Copyright©2015 Elsevier Ltd, Accepted date 24 May 2015, DOI:http://dx.doi.org/10.1016/j.jpdc.2015.05.008.

<u>การประชุมวิชาการ</u>

[2] Tipraporn Thanakulwarapas and Jeeraporn Werapun, “Dynamic Communication-Efficient Parallel Sorting on SMPs,” **In proceedings of the 11th IEEE International Conference on High Performance Computing and Communications**, Korea, June 2009**,** pp. 132-138.

[3] Tipraporn Thanakulwarapas and Jeeraporn Werapun, “Communication-space efficient parallel bitonic sorting on symmetric multiprocessors,” **In proceedings of Advances in Computer Science and Technology**, Malaysia, April 2008, pp. 74-78.

Accepted Manuscript

An optimized bitonic sorting strategy with midpoint-based dynamic communication

Tipraporn Thanakulwarapas, Jeeraporn Werapun

PII: S0743-7315(15)00098-2
DOI: http://dx.doi.org/10.1016/j.jpdc.2015.05.008
Reference: YJPDC 3405

To appear in: *J. Parallel Distrib. Comput.*

Received date: 8 January 2014
Revised date: 19 January 2015
Accepted date: 24 May 2015

Please cite this article as: T. Thanakulwarapas, J. Werapun, An optimized bitonic sorting strategy with midpoint-based dynamic communication, *J. Parallel Distrib. Comput.* (2015), http://dx.doi.org/10.1016/j.jpdc.2015.05.008

Highlights

> Optimized Bitonic Sort (OBS) is proposed to improve complexity and sorting time.
> Using midpoint-weight key in OBS can reduce logP(logP+1)/2 to 1,2,3,..,or logP iterations.
> A good key in PE-ranking can find the right place for (Pi,Pj) communication.
> Correctness and complexity analysis of OBS have been proved.
> Experimental results of OBS outperform those of the best of existing method 35%-54%.

.

An Optimized Bitonic Sorting Strategy with Midpoint-based Dynamic Communication

Tipraporn Thanakulwarapas
Department of Computer Science, Faculty of Science
King Mongkut's Institute of Technology, Ladkrabang
Bangkok 10520, THAILAND
S0067102@kmitl.ac.th

Jeeraporn Werapun
Department of Computer Science, Faculty of Science
King Mongkut's Institute of Technology, Ladkrabang
Bangkok 10520, THAILAND
ksjeerap@kmitl.ac.th

Abstract— **This paper proposes an optimized *Bitonic* sorting (*OBS*) strategy with *midpoint*-based *dynamic* communication. Our *OBS* strategy uses the *midpoint*-weight list ranking to improve complexity and reduce time of sorting on parallel and distributed systems. Applying a *better* key in the *PE*-list ranking can find the right place of (P_i, P_j) and improve communication time significantly (i.e., fewer iterations, better synchronization in each iteration, faster convergence to the result), while most of coarse-grain parallel sorting ($P < N$) approaches improve only a large amount of data exchange (N/P) in each of *static* ($s(s+1)/2$) iterations. Theoretically, the *OBS* method can reduce fixed ($s(s+1)/2$) iterations to *1, 2, 3, … ,* or $s=\log_2 P$ iterations, which are improved over those (≤ $s(s+1)/2$ iterations) of the *dynamic DCES* method. In performance evaluation, sorting was accomplished on *multicore* machines. Experimental results showed that our optimized *OBS* outperforms those of the *dynamic DCES* about 35% - 40% and those of the *static LBM* about 51% - 54% (for N = *10* to *100* million elements on an *8-multicore* computer).**

Keywords- Optimized Bitonic sorting; midpoint-based dynamic communication; midpoint-weight list-ranking.

1 INTRODUCTION

Sorting is a crucial computation in many scientific and engineering applications, across many business enterprises. In online information retrieval (IR) on large databases and data mining, many processes require data ordered before operated efficiently in successive tasks. However, sequential sorting (with large datasets of size N) may not be sorted in a reasonable time. On multicores, sequential sorting (e.g., the *quick* sort, etc.) can only be executed using one core or *PE* (processing element). Thus, in order to utilize all *PEs* properly we need a more efficient, parallel sorting method to achieve a faster response time.

For $P = N$ (fine-grain parallelism), there exist several efficient, parallel-sorting approaches with time complexity $O((\log_2 N)^2)$ such as the *Bitonic* sort [1, 2], the parallel merged sort [3, 4], and the network sort [5]. For $P < N$ (coarse-grain parallel sort), existing studies are *Bitonic*-communication sort [6, 8], and parallel load-balance merged sort [7]. These methods are focused on improving communication, especially data exchange between (P_i, P_j) with a large amount of workload $w = N/P$ elements.

In 2001, Kim et al. [6] proposed the Communication Efficient *Bitonic* Sort (*CEBS*) with local *quick* sort in $O(w \log_2 w)_{local\ sort} + O(ws^2)_{parallel\ sort}$, where $w=N/P$ and $s=\log_2 P$. That parallel sorting focused on improving w-data exchange between (P_i, P_j) by introducing three data-exchange patterns: 1. *hold* pattern (without data exchange), 2. *swap* pattern (exchange all w elements without merging), and 3. *partial* pattern (exchange ≤ w elements with result merging). The *CEBS* algorithm showed improved results over those of the original *Bitonic* sort ($P < N$) with full w-data exchange of at least 20%.

In 2003, Jeon and Kim [7] introduced a new parallel merged-sort algorithm with workload balancing, called "Load-Balanced Merge (*LBM*)", by maintaining workload (w) in each *PE* from the initial stage to the final stage. In addition, the *LBM* parallel sort reduced the data exchange between (P_i, P_j) in "*partial* pattern" for merging, according to the median of two data subsets in (P_i, P_j). In experiments, total response time using the *LBM* sort was improved over that of the *CEBS* strategy.

In 2008, the Communication-Space Efficient *Bitonic* Sorting (*CSEBS*) method [8] with local *quick* sort was presented to improve data exchange between (P_i,P_j) in "*partial* pattern". For that pattern, after partial data were identified, the size of partial data in each of (P_i, P_j) were sent before fewer data exchange (≤ w elements). As a result, the *CSEBS* reduced the communication time over that of the *CEBS* strategy.

However, the communication pattern (among *PEs*) associated with parallel sorting approaches [6 - 8] are *static*, and these have to repeat $s(s+1)/2$ iterations to meet all possible pairs of (P_i, P_j). In each iteration, all *PEs* perform data exchange between (P_i, P_j) and merge the results (with w-workload). Thus, time complexity associated with the parallel merge sort, after the initial local *quick* sort, is $O(ws^2)$ or $O(N/P \times (\log_2 P)^2)$, which is not optimal.

In 2009, the *D*ynamic Communication-*E*fficient parallel *S*orting (*DCES*) algorithm [9], our previous study, was proposed to reduce the number of *static* $s(s+1)/2$ iterations, providing for communication among *PEs*, by using the *dynamic PE*-list ranking. In practice, the *DCES* strategy can dynamically reduce $s(s+1)/2$ iterations to *1, 2, 3, … ,* or $s(s+1)/2$ iterations, according to *dynamic* (P_i, P_j) from the ordered *P*-records, updated in each iteration in $O(P)$ by the efficient *all-to-all* broadcast. The time complexity of the parallel merge sorted result of the *DCES* method is $O(P)$ in *best* case and $O(ws^2)$ in *worst* case. In experiments, results of the *DCES* strategy were found to be improved over those of the *LBM* sort. Practically, the *DCES* method is faster in sorting random data. However, for irregular data, the *DCES* strategy may process up to $s(s+1)/2$ iterations.

Recently, for the multi-core machines studied in 2011, two efficient algorithms for sequential sorting (the *quick* sort and the *merge* sort) [12] were developed using $s(s+1)/2$ iterations in communication and full w-data exchange. However, for scalable systems, the communication among *PEs* will decrease the sorting performance, and hence the *dynamic* communication among *PEs* is an ideal solution.

In this paper, we propose an *O*ptimized *Bitonic S*orting (*OBS*) strategy by using a *midpoint*-based *dynamic c*ommunication. Our contribution is a more ideal time complexity of the *OBS* algorithm (in case $P < N$) by sorting the *good* key in $O(s^2)$ in order to find the right place of (P_i, P_j), before data exchange ($\leq w$ elements), in fewer iterations (*1, 2, 3, …,* or *s*), while existing *Bitonic* sort ($P<N$) have to exchange a large amount of data ($\leq w$) between (P_i, P_j) in each of *static* $s(s+1)/2$ iterations. The time complexity of our *OBS* algorithm is $O(s^2)$ in the *best* case scenario and $O(ws)$ in *worst* case, where $w=N/P$ and $s=\log_2 P$. The *worst* complexity ($O(ws)$) is improved over that ($O(ws^2)$) of the *dynamic DCES* algorithm [9] by a factor of *s*. Practically, in order to evaluate the system performance, we implemented the *OBS* algorithm by using C language with MPI interface [10]. In our experiments, the proposed *OBS* method yielded improved results over the *dynamic DCES* 35% - 40% and over the *static LBM* 51% - 54% (for N=10M to 100M on an 8-multicore machine).

The remainder of this paper is organized as follows: Section 2 presents related work, including communication among *PEs (*in case $P < N$*)*, data-exchange patterns (*hold / swap / partial*) between (P_i, P_j) for result merging. Section 3 introduces the optimized *Bitonic* sorting (*OBS*) algorithm to reduce communication steps among *PEs* and better synchronization in data-exchange step. Section 4 proves the validity of the proposed *OBS* algorithm and provides a complexity analysis. Section 5 presents the experimental results of the sorting of a variety of data distributions. Finally, Section 6 discusses the conclusions of our study.

2 Related Work

Sorting is applied in many applications for ordering large quantities of data (N elements). Especially for online processing, data sorting should be returned in a reasonable amount of time. Usually, time complexity of a well-known sorting (i.e., the *quick* sort) is $O(N \log_2 N)$. For parallel sorting (P=N), time complexity of the efficient *Bitonic* sort (Fig.1(a)) is $O((\log_2 N)^2)$. For practical $P < N$, the time complexity of parallel sorting methods consists of local sort and parallel merge sort, which is $O(w \log_2 w)_{local\ sort} + O(ws^2)_{parallel\ sort}$, where w (workload) = N/P and s (steps) = $\log_2 P$. For the sequential local sorting part, a number of efficient techniques (i.e., the *radix* sort) [11] were introduced in 2002. For the parallel sorting part (Fig.2), existing parallel sort methods [6 - 9] have improved the response time of sorting with a variety of data-exchange patterns between (P_i, P_j). However, the time complexity of the parallel merge sort (workload w) is $O(ws^2)$. Recently, the second part of Algorithm 1 was improved with fewer iterations of communication among *PEs* [9] (see detail in Section 2.1), then fewer data exchange for merging [6 - 8] between (P_i, P_j) in Section 2.2.

Algorithm 1. *Static*-communication Parallel Sorting ($P < N$).

	Time Complexity
Parallel-SORT (N, P)	$O(w \log_2 w) + O(ws^2)$
1. **Local sort** (with workload w=N/P)	
Each PE makes Quick sort (N/P) in parallel	
2. **Parallel merge sort** among PEs with *static* communication pattern (s(s+1)/2 iterations)	
for I = 1, 2, … , s (=$\log_2$P)	
for J = 1, 2, … , I	
2.1 **Data Exchange** between (P_I, P_J)	
- Send/Receive (*min,max*)	$O(1)$
- Find data exchange from (*min,max*) (*Hold/Swap/Partial* patterns)	$O(w)$
- Send/Receive data ($\leq$ w elements)	$O(w)$
2.2 **Merging Result** in each PE ($\leq$ w steps)	$O(w)$
end for J	
end for I	
End	

2.1 Communication Patterns among PEs

In parallel communication among *PEs* (in case $P = N$), a popular *static* communication pattern with efficient time complexity $O((\log_2 N)^2)$ or $O(s^2)$ is the *Bitonic* sort [1, 2].

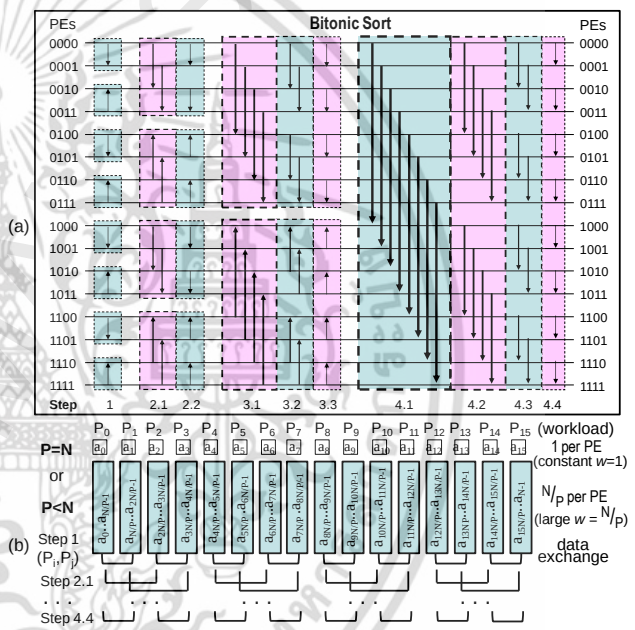


Fig.1. a) Communication pattern in *Bitonic* sort [1] for P=16 and b) a constant workload w=1 (for P=N) and a large amount of workload w=N/P (for $P < N$).

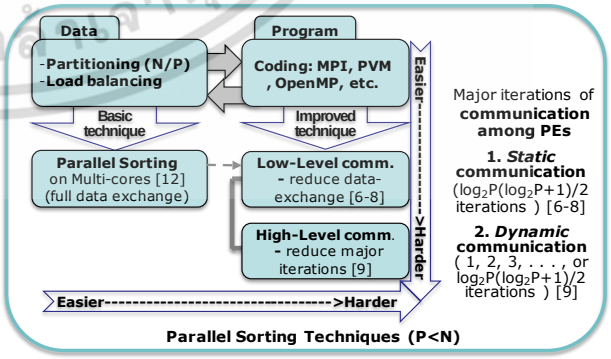


Fig.2. Related work [6 - 9, 12] on parallel sorting ($P < N$).

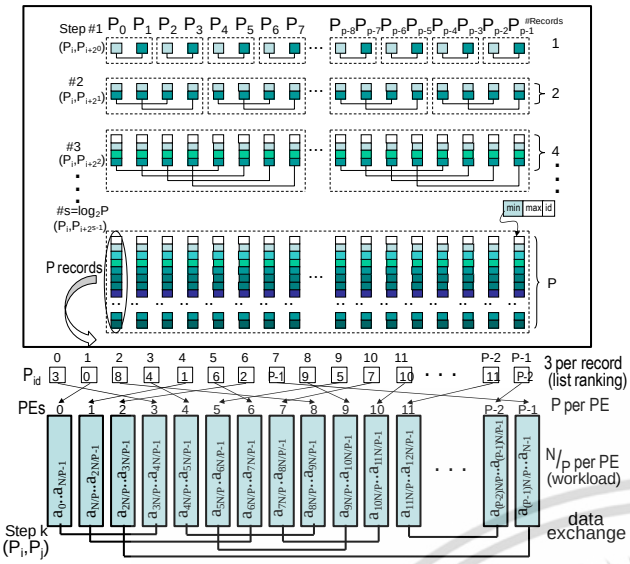


Fig.3. Constructing the *P*-record Broadcast-Checker Table [9].

Parallel communication of the *Bitonic* sort (Fig.1(a)) is a compare-exchange operation between (a_i, $a_{N/2+i}$) in (P_i, $P_{N/2+i}$) with *Bitonic* sequence (↓ and ↑), where $i = 0, 1, 2, \dots, N/2-1$, ↓ representing the increasing order ($min(a_i, a_{N/2+i})$ in P_i and $max(a_i, a_{N/2+i})$ in $P_{N/2+i}$), and ↑ representing the decreasing order ($max(a_i, a_{N/2+i})$ in P_i and $min(a_i, a_{N/2+i})$ in $P_{N/2+i}$).

For $P = N$, time complexity of the *static* communication is efficient, $O(s^2)$ or $O((\log_2 P)^2)$ where $s = \log_2 P$, since all steps of communication among *PEs* are $1 + 2 + 3 + 4 + \dots + s = s(s+1)/2$ steps.

For $P < N$, time complexity of the parallel sorting part to merge sorted results among *PEs* is $O(ws^2)$, which is processed after local sorting ($O(w \log_2 w)$) with workload $w = N/P$, since all $s(s+1)/2$ iterations of the major communication among *PEs*, are performed to exchange data (w) between (P_i, P_j) for merging results.

In 2009, we introduced the *D*ynamic *C*ommunication-*E*fficient parallel *S*orting (*DCES*) algorithm [9] to improve *static* ($s(s+1)/2$) iterations to $1, 2, 3, \dots,$ or $s(s+1)/2$ iterations dynamically. Unlike the *Bitonic*-communication pattern (Fig.1(b)), the flexible *dynamic* (P_i, P_j) parameters are formed, according to P sorted records of (min_{PE}, max_{PE}, id_{PE}) fields, ordered by increasing *min*-field, which are maintained in a "*Broadcast-Checker*" table (see Fig.3). That table is updated before starting each iteration of the *dynamic* communication by using the efficient *all-to-all* broadcast with sorted records in $O(P)$, where $P=2^s$. The construction of the efficient broadcast table requires s steps, where in each step i (= $0, 1, 2, \dots, s-1$) a number of (send/receive) records (2^i) are increased (i.e., $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow \dots \rightarrow P$ records). The time complexity of parallel constructing that table is $O(P)$ since in each step i (= $0, 1, 2, \dots, s-1$) of s steps, 2^i records are transferred in parallel and inserted in the sorted table (of size $2x2^i$ records). Therefore, total steps are $2(2^0)+2(2^1)+\dots+2(2^{s-1}) = 2(2^s-1) = 2(P-1)$. Then, to set *dynamic* (P_i, P_j), each *PE* searches in the *P*-record table (in $O(s)$) for finding its rank and set appropriate communication between (P_i, P_j).

2.2 *Data-Exchange Patterns between (P_i, P_j)*

In data exchange between (P_i, P_j), each *PE* sends its workload (w) to merge with its co-worker. Kim et al. [6] introduced three data-exchange patterns to improve communication time between (P_i, P_j), called *hold*, *swap*, and *partial* patterns. Later, the *partial* pattern was improved with fewer data exchange in 2003 [7] and 2008 [8].

Hold/Swap/Partial Patterns

In 2001, the *CEBS* strategy [6] introduced three data-exchange patterns for communication between (P_i, P_j): 1. *hold* pattern (without data exchange), 2. *swap* pattern (exchange all w elements without merging), and 3. *partial* pattern (exchange ≤ w elements with merging results). In particular, before data exchange between (P_i, P_j), each *PE* sends its local (*min*, *max*) to the co-worker and finds one of three patterns. Fig.4 illustrates communication patterns between (P_i, P_j) and their corresponding examples.

- The *hold* pattern (Fig.4(a)), the *best* case scenario occurs when two communicated *PEs* (P_i, P_j) hold local workload ($\{a_0, a_1, \dots, a_{w-1}\}$ in P_i , $\{b_0, b_1, \dots, b_{w-1}\}$ in P_j, where $a_{w-1} \le b_0$), which satisfy the merging condition ($max_i \le min_j$) and hence P_i and P_j do not need to exchange their workloads.
- The *swap* pattern (Fig.4(b)) is the *worst* case and occurs when (P_i, P_j) have to exchange their (w) workload ($\{b_0, b_1, \dots, b_{w-1}\}$ in P_i, $\{a_0, a_1, \dots, a_{w-1}\}$ in P_j, where $b_{w-1} \le a_0$) without merging (because of the condition $max_j \le min_i$).
- Otherwise, the *partial* pattern (Fig.4(c)) represents the *average* case, where (P_i, P_j) hold overlapping workload (e.g. $\{a_0, a_1, \dots, a_{k+1}, \dots, b_l\}$ in P_i and $\{a_k, \dots, b_{l-1}, \dots, b_{w-1}\}$ in P_j), and hence P_i and P_j need to exchange their partial data (≤ w elements) for merging results. For this pattern, in order to find the partial data (min_i' to max_i) in P_i, its local data (w elements) are scanned with min_j from max_i until finding min_i'. At the same time, for those (min_j to max_j') in P_j, its local data are scanned with max_i from min_j until finding max_j'.

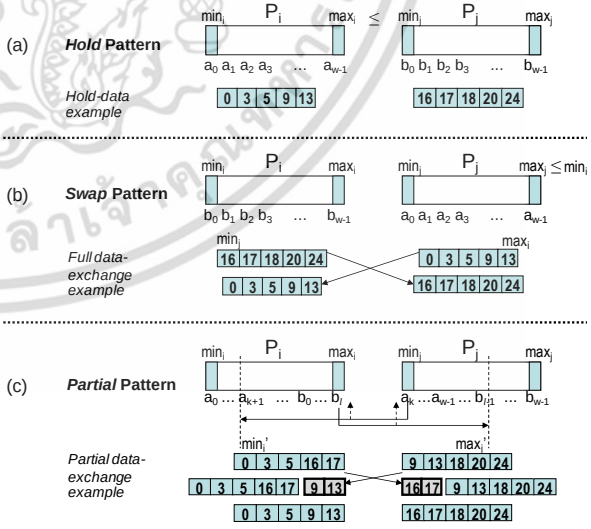


Fig.4. Three data-exchange patterns [6]: a) *Hold* pattern, b) *Swap* pattern, and c) *Partial* pattern.

Improved "Partial" Pattern

The *CSEBS* method [8] was introduced to improve the *partial* pattern (Fig.5) in data exchange by applying a *Binary* search to find min_i' in P_i and max_j' in P_j. Before data exchange between (P_i, P_j), each *PE* sends the size of its defined *partial* data to its co-worker. Thus, fewer partial data or $min(size_i, size_j) \le w$ elements are exchanged for merging between received data and local workload (w). Then, in the merging step the median merging [7], and modified merging [8], were introduced efficiently.

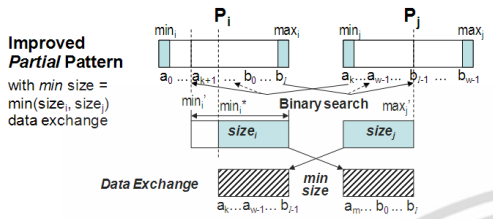


Fig.5. The improved *"Partial"* pattern with fewer partial data exchange [8].

3 The Optimized Bitonic Sorting

In this paper, we propose the optimized *Bitonic* sorting (*OBS*) algorithm (in case $P < N$) by using *midpoint*-based *dynamic* communication. Our contribution is the optimized complexity, based on applying a *better* key in *PE*-list ranking to find the right place of (P_i, P_j) dynamically. This can improve communication time significantly (i.e., fewer iterations, better synchronization per iteration, faster convergence to the final result). Time complexity of the list-ranking method is also improved from $O(P)$ to $O(s^2)$, where $P = 2^s$ (see Fig.6). In our approach, *dynamic iterations* of w-data exchange are reduced (1, 2, 3, …, or s iterations). In contrast, existing *Bitonic* sort ($P < N$) have to exchange data (w) between (P_i, P_j) in each of *static* $s(s+1)/2$ iterations, see Fig.1.

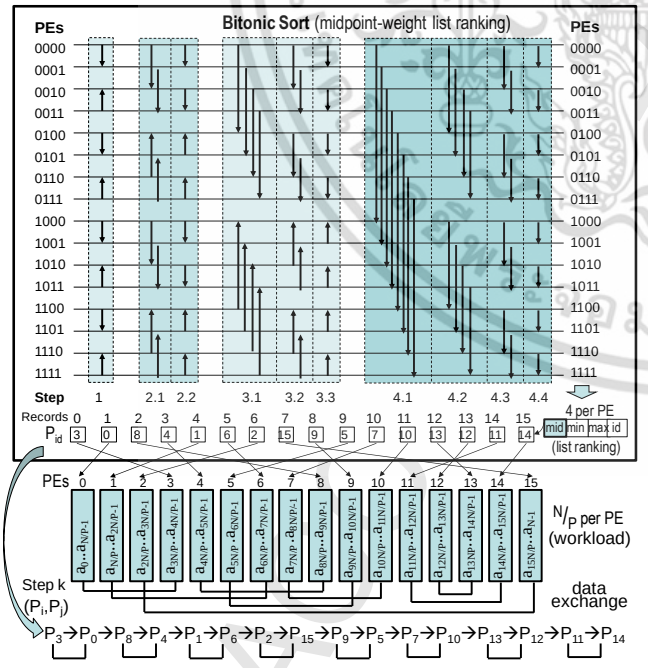


Fig.6. The optimized *Bitonic* sort (P=16) of the *midpoint* key-field.

Algorithm 2. The optimized *Bitonic* sorting (*OBS*) for $P < N$.

	Time Complexity
OBS SORT (N, P) // s = $\log_2$P	
1. **Local sort** (with workload w=N/P) // Initial process	**O(w log₂w) + O(ws)**
Each PE makes Quick sort (N/P) in parallel	
2. **Parallel merge sort** among PEs	
while (pattern-STATUS ≠ HOLD)	
2.0 All PEs compute *minpoint* = (*min*+*max*)/2	*O(1)*
Update **midpoint-list-ranking and (P_i,P_j)**	*O(s²)*
by ***Bitonic* sort** (of *midpoint* key-field) *Sec.3.1*	
if (pattern-STATUS=HOLD) call Function in STEP3	*O(s)*
2.1 **Data Exchange** between (P_i,P_j) *Sec.3.2*	
- Find pattern of data from (*min*, max) (*Hold/Swap/Partial* patterns)	*O(log₂w)*
- For *Partial* pattern, compute and exchange *local medians*	*O(1)*
- Find range of partial data from *median*	*O(log₂w)*
- Exchange *size* of partial data (before exchanged)	*O(1)*
- Send/Receive data (≤ w/2 elements)	*O(w)*
2.2 **Merging Result** in each PE(≤w) *Sec.3.3*	*O(w)*
end while // at most $\log_2$P iterations	
3. **Test to terminate sorting** *Theorem 1 (Sec.4.1)*	*O(s)+O(w)*
Apply "odd-even" pairs (P_i,P_j) to link the list	*O(s)*
if (pattern-STATUS & list-STATUS = HOLD) then terminate sorting	
else return to STEP 2	
End	

Clearly, the existing *Bitonic* sort in case $P < N$ has to transfer a large amount of data (w) in every of $s(s+1)/2$ iterations in $O(ws^2)$, whereas our optimized *Bitonic* sort can save major iterations of data transfer with a time complexity of $O(ws)$.

The proposed *OBS* (Algorithm 2) consists of two parts: 1. the local *quick* sort in $O(w \log_2 w)$ [or $O(w)$ when applying the *radix* sort] and 2. the parallel merge sort in $O(ws)$, where $s=\log_2 P$ and $w=N/P$. Our focus is in the second part. In this study, we firstly discuss the improvement in *dynamic* communication among *PEs* (in Section 3.1) with the *midpoint*-weight list-ranking. Second (in Section 3.2), we improve the minor process of data exchange between (P_i, P_j). In addition, the better synchronization of (P_i, P_j) in each iteration, according to the *midpoint*-based dynamic communication, is discussed.

3.1 *Efficient Dynamic Communication among PEs*

In *static* communication ($P < N$) [6 – 8], all *PEs* communicate in $s(s+1)/2$ steps (Fig.1(a)), but in each step data ($\le w$) between (P_i, P_j) are exchanged (Fig.1(b)), where P_i ($i < j$) stores the *min*-data set and P_j stores the *max*-data set, according to array-based *PEs' ids* and indexing of the data. In *dynamic* communication ($P < N$), the list-based *PEs* is utilized with proper (P_i, P_j) to reduce (data-exchange) steps or iterations to *1*, *2*, *3*, … , or s (in our *OBS*) or $s(s+1)/2$ (in *DCES* [9]). Table 1 shows improved major iterations between *static* and *dynamic* communications among *PEs*. In our optimized *OBS*, we focus on sorting the *midpoint* key to find the right place of (P_i, P_j) before the data ($\le w/2$) are exchanged. This can reduce the communication iterations for w-data exchange.

TABLE 1. A NUMBER OF ITERATIONS IN COMMUNICATION AMONG PES.

Methods	Comm. among PEs	Best Case	Worst Case	Update list ranking
LBM [7]	Static	s(s+1)/2	s(s+1)/2 x w	-
DCES [9]	Dynamic	1	s(s+1)/2 x w/2	2(P-1),P=2^s
OBS	**Dynamic**	**1**	**s x w/2**	**s(s+1)/2**

In the *dynamic* (*PE*-list-ranking) approach, to set *dynamic* (P_i, P_j) each *PE* must have information (min_{PE}, max_{PE}, id_{PE}) regarding other *PEs*. In *DCES* algorithm [9], the efficient *all-to-all* broadcast with *min*-field ordering was introduced in *O(P), where* $P = 2^s$, illustrated in Fig. 3. In that approach, to find its rank and set (P_i,P_j), each *PE* must search in the *P*-record list in *O(s)*. However, the *PE*-list ranking by using *min*-field as a key for the irregular data may need to process up to $s(s+1)/2$ iterations. For example, initially suppose a *PE* holds both the *maximum* and the *minimum* in its *w*-workload but the right place for the *minimum* is the first *PE* and that of the *maximum* is the last *PE* in the list.

Thus, this paper proposes a more efficient *Bitonic* sort with optimized *dynamic* communication among *PEs* ($\leq s = log_2P$ iterations) for all types of data by using a *midpoint*-weight list-ranking to set *dynamic* (P_i, P_j) in each iteration. In particular, the *PE*-list ranking and *dynamic* (P_i, P_j) are maintained by sorting *P* records (4 fields per record: mid_{PE}, min_{PE}, max_{PE}, id_{PE}) in non-decreasing order of the *midpoint* key, where $midpoint = (min_{PE} + max_{PE}) / 2$. Using the *midpoint*-weight in the list ranking can move the irregular data to the proper order within *s* steps (see Fig.7). For example, if the *minimum* and the *maximum* are initially in the same *PE*, that *PE* will be ranked close to the center first. After data exchange and list update (for next iteration), the *minimum* will have moved to the first half and the *maximum* will have moved to the second half. Within *s* iterations, the *midpoint*-weight list ranking can find the right place for them (proved in Section 4.1).

Fig.8 illustrates the process of *midpoint*-weight list-ranking to set *dynamic* (P_i, P_j) in Algorithm 2. After each *PE* makes a local sort on the workload *(w)*, in parallel all *PEs* find their (mid_{PE}, min_{PE}, max_{PE}, id_{PE}) records in *O(1)* time (Fig.8(a)), one record per *PE*, and then perform parallel list-ranking of *P* records by the *Bitonic* sort (Fig.6) on increasing *mid*-field and finally set appropriate (P_i, P_j) in Fig.8(b).

In particular, in setting *dynamic* (P_i, P_j) in *O(1)*, all P_a taking care record *a* (= *0, 1, 2,…, P-1*), performs parallel *point-to-point* communication (with "*even-odd*" pairs) to send *id*-filed of record *a* to P_b (i.e., $P_0 \leftrightarrow P_1$, $P_2 \leftrightarrow P_3$, $P_4 \leftrightarrow P_5$, …). Then, another parallel *point-to-point* communication (*P-P*) continues as follows: all P_a (a = *0, 1, 2,…, P-1*) send rank *a* and *dynamic* (P_i, P_j) to processor P_i, where *i* is an *id*-filed of record *a* and *j* is a received *id*-field (from the previous communication). The time complexity of the parallel *midpoint*-weight list-ranking to set *dynamic* (P_i, P_j) is $O(s^2)$, since (*P*-record) *Bitonic* sort performs in $s(s+1)/2$ steps. Clearly, setting the *dynamic* (P_i, P_j) for data exchange needs two steps of parallel *P-P* communications in a constant time *O(1)*. Table 2 illustrates the functions (along with time complexity and space required) of the *PE* list-ranking for assigning (P_i, P_j) in our *OBS* and *DCES* methods [9].

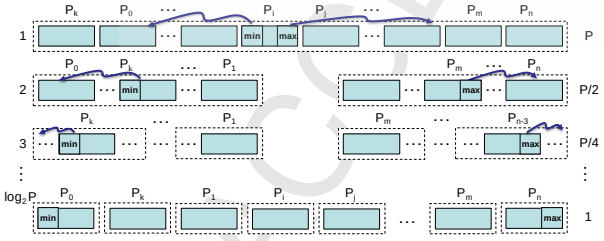


Fig.7. *Midpoint*-weight list-ranking with divide-and-conquer technique.

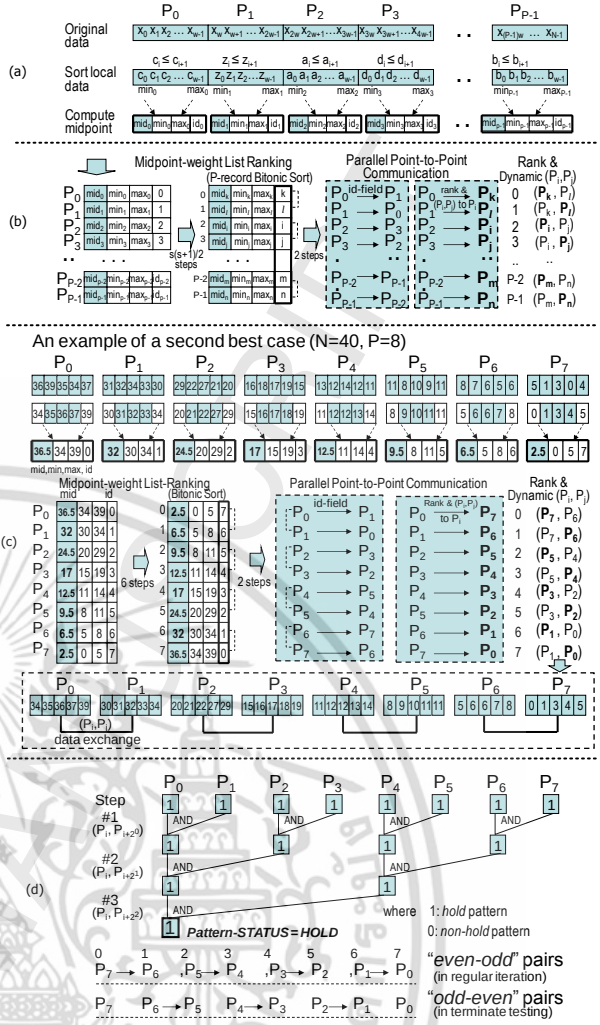


Fig.8. *Dynamic* (P_i, P_j): a) *P* records (mid_{PE}, min_{PE}, max_{PE}, id_{PE}), b) *midpoint*-weight list-ranking to set (P_i, P_j), c) an example (N=40, P=8), and d) the process of sorting termination by using parallel AND in $s=log_2P$ steps.

TABLE 2. FUNCTIONS AND TIME COMPLEXITY OF DYNAMIC (P_i, P_j).

Methods	List Ranking	Dynamic (P_i, P_j)	Space per PE
All-to-All Broadcast with Rank [9]	O(P)	O(*s*)	P
Midpoint-weight list-ranking	**O(s^2)**	**O(1)**	**1**

Example 1: In Fig.8(c), given a data set ($N = 40$ elements) to sort in parallel on *P=8 PEs* by using the *midpoint*-weight list-ranking to set *dynamic* (P_i, P_j).

After the local sort (initial process) on workload $w= 5$ in each *PE*, parallel *midpoint*-weight list-ranking are formed (such as $P_7 \rightarrow P_6 \rightarrow P_5 \rightarrow P_4 \rightarrow P_3 \rightarrow P_2 \rightarrow P_1 \rightarrow P_0$) in $s(s+1)/2 = 3(4)/2 = 6$ steps. Then all *dynamic* (P_i, P_j) pairs are set in two steps, illustrated in Fig.8(c). For instance, after list-ranking all "*even-odd*" pairs ($P_0 \leftrightarrow P_1$, $P_2 \leftrightarrow P_3$, $P_4 \leftrightarrow P_5$, $P_6 \leftrightarrow P_7$) send/receive *id*-fields in their records, such as P_0 sends *id=7* to P_1; P_1 sends *id=6* to P_0; P_2 sends *id=5* to P_3; etc. Next, in parallel P_0 sends rank 0 and ($\boldsymbol{P_7}$, P_6) to P_7; P_1 sends rank 1 and (P_7, $\boldsymbol{P_6}$) to P_6 ; P_2 sends rank 2 and ($\boldsymbol{P_5}$, P_4) to P_5; P_3 sends rank 3 and (P_5, $\boldsymbol{P_4}$) to P_4; and so on. After evaluating the status of communications (see Fig.8(d)), all (P_7, P_6), (P_5, P_4), (P_3, P_2), and (P_1, P_0) have *hold*-patterns. Then the list status =

HOLD, and hence sorting is terminated. When sorting by other algorithms [7, 9], the *dynamic DCES* [9] needs $2(P-1) = 14$ steps (for broadcasting with P ordered records) before status = *HOLD* is achieved, while the *static LBM* sort [7] requires $s(s+1)/2 = 6$ iterations, each of which needs data exchange ($w=5$) in *swap*-pattern (~30 steps).

Sorting such data shows that our *dynamic* approach can convert *worst* case of *static* communication (or *swap* pattern in $O(ws^2)$) to the *best* case (or *hold* pattern in $O(s^2)$), where $s = \log_2 P$ and $w = N/P$, discussed in Section 3.2 and proved in Section 4.2. Moreover, our optimized *OBS* can support sorting on scalable parallel and distributed systems (for large P) efficiently, as illustrated in case $N = 2^{20}$ (or *1M*) on $P = 256$. In this example, our new *OBS* strategy can completely sort in $s(s+1)/2$ ~36 steps (after updated *PE*-list ranking without data exchange), whereas the *dynamic DCES* [9] needs $2(P-1)$ ~510 steps (without data exchange), but the *static LBM* [7] requires 36xw ~147,456 steps (since each of 36 iterations needs $w=2^{20}/256=4096$ elements of data exchange).

Finally, in order to terminate sorting, all P *PEs* test terminating statuses by applying parallel *AND* (1: *hold* and 0: not *hold*) in s steps, illustrated in Fig.8(d). If the result of parallel *AND* in P_0 is 1 (all patterns of (P_i, P_j) are "*hold*" patterns), then *pattern-STATUS=HOLD*; otherwise *pattern-STATUS*=0 (or *not HOLD* since at least one *PE* have other patterns). In case of *pattern-STATUS=HOLD*, the sorting is terminated if the *list-STATUS* is also given as *HOLD* (by testing "*odd-even*" pairs), proved in *Theorem 1* (Section 4.1).

3.2 *Efficient Data Exchange between (P_i, P_j)*

Our optimized *Bitonic* sort (*OBS*) has improved the following data-exchange patterns (Fig.9): 1. *hold* (without data exchange), 2. efficient *swap* (without data exchange except the pointer), and 3. efficient *partial*-1 and *partial*-2 (≤ w/2-data exchange), according to *dynamic* (P_i, P_j).

- In a *hold* pattern, the *best* case, there is no data exchange because of the condition ($max_i \le min_j$), identified in $O(1)$.
- In an efficient *swap* pattern, another *best* case of our optimized *OBS*, there is no data exchange rather there is pointer exchange (after updated *PE*-list), where the condition ($max_j \le min_i$) is verified in $O(1)$.
- In an efficient *partial* pattern, the *average* case, we split the *partial* pattern into 2 groups for efficient testing. In the *partial-1* pattern (for boundary searching), the condition in P_i is ($min_i=a_0 \le min_j \le a_1$) or those in P_j are either ($b_{w-2} \le max_i \le max_j=b_{w-1}$) or ($max_i \ge max_j=b_{w-1}$) identified in $O(1)$. In the *partial-2* pattern (for internal *Binary*-searching), the partial data in P_i are (min_i' to max_i) and those in P_j are (min_j to max_j') by applying a *Binary* search to find the min_i' or max_j' (in $O(\log_2 w)$).
  - For P_i, if $min_j = x_{mid}$ then $min_i' = x_{mid+1}$. For ($min_j < x_{mid}$), if $min_j \ge x_{mid-1}$ then $min_i' = x_{mid}$; otherwise call a *Binary* search (min_i to x_{mid-1}). For ($min_j > x_{mid}$), if $min_j \le x_{mid+1}$ then $min_i' = x_{mid}$; otherwise call a *Binary* search (x_{mid+1} to max_i).
  - For P_j, if $max_i = y_{mid}$ then $max_j' = y_{mid-1}$. For ($max_i < y_{mid}$), if $max_i \ge y_{mid-1}$ then $max_j' = y_{mid}$; otherwise call a *Binary* search (min_j to y_{mid-1}). For (max_i y_{mid}), if $max_i \le y_{mid+1}$ then $max_j' = y_{mid}$; otherwise call a *Binary* search (y_{mid+1} to max_j).

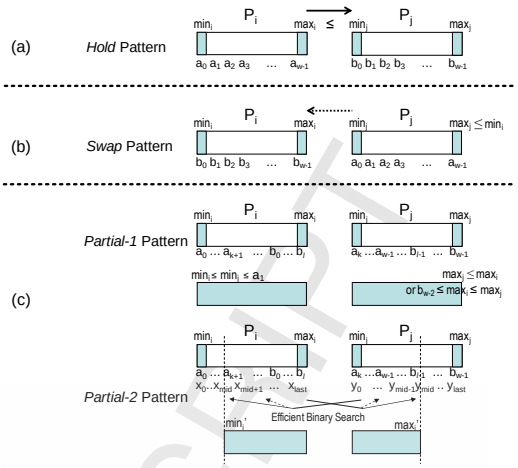


Fig.9. Three data-exchange patterns: a) *Hold* pattern, b) efficient *Swap* pattern, c) *Partial-1* | *Partial-2* pattern.

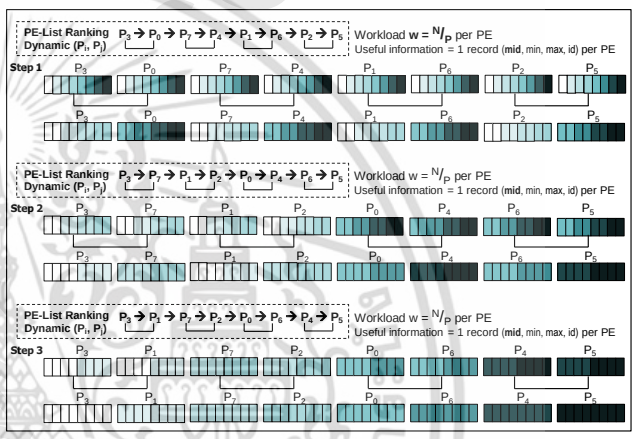


Fig.10. An example of consistency in data-exchange patterns in each iteration.

Like the *dynamic DCES* [9], the advantage of our optimized *OBS* with *dynamic* (P_i, P_j) is that it can convert *swap* pattern (*worst* case in *static* (P_i, P_j)) to *hold* pattern (*best* case in the *dynamic* approach) after updated the *PE*-list. In addition, our *OBS* strategy achieves another advantage in better synchronization during data exchange in each iteration, see Fig.10. Applying the *better* key such as the *midpoint*-weight in *PE*-list-ranking to find the right place of (P_i, P_j) before w-data exchange yields more consistency of data-exchange patterns in each iteration and hence improves the synchronization time. In *static* communication [6 – 8] of $s(s+1)/2$ iterations, in each iteration although some *PEs* of (P_i, P_i) achieve *hold* patterns in $O(1)$, they must wait for other processors that are busy in doing *swap* or *partial* patterns in $O(w)$, before starting the next process, because of the synchronization requirement. For example, Fig.10 shows the sorting in s steps of a large amount of data (of size N) on $P=8$, w-workload ($N/8$). Suppose after workload partitioning and local sorting, the result of *PE*-list ranking is $P_3 \to P_0 \to P_7 \to P_4 \to P_1 \to P_6 \to P_2 \to P_5$. In step 1, all ($P_i$, P_j) do the *partial* data-exchange pattern. In the next step, our approach can find the right place of *PEs* for (P_i, P_j) data exchange, which are $P_3 \to P_7 \to P_1 \to P_2 \to P_0 \to P_4 \to P_6 \to P_5$. Lastly in step 3, after data exchange between (P_i, P_j) from the *PE*-list $P_3 \to P_1 \to P_7 \to P_2 \to P_0 \to P_6 \to P_4 \to P_5$, the sorting is terminated.

Improved "Partial" Pattern of our OBS

The *OBS* technique improves the *partial* pattern for fewer data exchange between (P_i, P_j) in efficient time, based on *min* size and *median* computing (Fig.11(a)), as follows: Before finding min_i" (in P_i) and max_j" (in P_j), each of (P_i, P_j) computes the *median* of four local medians (med_{i1}, med_{i2}, med_{j1}, med_{j2}), where (med_{i1}, med_{i2}) are in P_i and (med_{j1}, med_{j2}) are in P_j. Note: computing the *median* of two local medians of partial data in P_i & P_j (med_{i1}, med_{j1}) is close to the computed *median* of (P_i, P_j)'s workloads (for random data). But for skewed *partial* data, applying the *median* of local medians from the local w-workload in P_i and P_j (med_{i2}, med_{j2}) yields a more accurate *median*. Thus, to support arbitrary data types, the combination of the four medians are computed in our *OBS* strategy. After computing *the median* of (med_{i1}, med_{i2}, med_{j1}, med_{j2}), fewer partial data are defined, according to ($size_i$, $msize_i$) of P_i and ($size_j$, $msize_j$) of P_j, where *msize* is the amount of partial data, according to the *median*. If ($msize_i$ = $msize_j$) we obtain the correct *median* and fewer partial data (of *minimum* size = $msize_i$) are exchanged. Otherwise ($msize_i \neq msize_j$) the amount (or size) of the partial data exchange is the *minimum* between *min* ($size_i$,$size_j$) and *max* ($msize_i$, $msize_j$).

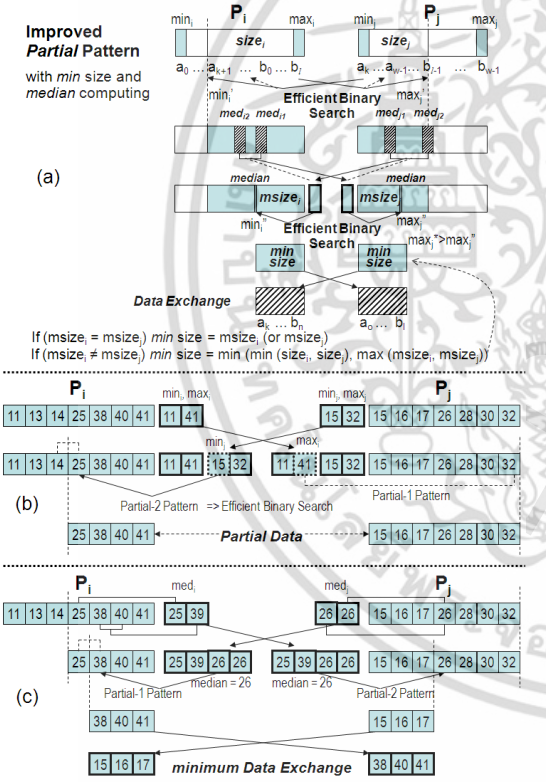


Fig.11. a) Improved *partial* pattern with *min* size and *median* computing, b) examples of *partial-1*, *partial-2* patterns, and c) median-based data exchange.

For example, Fig.11(b) illustrates the processes of *partial-1* / *partial-2* patterns between (P_i, P_j) with a workload w=7 elements. First, each of (P_i, P_j) identifies one of the three patterns (*Hold* / *Swap* / *Partial-1* | *Partial-2*). For that data, P_j applies *partial-1* pattern since the condition $max_i(41) > max_j(32)$ then max_j'= 32. In parallel, P_i applies the *partial*-2 pattern with an efficient *Binary* search since the conditions in P_i do not satisfy *hold/swap/partial-1* patterns. First, P_i computes x_{mid}=25 that satisfies condition $min_j(15) < x_{mid}$. Before reapplying the *Binary* search ($min_i(11)$ to $x_{mid-1}(14)$), P_i tests $min_j(15) > x_{mid-1}(14)$ and hence min_i'= x_{mid} = 25 and the partial data defined in P_i, are {25, 38, 40, 41}.

After defining the partial data of (P_i, P_j) from (*min*, *max*) of the co-worker, the next step (Fig.11(c)) is to apply the improved *partial* pattern. P_i computes med_{i2} (from workload) = 25 and med_{i1} (from the partial data {25, 38, 40, 41}) = (38+40)/2 = 39, while P_j computes med_{j2} = 26 and also med_{j1} =26. After the exchange of local medians between (P_i, P_j), each *PE* then computes the *median* of the local medians (25, 26, 26, 39) = 26. Then the efficient *Binary* search for fewer partial data is applied according to *median* = 26, which are {38, 40, 41} in P_i and {15, 16, 17} in P_j. Finally, the (P_i, P_j) exchange ($msize_i$, $msize_j$) = (3, 3) and ($size_i$, $size_j$) = (4, 7) to compute the amount of partial data exchange (3 elements) since $msize_i$ = $msize_j$ = 3. Clearly, the result of this example shows that the *median* of the four local medians can be used to estimate the computed *median* for skewed partial-data.

Inverted Data Exchange in "Partial" Pattern of our OBS

Like the *dynamic DCES* method [9], another advantage of our optimized *OBS* strategy is that no more than half of *N/P*-workload (or ≤ w/2) are exchanged since for *min* size > w/2, the inverted partial data are exchanged (illustrated in Fig.12(a)). Then, (P_i, P_j) *will* handle the opposite value of data, which are *max*-data set in P_i and *min*-data set in P_j. However, after updating the *PE*-list ranking, only the pointer is exchanged from ($P_i \rightarrow P_j$) to ($P_j \rightarrow P_i$).

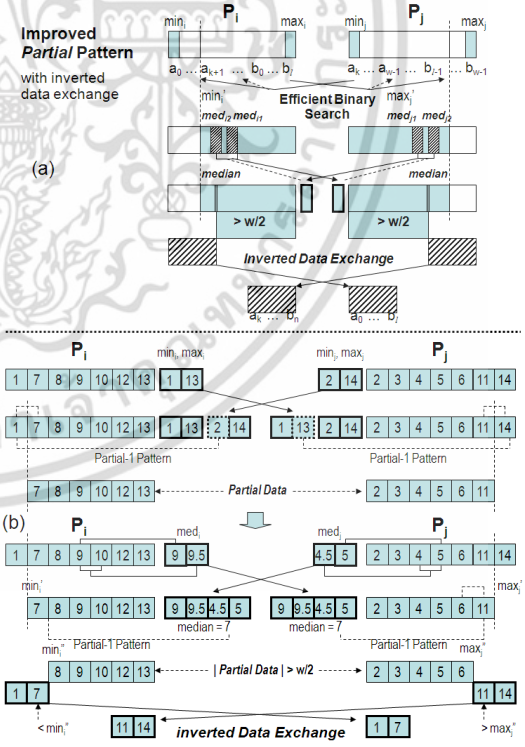


Fig.12. a) Improved *partial* pattern with inverted data exchange and b) an example of inverted data exchange.

For example, Fig.12(b) illustrates the inverted data exchange in (P_i, P_j) on $w = N/P = 7$, where the partial data in P_i are {8, 9, 10, 12, 13} with $min_i''= 8$ and in P_j are {2, 3, 4, 5, 6} with $max_j''= 6$ and $msize_i = msize_j = 5 > w/2$. Therefore, the inverted partial data (of inverted *min* size = $w–5 = 2$) are exchanged ({1, 7} in P_i and {11, 14} in P_j).

3.3 *Efficient Result Merging in each PE*

Efficient merging of our *OBS* strategy (defined in Fig.13) are 1) *block* merging and 2) *1-1* merging.

- In *block* merging (Fig.13(a)), if $max_j'' \le min_i'$ in P_i or $min_j'' \ge max_j'$ in P_j, the block of partial data are replaced for P_i from min_i' (left to right) & for P_j from max_j' (right to left).
- In *1-1* merging (Fig.13(b)), if the merging is not *block* merging then each element of the two partial data sets are compared and merged one by one (in increasing order) for P_i from min_i' (L to R) and compared one by one (in decreasing order) for P_j from max_j' (R to L).

Similarly for the inverted data exchange, efficient merging is also achieved through inversion.

- In inverted *block* merging (Fig.14(a)), a block of partial data is replaced for P_i from max_i (right to left) and for P_j from min_j (left to right).
- In inverted *1-1* merging (Fig.14(b)), each element of two partial data sets are compared one by one (with decreasing order) for P_i from max_i (right to left) or compared and merged one by one (with increasing order) for P_j from min_j (left to right).

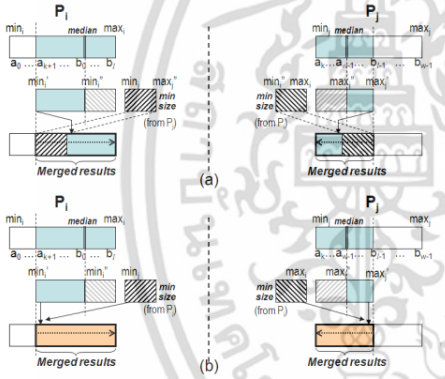


Fig.13. a) Block merging and b) 1-1 merging.

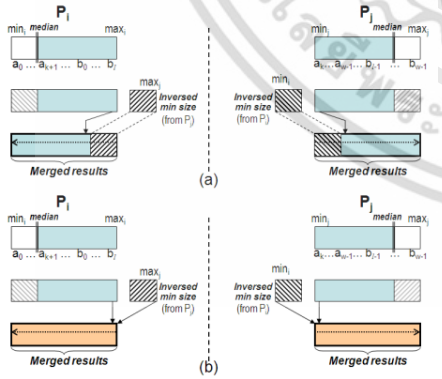


Fig.14. a) Inverted block merging and b) Inverted 1-1 merging

4 Proof of Correctness and Complexity Analysis

This section provides the proof of correctness of our optimized *Bitonic* sort (*OBS*) algorithm, which are; 1) the correctness of the *dynamic* communication among *PEs* (using efficient *midpoint*-weight list-ranking) as given in Section 4.1 and 2) the correctness of the *median*-based data-exchange pattern between (P_i, P_j) as given in Section 4.2. Finally, the time complexity of the proposed *OBS* algorithm is analyzed in Section 4.3

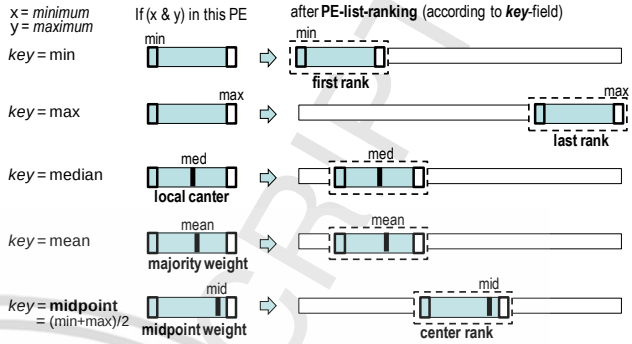


Fig.15. Effect of *key*-fields (i.e., *midpoint*, *min*, etc.) in *PE*-list ranking.

TABLE 3. EFFECTS OF FIVE KEYS IN PE-LIST RANKING.

key-field in ranking	Time	Sorted data (best)	Skewed data (worst)	Random data (average)
min [9]	O(1)	1	s(s+1)/2	1 to s(s+1)/2
max	O(1)	1	s(s+1)/2	1 to s(s+1)/2
median	O(1)	1	s(s+1)/2	1 to s(s+1)/2
mean	O(N/P)	1	s(s+1)/2	1 to s(s+1)/2
midpoint	**O(1)**	**1**	**$s=\log_2 P$**	**1 to s**

4.1 *Correctness of Dynamic Communication*

The *dynamic* communication in our *OBS* strategy is focused on the efficient *midpoint*-weight list-ranking, which can order the *PE*-list (with *key*=*midpoint*) efficiently.

Table 3 and Fig.15 present five *key*-fields (*min*, *max*, *median*, *mean*, *midpoint*) and their effects on a variety of data, where the *good key* must be computed in *O(1)* and can move the irregular data to the right *PE* within $s=\log_2 P$ steps. Each of the *key*-fields can be computed in *O(1)*, except the *mean* (*O(w)*). For special data (in *best* case), all *key*-fields offer one iteration sorting. For irregular data (i.e., the *minimum* & *maximum* are in the same *PE*), the *PE*-list ranking can rank such a *PE* to the center of the list first (because of the *midpoint* weight) before moving each of irregular data to the proper order in ≤ *s* steps, as shown in Fig.7 (Section 3.1), while other *key*-fields cannot set that order efficiently. For example, using the *min*-field [9] will rank that *PE* in the first order (*best* for the *minimum* but *worst* for the *maximum*), and hence moving the *maximum* to the last order in the *PE*-list may take $s(s+1)/2$ iterations. When using the *max*-field, sorting may return after $s(s+1)/2$ iterations as the sorting result of using the *min*-field. Using the *median*-field represents the local center but after *PE*-list ranking it is not the global center, and hence sorting may take up to $s(s+1)/2$ iterations. The effect of using the *mean*-field is better than using the *median* but the *mean* is computed with the *majority*-weight, which is not better than the *midpoint*-weight. Finally for random data, the *midpoint*-weight list-ranking can set iterations between 1 and *s*, while others require more iterations between 1 and $s(s+1)/2$.

Therefore, the better weight of the *PE*-list rank is the *midpoint* or *mid* = *(min + max)*/2, where *min* is the *minimum*

value of local workload and *max* is the *maximum* value. This parameter is the *best* key for the *PE*-list ranking since it can find the right place of *PEs* at most *s* steps, according to the *best* effective weight of the *midpoint*.

Correctness of the Best Case: There exist some *best* cases (one iteration of communication among *PEs*) such as

- Input data are already ordered in non-decreasing order (i.e., all *PEs* hold their data without data exchange), or
- Input data are ordered in decreasing order. After local sort and *PE*-list ranking, the output data are properly sorted (see this case in example 1 (Section 3.1)).

Correctness of the Worst Case: In practice, most of the input data are random, and sorting on those data can finish in 1, 2, 3, … ,or *s* iterations but for the irregular data (*worse* case) may need *s* iterations. In our *OBS* algorithm, since the *P*-record is sorted by the weight of *midpoint = (min + max) / 2*, in the input dataset if the *minimum* (*x*) and the *maximum* (*y*) of the data set is in the same *PE*, first its rank is set close to the center of the *PE*-list (see Fig.7 in Section 3) and for next iteration *x* and *y* will be moved to appropriate *PEs* (i.e., move *x* to the first half and *y* to the second half). The same process is repeated (for communication among *P*/2, *P*/4, *P*/8,…,4, 2 *PEs*), until the data are sorted. This *midpoint*-weight list-ranking may take up to *s* iterations to move *x* to the first *PE* and *y* to the last *PE* (in the list).

Example 2: Given a data set (*N = 40* elements) to be sorted on *P = 8 PEs* with workload *w=5*. Assume input data contain irregular data (i.e., *minimum* (1) and *maximum* (40) are initially assigned to the processor P_0).

After assigning workload (*w*=5 elements) to each *PE*, all *PEs* perform local sort (see Fig.16(a)). Then each P_i creates a record of 4-field (*mid*, *min*, *max*, *id*), where *mid* = (*min*+*max*)/2. After finished *PE*-list ranking, P_0 (with *min*=1, *max*=40, *mid*=(1+40)=20.5) will be ranked in the center. After iteration 1 (data exchange and result merging) *PE*-list ranking can move the *minimum* (1) from P_0 (in center) to the 1st order and move the *maximum* (*y*=40) from P_0 (in center) to P_3 (the 6^{th} order).

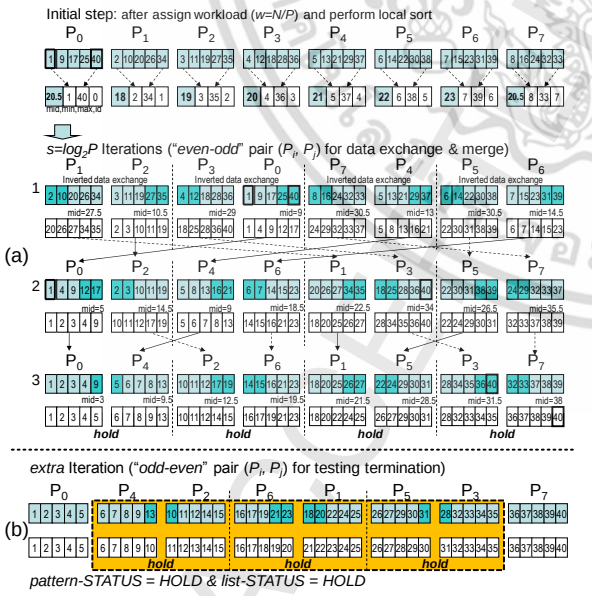


Fig.16. a) An example of irregular-data processing: a) log_2P iterations of *even-odd* (P_i, P_j) and b) the final *odd-even* (P_i, $P_{j'}$) *for termination*.

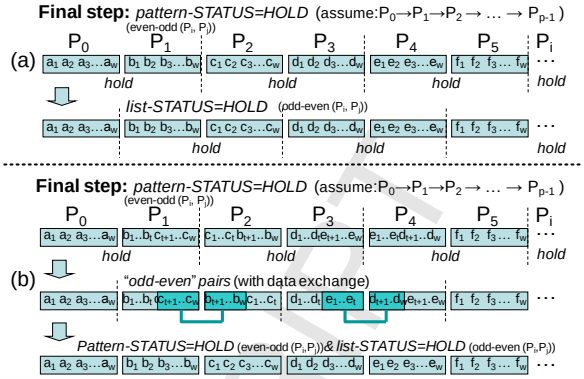


Fig.17. The *pattern*-STATUS and *list-STATUS* in termination process: a) Case 1 (without data exchange) and b) Case 2 (with data exchange).

After iteration 2, *y*=40 is removed to P_3 (the 7^{th} order). After data has been exchanged in iteration 3, *y*=40 is moved to the last order and all *PEs* have "*hold*" statuses. In the final step, (Fig.16(b)) special "*odd-even*" pairs for testing (P_i, P_j) between two pairs of the *PE*-list is performed (with data exchange). Finally, the *pattern-STATUS = HOLD* and *list-STATUS = HOLD*, then the sorting is correctly terminated.

Theorem 1: If *pattern-STATUS = HOLD* and *list-STATUS = HOLD*, then intermediate results (of all *PEs* in the *PE*-list ranking) are correctly sorted.

Proof. After *pattern-STATUS = HOLD* (≤ $s=log_2P$ iterations) in all pairs (P_i, P_j) is obtained, the last iteration is set with "*odd-even*" pairs (for testing the linked-list order among *PEs*).

Case 1: *pattern-STATUS=HOLD* & *list-STATUS=HOLD* and results are properly sorted (without data exchange).

See Fig.17(a), the *hold*-status of the "*even-odd*" pair means that the partial data are sorted between (P_i, P_j), such as $P_0 \rightarrow P_1$, $P_2 \rightarrow P_3$, $P_4 \rightarrow P_5$, etc., and the successive *hold*-status of the "*odd-even*" pair means the data are sorted among *PEs* ($P_1 \rightarrow P_2$, $P_3 \rightarrow P_4$, $P_5 \rightarrow P_6$, etc.).

Case 2: *pattern-STATUS=HOLD* & *list-STATUS=HOLD* and results with data exchange are properly sorted.

See Fig.17(b), after data exchange in the last iteration, the list-ranking is stable with the same order while the *pattern-STATUS=HOLD* and *list-STATUS=HOLD*, then the proof of the next process is similar to that of case 1.

4.2 *Correctness of Data Exchange between (P_i,P_j)*

In this section we will prove that the condition(s) in each of the three data exchanged patterns (*hold* / *swap* / *partial-1* | *partial-2*) between (P_i, P_j) are correctly defined (see Fig.18).

Correctness of the Hold pattern

Both (P_i, P_j) have *hold* patterns (*best* case) if the condition is ($max_i \le min_j$). Since (P_i, P_j) in *dynamic* pairs is set according to the *min*-field ($min_i \le min_j$) and the local workload (*w=N/P* elements) in P_i and P_j are locally sorted, clearly if ($max_i \le min_j$) then the data are sorted between two *PEs* in the order of $P_i \rightarrow P_j$ (without data exchange) in *O(1)* time.

Correctness of the efficient Swap pattern

Both (P_i, P_j) have *swap* patterns if the condition is ($max_j \le min_i$). Since local workload (*w*) in P_i and P_j are locally sorted and after the updated list ranking, the *swap* pattern is converted to the *hold* pattern (*best* case), and hence the data are sorted between two *PEs* in the order of $P_j \rightarrow P_i$ (without data exchange but only pointer exchange) in *O(1)* time.

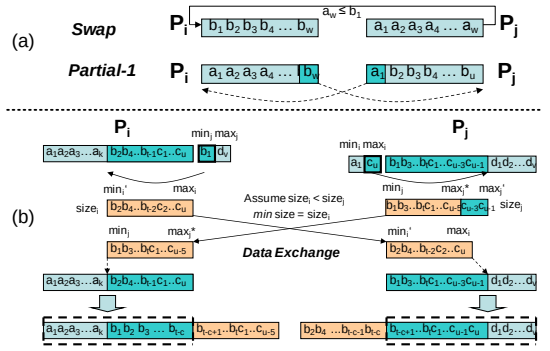


Fig.18. Conditions of a) *swap* and *partial-1* patterns and b) *partial-2* pattern.

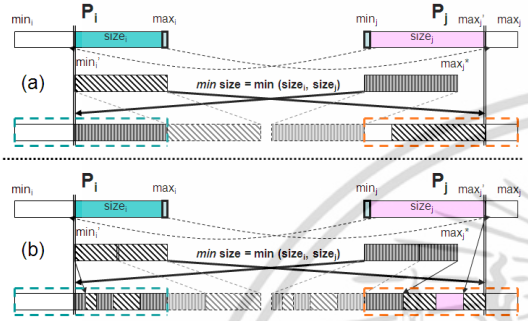


Fig.19. a) *Non-overlap merging* and b) *Overlap merging* in *partial-2* pattern.

Correctness of the Partial-1 pattern

In P_i or P_j, the *partial-1* pattern is applied for "boundary searching" if the condition is not *hold* or *swap* but ($min_i = a_1 \le min_j \le a_2$) in P_i or either ($max_j \le max_i$) or ($b_{w-1} \le max_i \le max_j = b_w$) in P_j (see Fig.18(a)). In this case, the merging result in (P_i, P_j) may need overlap data starting from the 1^{st} or 2^{nd} elements (of P_i and P_j), performed in P_i, or similarly the last or 2^{nd} last elements (of P_i and P_j) that are carried in P_j.

Correctness of the Partial-2 pattern

P_i or P_j apply the *partial-2* pattern for "internal *Binary* searching" if the condition is not *hold* / *swap* / *partial-1*. This *partial* pattern applies the efficient *Binary* search ($O(\log_2 w)$) for identifying partial data (min_i' to max_i) in P_i and (min_j to max_j') in P_j, illustrated in Fig.18(b).

To find overlap data between (P_i, P_j), P_i uses min_j to find partial data (min_i' to max_i), where $min_j < min_i'$ and P_j uses max_i to find partial data (min_j to max_j'), where $max_i > max_j'$. Next is to prove that using the minimum amount of *partial* data exchange *min* size = *min* ($size_i$, $size_j$) provides the correct range for merging, illustrated as follows.

Case 1: Non-overlap Merging (Fig.19(a))

Assume the $size_i \le size_j$ and (P_i,P_j) are assigned to the *min*-data set and *max*-data set. The merging results are

- P_i sends the partial data ($size_i = \lfloor min_i' \text{ to } max_i \rfloor$, where $min_i' > min_j$) to P_j, and hence requires at most $size_i$ elements from P_j. In non-overlap merging, after receiving (from P_j) the *min-value* of the partial data ($size_i = min_j$ to $max_j{}^* \rfloor$, where $max_j{}^* < min_i'$), their blocks of partial data are replaced (in P_i) from min_i'. In some cases, the remaining (larger) values in P_j (since $size_j > size_i$) are not needed in P_i (for *min*-data set). Thus, P_i receives proper range of data from P_j and *min*-data set merging in P_i is correct.
- P_j sends partial data ($size_i = \lfloor min_j$ to $max_j{}^* \rfloor$, where max_j' and $max_j{}^* < max_i$) to P_i. In non-overlap merging, after receiving (from P_i) the *max-value* of the partial data ($size_i = \lfloor min_i'$ to $max_i \rfloor$, where $min_i' > max_j'$), their blocks of partial data are replaced (in P_j) from max_j'. Clearly, the remaining (smaller) values in P_i (since $size_j > size_i$) are not needed in P_j (for *max*-data set). Thus, P_j receives proper range of partial data from P_i and *max*-data set merging in P_j is correct.

Case 2: Overlap Merging (Fig.19(b))

Assume that the $size_i \le size_j$ and (P_i, P_j) are set for a *min*-data set and a *max*-data set. The merging of results are

- P_i sends partial data ($size_i = \lfloor min_i'$ to $max_i \rfloor$, where $min_i' > min_j$) to P_j, and requires at least $size_i$-element partial data from P_j. In overlap merging, after receiving (from P_j) the *min-value* of partial data ($size_i = \lfloor min_j$ to $max_j{}^* \rfloor$), they are compared and merged 1-1 (in P_i) from min_i' to the right. However, some (larger) values of (min_j to $max_j{}^*$) are not needed for merging in P_i (for *min*-data set). Thus, P_i receives the proper range of partial data from P_j and the *min*-data set merging in P_i is correct.
- P_j sends partial data ($size_i = \lfloor min_j$ to $max_j{}^* \rfloor$, where max_j' and $max_j{}^* < max_i$) to P_i. In overlap merging, after receiving (from P_i) the *max-value* of the partial data ($size_i = \lfloor min_i'$ to $max_i \rfloor$), they are compared and merged 1-1 (in P_j) from max_j' to the left. Clearly, some (smaller) values of (min_i' to max_i) are not needed for merging in P_j (for *max*-data set). Thus, P_j receives the proper range of partial data from P_i and the *max*-data set merging in P_j is correct.

Note that in the case of $size_i > size_j$, the proof is defined in a similar way to the above cases ($size_i \le size_j$).

Thus in both cases, the data exchange between (P_i, P_j) by using *min* size = *min*($size_i$, $size_j$) provide an appropriate range of the partial data for correct merging, since at most the *min* size (received) elements are merged in each of (P_i, P_j), while the remaining non-merged elements must be merged in its corresponding co-worker.

Next, in order to reduce the amount of data exchange, the *OBS* strategy provides improved *partial* pattern with fewer data exchange by *min* size and *median* computing.

Correctness of improved "Partial" pattern

To reduce the data exchange fewer than *min*($size_i$, $size_j$), we need the *median* of the two data sets in (P_i, P_j), where the *median* of a sorted data set (a_1, a_2, … , a_w , b_1, b_2, … , b_w) of (P_i, P_j) is (a_w+ b_1)/2. However because of overlap data in (P_i, P_j), their *median* cannot be computed before the data ($w = N/P$ elements) in each of (P_i, P_j) are exchanged. For efficient computation, our optimized *OBS* method computes the estimated *median* from the four local medians of the local data in each *PE* (to handle not only the random data but also the skewed data). Before fewer data are exchanged, four local medians between (P_i, P_j) are exchanged.

med_{i1} is local median of partial data ($\le w$) in P_i,
med_{i2} is local median of local workload ($= w$) in P_i,
med_{j1} is local median of partial data ($\le w$) in P_j,
med_{j2} is local median of local workload ($= w$) in P_j.

The *median* of the sorted (med_{i1}, med_{i2}, med_{j1}, med_{j2}) is the correct *median* if the sizes of fewer partial data ($msize_i$, $msize_j$), according to the *median* in P_i and P_j are equal ($msize_i = msize_j$). Note: our *OBS* strategy yields $msize_i = msize_j$ for most of data types but if $msize_i \ne msize_j$ then *Case 2* is applied.

Case 1: $msize_i = msize_j$

In this case, after the minimum data are exchanged and

merged in each *PE*, clearly the estimated *median* is $(a_w+b_1)/2$, illustrated in Fig.20(a). In addition, the estimated *median* can be either a_w (last element in P_i) or b_1 (first element in P_j). See the corresponding examples in Section 3.2. Therefore, the data exchange between (P_i, P_j) by using the *min* size = $msize_i$ (=$msize_j$) provide the correct range of the partial data (from the *median*), required for merging, (since exactly *min* size (received) elements are merged in each of (P_i, P_j)).

Case 2: $msize_i \neq msize_j$

If $msize_i < msize_j$ then the estimated *median* is greater than the computed *median* and hence the appropriate range of the partial data is $msize_j$. If $msize_i > msize_j$ then the estimated *median* is lower than the computed *median* and then the appropriate range of the partial data exchange is $msize_i$. Thus, for ($msize_i \neq msize_j$) the partial data exchange should be $max(msize_i, msize_j)$ to cover the correct range. However, in some cases $max(msize_i, msize_j)$ is greater than $min(size_i, size_j)$. As proved before in the correctness of *partial-2* pattern, the data exchange is $min(size_i, size_j)$. Therefore, using *min* size = *minimum* of $min(size_i, size_j)$ and $max(msize_i, msize_j)$ provides an appropriate range of fewer partial data, required for merging (where $msize_i \neq msize_j$).

Correctness of inverted Data exchange of "Partial" pattern

On the other hand, to maintain the proper partial data exchange, if the *min* size > $w/2$ elements then the inverted data exchange is processed with the inverted *min* size = w - *min* size (< $w/2$ elements). The proof of the inverted case is similar to that of the common data exchange, except all processes are inverted (i.e., inverted *min* size, inverted data exchange, and inverted merging, etc.), as illustrated in Fig.20(b). After merging, P_i stores *max*-data set and P_j stores *min*-data set, and then the pointer ($P_i \rightarrow P_j$) is exchanged ($P_j \rightarrow P_i$) after the *PE*-list ranking is updated.

4.3 *Time Complexity Analysis*

For the time complexity analysis of our *OBS* algorithm (for $P < N$), we perform the analysis of the *best* case, the *worst* case, and the *average* case, as follows:.

Best Case Time Complexity: $O(w) + O(s^2)$

The *best* case of sorting ($P < N$) occurs when the data are already sorted in increasing order. Thus, after the local sort of the w-workload in $O(w \log_2 w)$ by the *quick* sort (or $O(w)$ by the *radix* sort), the *Bitonic* sort of *midpoint* keys in the *midpoint*-weight list ranking is applied in $O(s^2)$ before terminating the sort.

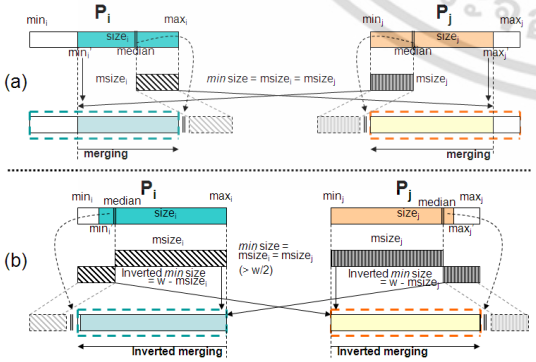


Fig.20. a) Median-based data-exchange and merging and b) Inverted data-exchange and merging.

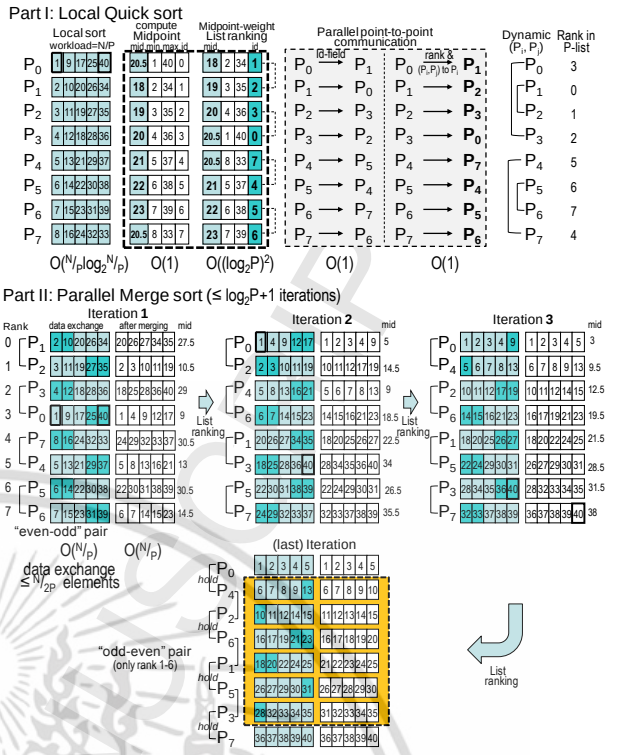


Fig.21. An example of the *worst* case with irregular data ($s=log_2P$ iterations).

Moreover, one of the *worst* cases in *static* sorting approaches is the second *best* case in our *dynamic* sorting approach. Such a case occurs when the data are already sorted in the decreasing order. Like the *best* case of our *OBS* algorithm, after the local sort of the w-workload, the *midpoint*-weight list-ranking is applied (with only pointers exchange) in $O(s^2)$ before finishing the sort.

Worst Case Time Complexity: $O(w) + O(ws)$

The *worst* case may occur with irregular data, as illustrated in example 2 (Fig.16) in Section 4.1 and Fig.21 in this section, which requires at most s iterations and data exchange (≤ $(w/2)$ in each iteration in $O(ws)$ and hence $O(w)_{radix\text{-}sort} + O(ws)_{OBS\ merge}$, including the initial local sort.

In particular, the time complexity of each function in setting *dynamic* (P_i, P_j) for result merging is illustrated as follows:

- In parallel, all *PEs* compute their *key*-sorting records (*mid*, *min*, *max*, *id*), one record per *PE*, in $O(1)$ time.
- All *PEs* perform parallel list ranking (by *Bitonic* sort), according to the *mid*-field of *P* records in $s(s+1)/2$ steps ($O(s^2)$) and then set two steps of the parallel *point-to-point* communication to assign *dynamic* (P_i, P_j) in $O(1)$.

For example ($P=8$ and $s=log_2P=3$) in Fig.21, iterations 1-3 of computing these functions (i.e., (P_i, P_j) for data exchange and result merging) are repeated. Finally, the last iteration is conducted to terminate sorting.

After data exchange ($O(w)$) in each iteration, the *midpoint*-weight list-ranking is updated and then the termination-status is verified (see *Theorem* 1). This example shows the *worst* case that can complete sorting in s iterations with time complexity $O(ws)$. Theoretically, in the total time complexity $O(ws)$, only the dominate time ($O(w)$)

in each iteration (of $s \leq log_2P$ iterations) is presented, where $s=log_2P$ and $w=N/P$. Note that all of other functions in each iteration include 1. compute *midpoint* in $O(1)$, 2. *Bitonic* sort in $O(s^2)$, 3. *Binary* search in $O(log_2w)$, and 4. data exchange in $O(w)$, In practice, response-time effects of these functions will be investigated in the experiments (in Section 5).

Average Case Time Complexity: $O(w) + O(ws)$

The average of the *best* case and other cases including the *worst* case is computed approximately from the average of the *best* and *worst* cases, which is $O(w)_{radix\ sort} + O(ws)_{OBS\ merge}$.

Lastly, for the case $P=N$, our *OBS* method is equivalent to the efficient *Bitonic* sort with a constant *key* exchange (to update the list ranking) in $O(s^2)$ but the *dynamic* DCES [9] is $O(P)$, which is not optimal.

5 Experimental Results

In the performance evaluation, we implemented the optimized *OBS* strategy, the *dynamic DCES* method [9] and the *static LBM* sort [7] on multicore machines. Our focus is on the improvement of the parallel sorting part (especially the communication among *PEs*). Therefore, we apply the local *radix* sort (in the initial step) in three sorting methods.

In our environment, the system consists of 8 computing nodes (or cores) with speeds of 3.1 GHz and 8 GB of RAM. Programming code was written in the C language with the MPI communication library. Experiments have been performed by varying the number of cores ($P = 1, 2, 4, 8$) and various problem sizes ($N = 10, 20, 30, \ldots, 90, 100$ million (M) elements). The input data were synthetically generated with three distributions: 1) Uniform distribution for random datasets 2) Left-skew distribution (L-skew) for irregular datasets and 3) Right-skew distribution (R-skew) for other irregular datasets. In each case, a number of sampling datasets were executed and repeated up to a stable state (~ 10000 different datasets). The performance results were computed in terms of the average response time, the average speedup, and the average efficiency, where the speedup $S=T_1/T_P$ and the efficiency $E=S/P$, T_1 is the time of running the sequential program on one core and T_p is the time for running the parallel program on P cores (or *PEs*). In each experiment, the results of the optimized *OBS* have been investigated and compared to the best *static* approach (*LBM* sort [7]) and the *dynamic DCES* method [9].

The average response time (T_p) of the proposed *OBS*, the *DCES*, and the *LBM* were investigated on $P = 1, 2, 4$, and 8 cores (or *PEs*) using various problem sizes ($N = 10^7$ (or 10M) up to 10^8 (or 100M) elements). First, we investigated the effect of problem sizes (N) by varying N on fixed *PEs*. Table 4 shows the experimental results (average response time (in secs.)) obtained by varying the problem sizes ($N = 10^7 – 10^8$) across $P = 8$ *PEs*. Clearly, the response time of data sorting by one *PE* did not return in a reasonable time, where the larger sizes of input data (N), the longer response time (T_1) such as 10M needed 1.37 secs., 100M needed 15.44 secs. and so on. Using more *PEs* to process the problems in parallel can reduce the response time of sorting. Among the three parallel sorting approaches (optimized *OBS*, *dynamic DCES*, and *static LBM* sort, our *OBS* strategy yielded the *fastest* sorting time for all problem sizes ($N = 10^7 – 10^8$ elements). Because of the effect of *dynamic* communication among *PEs* (between 1 and s iterations), the results of our *OBS* method outperformed those of existing approaches, where the *static LBM* sort was processed by using fixed ($s(s+1)/2$) iterations and the *dynamic DCES* [9] executed between 1 and $s(s+1)/2$) iterations.

Fig.22 presents the (average) response time of three sorting methods for the problem sizes $N = 10M - 100M$ (on $P=8$ *PEs*) under the left-skew distribution. Our new *OBS* method returned the *minimum* response time in all tests, which improved over those of the *DCES* method 35% - 40% and improved over those of the *LBM* sort by approximately 50% - 53%. Note that for other distributions (i.e., random datasets, R-skew datasets), the performance results are also improved similar to those of the left-skew distribution.

The next issue to investigate was the effect of increasing processors (P). We set the problem size to $N = 100$ million (M) elements with three distributions (random, left-skew (L), right-skew (R)) and varied $P = 2, 4, 8$ *PEs*. Table 5 illustrates the response time (T_P) of those three sorting approaches (*OBS*, *DCES*, and *LBM*), where the more utilized processors, the faster response time.

Using $P=2$ *PEs* (or cores), the optimized *OBS* method improved by 13% over the *LBM* sort and improved by 9% over the *dynamic DCES* method since both *dynamic* and *static* communication finished sorting in one iteration. Using $P=4$, our optimized *OBS* method improved the response time by 7% over the *dynamic DCES* and improved by 35% over the *static LBM* since the *dynamic* communication can finish sorting in 1 or 2 iterations, while the *static* communication needed 3 iterations to complete.

Table 4. Response time (T_P) of three sorting methods with P=8.

Problem sizes (N)	Input Data	T_1 (secs.)	LBM [7] (static)	DCES [9] (dynamic)	**OBS (dynamic)**
	Random		0.37	0.27	**0.18**
$10x10^6$	L-Skew	1.37	0.39	0.28	**0.18**
	R-Skew		0.40	0.29	**0.18**
	Random		1.03	0.79	**0.54**
$40\ x10^6$	L-Skew	4.08	1.09	0.85	**0.51**
	R-Skew		1.12	0.86	**0.52**
	Random		2.44	1.74	**1.28**
$70\ x10^6$	L-Skew	9.42	2.48	1.83	**1.19**
	R-Skew		2.50	1.84	**1.20**
	Random		3.92	2.75	**2.10**
$100\ x10^6$	L-Skew	15.44	3.96	2.99	**1.95**
	R-Skew		3.98	3.04	**1.96**

Table 5. Response Time (T_P) of Three Sorting Methods with N=100M.

PEs (P)	Input Data	LBM [7] (static)	DCES [9] (dynamic)	**OBS (dynamic)**
	Random	9.22	8.79	**7.97**
2	L-Skew	9.36	8.85	**8.11**
	R-Skew	9.43	8.91	**8.15**
	Random	6.24	4.32	**4.04**
4	L-Skew	6.29	4.38	**4.08**
	R-Skew	6.31	4.42	**4.12**
	Random	3.92	2.75	**2.10**
8	L-Skew	3.96	2.99	**1.95**
	R-Skew	3.98	3.04	**1.96**

Finally, using *P=8*, our optimized *OBS* strategy improved the response time by 24% (random), 35% (L-skew and R-skew) over the *dynamic DCES* and improved up to 46% (random), 51% (L-skew and R-skew) over the *static LBM*, according to the *dynamic* (1 - 3) iterations whereas the *static* approach required 6 iterations.

Clearly, when sorting with more *PEs*, our *OBS* can handle the better synchronization (for data exchange) in each iteration, and hence require fewer iterations than those of the *dynamic DCES* due to the *midpoint*-weight list ranking, especially for the skewed datasets.

Next, in the evaluation of speedup and efficiency when increasing *P* cores or *PEs*, the performance results of three sorting methods (optimized *OBS*, *dynamic DCES*, and *static LBM*) were compared in Fig.23 and 24 on the datasets, obtained from the left-skew distribution.

Fig.23 shows the (average) speedup of sorting (N = 100M), which our *OBS* method yielded the *maximum* speedup in all the tests conducted. In particular, when using $P = 8$ *PEs*, the speedup of our *OBS* is 7.92, while those of the *DCES* and *LBM* are 5.16 and 3.90, respectively. When increasing the number of *PEs* (P = 2, 4, 8), speedup (S_P) of our *OBS* method increase (S_P = 1.90, 3.78, and 7.92), which are close to the ideal case (P = 2, 4, and 8), while those of the *dynamic DCES* and *static LBM* also increase, but the lower rate because of using *1* to $s(s+1)/2$ iterations (in the *dynamic DCES* communication) and fixed $s(s+1)/2$ iterations in the *static LBM* communication.

Fig.24 displays the (average) efficiency of sorting (N = 100M), which our *OBS* strategy provided the *maximum* efficiency in all test cases (on P = 2 to 32 processors). The efficiency of our *OBS* method are close to the ideal value = 1 (i.e., 0.95 (on P = 4), 0.99 (on P = 8), 0.97 (on P = 16), and 0.98 (on P = 32), while those of the *DCES* method are 0.88 (on P = 4) and 0.65 (on P = 8) and those of the *LBM* sort are 0.61 (on P = 4) and 0.49 (on P = 8), and so on.

Finally, in order to confirm the effect of *dynamic* iterations in our optimized *OBS* strategy and the *dynamic DCES* method, we collected a number of *dynamic* iterations during the experiments by counting repeated frequency of best, worse, and average cases in Table 6.

TABLE 6. FREQUENCY OF DYNAMIC ITERATIONS (FROM 10,000 SAMPLES) OF TWO DYNAMIC SORTING METHODS WITH P=8.

Problem sizes (N)	Input data (iteration)	BEST DCES (1)	BEST **OBS** (1)	Others DCES (2-3)	Others **OBS** (2)	WORST DCES (4-6)	WORST **OBS** (3)
	Random	453	**847**	2196	**3604**	6851	**6049**
$10x10^6$	L-Skew	363	**1037**	1391	**2709**	7846	**6654**
	R-Skew	273	**1227**	953	**2347**	8174	**7026**
	Random	468	**832**	1733	**3467**	6899	**6601**
$40 x10^6$	L-Skew	262	**838**	1527	**3473**	7211	**6689**
	R-Skew	140	**560**	1362	**3538**	7498	**6902**
	Random	259	**641**	2319	**3481**	7422	**5878**
$70 x10^6$	L-Skew	315	**985**	1375	**2925**	7110	**7290**
	R-Skew	302	**898**	1629	**3171**	7369	**6631**
	Random	174	**326**	1962	**3238**	7314	**6986**
$100 x10^6$	L-Skew	121	**479**	1755	**3445**	7474	**6726**
	R-Skew	267	**533**	1674	**3726**	7159	**6641**

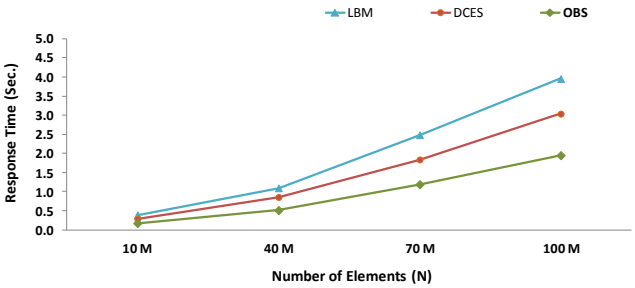


Fig.22. Response time of sorting with Left-skew distribution (P=8).

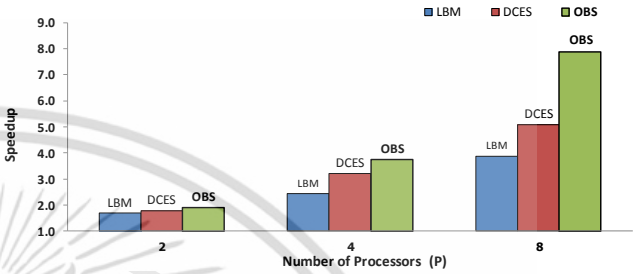


Fig.23. Speedup of sorting with Left-skew distribution (N=100M).

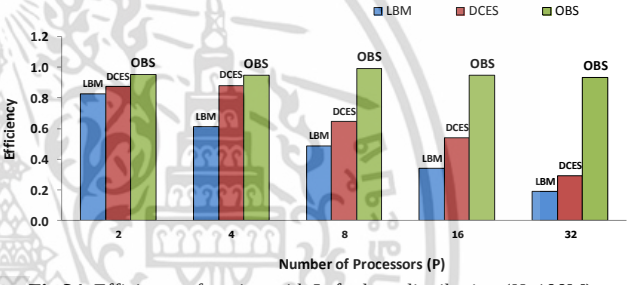


Fig.24. Efficiency of sorting with Left-skew distribution (N=100M).

For example, on P = 8 *PEs*, our *OBS* strategy can sort in 1 iteration (in the *best* case) and 3 iterations (in the *worst* case), while the *dynamic DCES* method also can sort in 1 iteration (in the *best* case), but 6 iterations (in the *worst* case) and between 2 to 5 iterations (in other cases). In each iteration, the data exchange is at most $w/2$ elements for both *dynamic* sorting methods.

In Table 6 (for N = 10M), the number of iterations in each case (*best*/*worst*/*others*) were accumulated from executing ~10,000 different datasets. For the random data, our *OBS* strategy finished sorting in 1 iteration (*best* case) ~8% (847 datasets), 3 iterations (*worst* case) ~60% (6049 datasets), and other cases with 2 iterations ~36% (3604 datasets). The *DCES* method finished sorting in 1 iteration (*best* case) ~4%, 4-6 iterations (*worst* case) ~68%, and others (in 2-3 iterations) ~22%. For irregular data (Left-skewed distribution), our *OBS* strategy finished sorting in *best* case (1 iteration) ~10%, *worst* case (3 iterations) ~66%, and others (2 iterations) 27%. The *DCES* method finished sorting in *best* case (1 iteration) ~3%, *worst* case (4-6 iterations) ~78%, and others (2-3 iterations) ~14%. Note that the number of *static* iterations in the *LBM* sort are fixed in 6 iterations on P = 8 *PEs* ($\leq w=N/P$ data exchange per iterations).

6 Conclusion

In this paper, the optimized *Bitonic* sort (*OBS*) for $P < N$ is presented which incorporates the *midpoint*-based *dynamic* communication. We use the *midpoint*-weight list-ranking and the 2-step parallel point-to-point communication to handle the *dynamic* communication efficiently. The *dynamic* iterations of our proposed *OBS* method are varied between 1 and s, while those of the *dynamic DCES* method can occur between 1 and $s(s+1)/2$ and those of the *static LBM* are fixed in $s(s+1)/2$ iterations. The response time of our new *OBS* method was improved according to the optimized communication among *PEs* with fewer *dynamic* iterations and better synchronization with consistency in data exchange between (P_i, P_j). In experiments on multi-core machines, the results of the proposed *OBS* strategy outperformed those of the *dynamic DCES* method by approximately 35% – 40% and those of the *LBM* sort (the best of *static* approach) by approximately 51% – 54% (for N = 10M – 100M elements on an 8-multicore machine).

7 References

[1] K. Batcher, "Sorting networks and their applications," *In proceedings of the AFIPS Spring Joint Computer Conference 32*, Reston, Virginia, 1968.

[2] J. D. Lee and K. Batcher, "Minimizing communication in the bitonic sort," *IEEE Transactions on Parallel and Distributed Systems*, v.11(5), 459-473, 2000.

[3] D. Bitton, D. DeWitt., D. K. Hsiao, and J. Menon, "A taxonomy of parallel sorting," *ACM Computing Surveys*, v.16(3), 287-318, 1984.

[4] R. Cole, "Parallel merge sort," *SIAM Journal of Computing*, v.17(4), 770-785, 1998.

[5] J. Seiferas, "Networks for sorting multitonic sequences," *Journal of Parallel and Distributed Computing*, v.65(12), 1601-1606, 2005.

[6] Y. Kim., M. Jeon,, D. Kim, and A. Sohn, "Communication-efficient bitonic sort on a distributed memory parallel computer," *In proceedings of International Conference on Parallel and Distributed Systems*, pp.165-170, Kyongju City, Korea, June 2001.

[7] M. Jeon and D. Kim, "Parallel merge sort with load balancing," *Journal of Parallel Programming*, v.31(1), 21-33, 2003.

[8] T. Thanakulwarapas and J. Werapun, "Communication -space efficient parallel bitonic sorting on symmetric multiprocessors," *In proceedings of Advances in Computer Science and Technology*, pp. 75-79, Langkawi, Malaysia, April 2008.

[9] T. Thanakulwarapas and J. Werapun, "Dynamic Communication-Efficient Parallel Sorting on SMPs," *In proceedings of the 11th IEEE International Conference on High Performance Computing and Communications*, pp. 132-138, Seoul, Korea, June 2009.

[10] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: portable parallel programming with the massage passing interface, The MIT Press, Cambridge, 1994.

[11] D. Jim´enez-Gonz´alez, J. Navarro, and J.-L. Larriba-Pey, "The Effect of local sort on parallel sorting algorithms," *In proceedings of Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pp. 360-367, Canary Islands, Spain, January 2002.

[12] El-Nashar A.I. "Parallel Performance of MPI Sorting Algorithms on Dual-Core Processor Windows-Based Systems." *International Journal of Distributed and Parallel Systems(IJDPS)*, v.2(3), 1-14, 2011.

Tipraporn Thanakulwarapas received the B.S. degree in Computer Science in 2003 from Mahidol University, Bangkok, Thailand and the M.S. degree in Computer Science in 2005 from King Mongkut's Institute of Technology, Ladkrabang(KMITL), Bangkok, Thailand. Since 2005, she has been a researcher at National Electronics and Computer Technology Center (NECTEC), Bangkok, Thailand. Miss Thanakulwarapas is a Ph.D. candidate (in Computer Science) at KMITL, Bangkok, Thailand. Her research interests are parallel sorting, parallel algorithms, and parallel computing.

Jeeraporn Werapun received the B.S. degree from Kasetsart University, Bangkok, Thailand, the M.S. degree from Chulalongkorn University, Bangkok, Thailand, the M.S. degree in Computer Science in 1993, and the D.Sc. degree in Computer Engineering (Parallel and Distributed Systems) in 1999 from the George Washington University, Washington, D.C., U.S.A. Dr. Werapun is an Associate Professor of Computer Science at King Mongkut's Institute of Technology, Ladkrabang (KMITL), Bangkok, Thailand. Her research interests include parallel algorithms, parallel and distributed computing, and parallel computer architectures.

Dynamic Communication-Efficient Parallel Sorting on SMPs

T. Thanakulwarapas[1] and J. Werapun[2]
Department of Computer Science, Faculty of Science,
King Mongkut's Institute of Technology at Ladkrabang (KMITL)
Ladkrabang, Bangkok 10520, THAILAND
[1] onenoi@hotmail.com [1], S0067102@kmitl.ac.th, [2] ksjeerap@kmitl.ac.th

Abstract

This paper proposes a dynamic communication-efficient pattern and introduces a new efficient data-exchange process in parallel sorting, called the DCES (Dynamic Communication-Efficient parallel Sorting) algorithm, to improve the communication time. In this approach, we present the dynamic communication pattern by using a "Broadcast-Checker" table, which can reduce total iterations to one iteration (in best case), or 2, 3, …, or up to $log_2P(log_2P+1)/2$ (in worst case), while total (static) iterations of the recently study are fixed = $log_2P(log_2P+1)/2$. Finally to evaluate the sorting performance, we implemented our DCES algorithm on the SGI Origin2000. Our investigated experimental results have been compared to those of the best of existing algorithms (LBM: Load Balanced Merge sort). In the experiments, the proposed DCES algorithm yielded the improved results over those of the LBM algorithm at least 24% on the system of size P = 4 and at least 34% on the system of size P = 8.

1. Introduction

Sorting is one of common, yet significant, operations for many applications such as information retrieval, which requires a sorted data list to operate efficiently. In addition, those operations can be used to solve problems in graph theory, computation geometry, and image processing in optimal time. In the past, several parallel sorting approaches were proposed such as Bitonic sort [1, 9], parallel merge sort [2, 3], parallel radix sort [2], and network sort [4, 10], to reduce total execution time. Usually, computation time decreases when processors (P) increases in fine-grained sorting ($P = N$). However, in coarse-grained sorting ($P < N$), the great impact on the total execution time are balancing workload (N/P) in each processor as well as inter-processor communication for data exchange.

Recently studies have been focused on the later case ($P < N$) by improving the total computing time, especially the communication time for data exchange. In 2001, the communication efficient Bitonic Sort (CEBS) with local quick sort [8] was introduced. The CEBS algorithm showed the improved results over those of the original Bitonic Sort with local quick sort at least 20%. In 2003, Jeon and Kim [7] introduced a new parallel merge-sorting algorithm with balancing workload, called "Load-Balanced Merge" (LBM). That approach applied quick sort in local sorting of N/P keys (in each processor) and parallel merge sort with median computing to improve merging time. In 2008, the communication-space efficient Bitonic sorting (CSEBS) method [11], was introduced with the new efficient partial data exchange between (P_i, P_j). In that data exchange, either P_i or P_j, identifying less partial data to be exchanged, was a sender. Thus, that approach reduced not only the communication time but also the memory storage over those of the Odd-Even sort [2], and the Bitonic sort [8].

In this paper, we propose a new efficient data exchange method in parallel sorting in order to improve the communication time. In particular, we introduce a "dynamic" communication pattern to reduce total iterations (1, 2, 3, …, or $log_2P(log_2P+1)/2$) over those ($log_2P(log_2P+1)/2$) of the "static" communication pattern in the existing methods [7, 8, 11]. In addition, for each iteration (if any) we introduce the "efficient median computing" by modifying the "median computing" strategy [7] to reduce the amount of data exchange between processors (P_i, P_j). Our new sorting is called the DCES (Dynamic Communication-Efficient parallel Sorting) algorithm. Finally in order to evaluate the performance, we implemented our new parallel sorting approach on the SGI Origin2000, an SMP (Symmetric Multiprocessor) [6], by using MPI [5]. In the performance evaluation, results of our DCES algorithm were compared to those of the LBM strategy [7], the CEBS strategy [8], and the CSEBS strategy

978-0-7695-3738-2/09 $25.00 © 2009 IEEE
DOI 10.1109/HPCC.2009.50

[11]. The experimental results showed that our DCES algorithm yielded the improved results over those of the CSEBS, LBM, and CEBS strategies at least 23%, 24% and 31% on the system with $P = 4$ processors and at least 26%, 34% and 49% on the system with $P = 8$ processors respectively.

The remainder of this paper is organized as follows: Section 2 presents related work, which are efficient parallel sort ($P = N$) and communication-efficient parallel sort ($P < N$). Section 3 introduces our DCES (Dynamic Communication-Efficient parallel Sorting) algorithm. Section 4 shows time complexity analysis of our DCES algorithm. Section 5 presents the investigated results in the performance evaluation. Finally, Section 6 presents the conclusion of our study.

2. Related Work

Sorting is often applied in many applications for ordering a large quantity of data (N). Time complexity of the sequential sorting is $O(Nlog_2N)$. When parallelizing, time complexity of the efficient sorting algorithm is $O(log_2N)^2$, where N is a number of elements in the list and P is a number of processors in the system ($P = N$). For $P = N$, the efficient sorting is Bitonic sorting [1]. For $P < N$, most studies focus on communication-efficient sorting with N/P balancing workload [7, 8, 11].

A. Bitonic Sorting

For $P = N$, the Bitonic sorting is an efficient parallel sorting algorithm since its time complexity is only $O(log_2N)$ for a Bitonic sequence and $O(log_2N)^2$ for any sequence.

Definition 1: A Bitonic sequence is a sequence of a data set $(a_0, a_1, a_2, ..., a_i, ..., a_{N-1})$, where $a_0, a_1, a_2, ..., a_i$ is a monotonic increasing sub-sequence and $a_{i+1}, a_{i+2}, a_{i+3}, ..., a_{N-1}$ is a monotonic decreasing sub-sequence ($0 \le i \le N-1$). An instance of a Bitonic sequence (N=16) is 4, 5, 6, 8,10,15,20,30 28,25,20,15,12, 9, 3, 1.

The Bitonic sorting for N elements (of any sequence) on P processors consists of two steps (Figure 1): 1) Transformation (of any sequence into a Bitonic sequence) and 2) Bitonic sorting. When $P = N$, there are log_2N steps. For each step i (i = 1,2,3,…,log_2N), there are i iteration(s), and hence total iterations are $1+2+3+\ldots+log_2N = log_2N(log_2N+1)/2$ iterations.

The inter-process communication in the Bitonic sorting is called a compare-exchange operation, which is compared in parallel between $(a_i, a_{N/2+i})$ in processors $(P_i, P_{N/2+i})$ for min $(a_i, a_{N/2+i})$ and max $(a_i, a_{N/2+i})$, where $i = 0, 1, 2, .. N/2 - 1$.

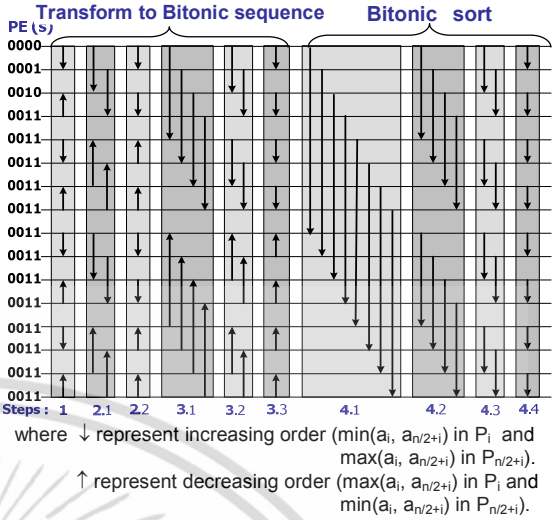


Figure 1. Communication pattern in Bitonic sorting.

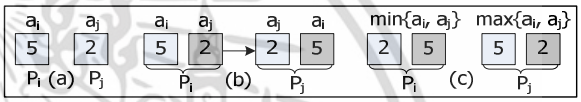


Figure 2. Compare-Exchange Operation.

Figure 2 shows an example of the compare-exchange operation between any two processors (P_i, P_j). Initially (Figure 2(a)), each processor (P_i) has one element (a_i) in its local memory. Next (Figure 2(b)), each processor in the communication pair (P_i and P_j) sends its element to and receives another element from its partner. Finally (Figure 2(c)), after the compare-exchange process (↓), P_i stores min(a_i, a_j) and P_j stores max(a_i, a_j). Note: in case of operation (↑) in the communication pattern, P_i stores max(a_i, a_j) and P_j stores min(a_i, a_j).

B. Balancing Workload Parallel Sorting

In practical usage (when $P<N$), each processor (P_i) holds a local workload (of N/P elements) and performs local sorting. In the inter-process communication between any two processors (P_i, P_j), the compare-exchange operation combined with sub-sequence merge-sort pattern (Figure 3) is called the compare-split operation, as shown in Figure 4.

Figure 4 illustrates an example of merge sorting ($P < N$) between P_i and P_j with workload N/P=5 elements in each PE, where P_i and P_j hold {6,1,11,15,8} and {16,20,13,10,12} (Figure 4(a)). Each processor is responsible for sorting 5 elements locally (Figure 4(b)). Then (Figure 4(c)), processors P_i and P_j need to communicate for data exchange and perform merge sort (Figure 4(d)). In particular, each processor in the communication pair (P_i and P_j) sends its workload to and receives another workload from its partner.

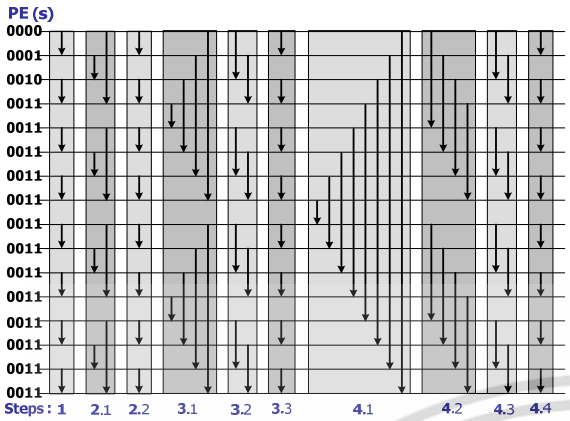


Figure 3. Communication pattern in merge sorting.

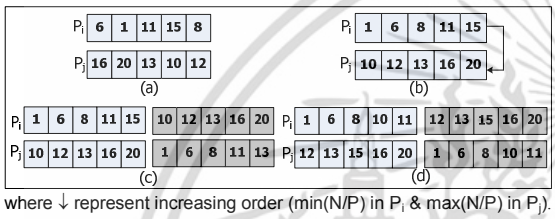


Figure 4. Compare-Split Operation.

In Figure 4, after performing communication merge sort, P_i stores the minimum sub-sequence and P_j stores the maximum sub-sequence.

For $P < N$, that parallel sorting is performed with N/P-workload in $O(N/P(log_2N/P) + N/P(log_2P)^2)$ time since the local quick-sort requires $O(N/P(log_2N/P)$ and the parallel merge-sort requires $O(N/P(log_2P)^2)$ by using the communication pattern with log_2P steps or $log_2P(log_2P+1)/2$ iterations for applying either Bitonic sort (Figure 1) or parallel merge-sort (Figure 3).

However, the communication time of this simple merge sort is not efficient because in some cases processors P_i and P_j do not need to exchange their data. In 2001, the communication-efficient Bitonic Sort (CEBS) [8] was introduced to improve the communication time ($O(N/P(log_2P)^2)$) by reducing the amount of data exchange ($\leq N/P$) in each iteration. In that study, there are three sub-patterns of the compare-split operation: 1) "hold" pattern (no data sending/ receiving) in best case, 2) "swap" pattern (send/receive all (N/P) data) in worst case, and 3) "partial" pattern (send/receive partial ($<N/P$) data). In 2003, the Load-Balanced Merge (LBM) approach [7] was introduced to improve the execute time by computing "median" (to reduce the merging time of the partial sorted ($<N/P$) data between P_i and P_j in the "partial" pattern.) Later in 2008, "the communication-space efficient Bitonic sorting (CSEBS) method" [11], our previous study, was introduced. The CSEBS method also consists of three sub-patterns for the merge-and-split operation: 1) "hold" pattern, 2) "swap" pattern, and 3) new efficient "partial" pattern. That approach presented the more efficient partial data exchange between (P_i, P_j). The idea is that either P_i or P_j, identifying less partial data to be exchanged, was a sender. Thus, that approach reduced not only the communication time (~15%) but also the memory storage over those of the Odd-Even sort [2], and the Bitonic sort [8].

However, in those existing studies [7, 8, 11], the communication pattern is "static" because total iterations are always equal to $log_2P(log_2P+1)/2$.

3. Our Efficient Parallel Sorting

Our efficient sorting, called "Dynamic Communication-Efficient parallel Sorting (DCES) algorithm", consists of two main parts: 1) "Local sort" for N/P-key workload and "Parallel merge-sort" with P processors. In the 1st part (local sorting), we apply the quick sort in $O(N/P\ log_2N/P)$ time. For the 2nd part (parallel merge-sort), we propose the efficient parallel merge-sort algorithm to reduce total iterations ($log_2P(log_2P+1)/2$) to one iteration (in best case) and ($[log_2P(log_2P+1)/2]+1)/2$ iterations (in average case) by utilizing a "Broadcast-Checker" table for identifying the "dynamic" communication pattern. In addition, in each iteration we also improve the communication time in the "partial" pattern by efficiently computing "median" to reduce the amount of data exchange ($<(N/P)/2$). Moreover in our "dynamic" approach, there is no "swap" pattern since the "swap" pattern (the worst case in [7, 8,11]) can be converted to the "hold" pattern (the best case in our efficient DCES algorithm).

A. The "Broadcast-Checker" Table

In traditional communication-merge pattern (Figure 3) of multiple sorted lists, any P_i stores a smaller set of data and P_j stores a larger set of data for processor's ID i < j. For instance, P_0 stores a minimum set of data and P_{P-1} stores a maximum set of data. That common parallel merge-sort is the "array-based IDs", which consists of log_2P steps. In each step i, the merging process requires i iterations to meet all IDs in the group (of 2^i PEs), where $i = 1, 2, \ldots, log_2P$. Therefore, total (fixed) iterations to complete the sorting are $log_2P(log_2P+1)/2$.

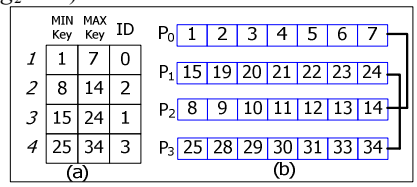


Figure 5. A "Broadcast-Checker" table for $P = 4$.

Our "dynamic" communication-merge pattern is the "linked-list-based IDs" and requires only one iteration (in best case) and ($[log_2P(log_2P+1)/2]+1)/2$ iterations (in average case). In the list of IDs, P_0 is not necessary to store the minimum set of data and hence P_{P-1} is not necessary to store the maximum set of data. In order to maintain useful information in each processor (P_i), the "Broadcast-Checker" table is introduced. In such a table, there are P records, each record contains 3 fields (min key, max key, processor's ID). The list of IDs is linked, according to the sorted "min key" of P records. The sorted list of P records is updated by using an efficient merged-sort among P PEs, representing "all-to-all broadcast and sort", in log_2P steps ($P=2^k$). In each step i, all (P_i, P_j)s exchange 2^i sorted records, merge and store 2^{i+1} records in each PE, where $j = i+2^i$ and $i = 0,1,2,\ldots, k\text{-}1$. For k steps, total merges are $2.2^0 + 2.2^1 + 2.2^2 + \ldots + 2.2^{k-1} = 2(2^{k-1+1}\text{-}1)/(2\text{-}1) = 2.2^k\text{–}1 =$ O(P).

For example, to sort a data set (N=28 keys) {1, 7, 5, 4, 3, 2, 6, 20, 19, 15, 23, 21, 24, 22, 9, 13, 12, 10, 8, 14, 11, 29, 28, 25, 34, 31, 33, 30} by using P = 4 processors, the 4-record table is depicted in Figure 5(a). After performing the local sorting, the result in each processor (P_i, i = 0,1,2,3) is shown in Figure 5(b). Form this 4-record table, the adjacent processors (P_i, P_j) in the list (or the 3rd field) are communicated (e.g., (P_0,P_2) and (P_1,P_3)). This example is one of the best cases in our "dynamic" communication pattern since only one iteration is needed to complete the sorting, stored in the list ($P_0 \rightarrow P_2 \rightarrow P_1 \rightarrow P_3$). Note: a number of iterations in the traditional communication [7, 8, 11] for this example is $log_24(log_24+1)/2 = 3$ iterations.

In the "static" communication pattern of parallel merge-sort ($P < N$), the number of iterations are fixed ($log_2P(log_2P+1)/2$) such as those of the Bitonic sorting (Figure 1) and the Merge sorting (Figure 3). In our "dynamic" communication pattern, the number of iterations can vary from 1, 2, 3, 4, …, up to ($log_2P(log_2P+1)$)/2 iterations, as shown in Table 1.

Table 1. The number of iteration(s) in communication pattern with the "static" vs. "dynamic" Merge sort.

Methods	"Static" Merge Sort	"Dynamic" Merge Sort
Best Case	$log_2P(log_2P+1)/2$	1
Average Case	$log_2P(log_2P+1)/2$	$([log_2P(log_2P+1)/2]+1)/2$
Worst Case	$log_2P(log_2P+1)/2$	$log_2P(log_2P+1)/2$

Moreover, in each iteration the communication time of our DCES algorithm can be improved in both "swap" pattern and "partial" pattern. Because of the "dynamic: communication of processors' IDs in the list, consequently, there is no "swap" pattern in our approach. In this case, it is automatically converted to the "hold" pattern in the list with 100% improved.

B. The Efficient "Median" Computing

In recently study [7], the "median" computing was introduced in the "partial" pattern to improve the merging time of partial data in P_i and P_j over that of their previous study [8]. However, the amount of partial data exchange ($<N/P$) was not improved since the partial data of P_i and P_j are exchanged before computing the "median".

In our DCES algorithm, the efficient "median" computing is introduced to reduce not only the amount of data exchange in the "partial" pattern but also the merging time. In this case, only the minimum partial data ($\leq(N/P)/2$) are exchanged between P_i and P_j. First, each processor (P_i) computes and sends the local median (Med) to its partner (P_j). Then, the "median" is computed from the local median (Med) of P_i and P_j directly. Finally, the minimum partial data set ($\leq(N/P)/2$) in each processor (P_i) is identified from the "median" and sent to be merged in its partner (P_j).

C. The DCES Algorithm

The Dynamic Communication-Efficient parallel Sorting (DCES) algorithm is

1. Each processor (PE) makes local quick-sort on a list of N/P elements. *O(N/P log₂N/P)*
2. For (at most) log_2P steps (if any), each step i needs i iterations (i = 1,2,…,log_2P), each PE performs the following computation:
   (Note: one iteration (in best case) and $log_2P(log_2P+1)/2$ iterations (in worst case).)
   2.1 Send/Receive (*Max, Min*) to/from its partner. *O(1)*
   2.2 Find a *partial data* set from (*Max, Min*) by applying "Binary search" algorithm. *O(log₂N/P)*
   2.3 Find *local median (Med)* of the partial data set (n elements) and send to its partner (n<N/P).
   (Note: For more precision, |*Med*| is 1 element (if n is odd) or 2 elements (if n is even) are exchanged.) *O(1)*
   2.4 Compute the "*median*" from ≤ 4 elements of its local *Med* and its partner *Med*. *O(1)*
   2.5 Compute "*minimum partial data exchange*". *O(N/P)*
   - Find and exchange *minimum partial data* (Δ) from the "*median*" with its partner ($\Delta \leq (N/P)/2$).
   - Merge results (Δ) received from its partner.
   (Note: if |*partial data*| > (N/P)/2), then the *complement* set of that *partial data* set are exchanged to maintain the minimum $\Delta \leq n$)
   2.6 Update "*Broadcast-Checker*" table (T) by using "efficient P-merged sort" for next iteration. *O(P)*

For example, to sort a data set of $N = 28$ using $P = 4$, the "dynamic" communication and the efficient "median" computing in our DCES algorithm is explained in Figure 6. In Figure 6(a), initially each P_i, i = 0, 1, 2, 3 holds 7 keys ($N/P = 7$), which are

{8, 1, 7, 9,13,12,10}, {20,19,15,23,21,25,29},
{5, 4, 3, 2, 6,14,11}, {28,24,22,34,31,33,30}.

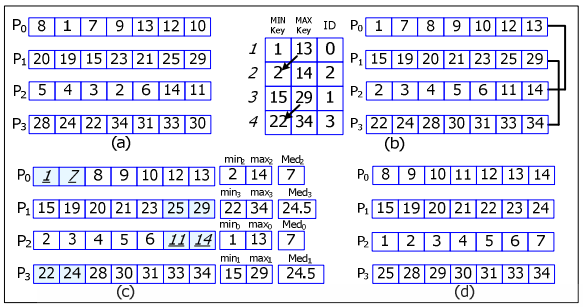


Figure 6. An example of the efficient "median" computing in the DCES algorithm.

Figure 6(b) shows the "Broadcast-Checker" table with the list of IDs ($P_0 \rightarrow P_2 \rightarrow P_1 \rightarrow P_3$) and results of local sorting in each PE. Before performing the "minimum partial data exchange" between {P_0, P_2} and {P_1, P_3} (Figure 6(c)), P_0 sends (Max_0, Min_0) = (1, 13) to P_2 while P_2 sends (Max_2, Min_2) = (2, 14) to P_0 and P_1 sends (Max_1, Min_1) = (15, 29) to P_3 while P_3 sends (Max_3, Min_3) = (22, 34) to P_1.

Next, the partial data set is identified in each PE by applying the "Binary search" in O(log_2N/P) time and then the local median (Med) of the partial data set (n) is computed in O(1) time. In this case, P_0 finds its partial data (n=6): {7,8,9,10,12,13} and sends its Med = (9,10) to P_2, while P_2 finds its partial data (n=6):{2,3,4,5,6,11} and sends back its Med = (4,5) to P_0. P_1 finds its partial data (n=3):{23,25,29} and sends Med = (25) to P_3, while P_3 finds its partial data (n=3):{22,24,28} and sends back its Med = (24) to P_1. Each P_i and P_j computes the "median" from the Med of P_i and P_j and then finds the minimum partial data exchange ($\Delta \leq (N/P)/2$), to be merged in P_i and P_j. Finally, each processor performs merging the minimum partial data Δ (Figure 6(d)).

Note: the complement set of partial data set to be exchanged (in step 2.5) is allowed in our approach (Figure 6(d) between P_0 and P_2) to maintain the minimum data exchange (Δ). In this case P_0 sends the complement set of partial data set or the minimum partial data Δ = {1, 7} to P_2, while P_2 sends the complement set of partial data set Δ = {11, 14} to P_0. However, P_1 sends the regular minimum partial data set Δ = {25, 29} to P_3, while P_3 sends the regular minimum partial data set n = Δ = {22, 24} to P_1.

4. Performance Analysis

Let N represent a number of elements in the data set.
P represent a number of processors in the system.
T_{seq} represent the computing time of sequential (local) sort of N/P-key workload in each PE.
T_{par} represent total computing time of parallel sort.
T_{comp} represent time of parallel data merging.
T_{comm} represent time of inter-process communication to exchange data (Δ keys), where $\Delta \leq (N/P)/2$, and data (P-record) broadcasting.

In general, the response time (T) of the sorting process using P processors consists of

$T = T_{seq} + T_{par}$, where $T_{par} = T_{comp} + T_{comm}$

When applying the quick sort for local sorting with N/P-workload in each PE,

$T_{seq} = O(N/P \log_2 N/P)$

When apply our efficient parallel sorting, called the DCES algorithm, $T_{par} = T_{comp} + T_{comm}$ is

$T_{par} = O(1) + O(P)$ for one iteration in best case,
$= O((N/P \log^2 P) + O((P \log^2 P) = O(N/P \log^2 P)$ for $\leq (\log_2 P (\log_2 P+1))/2$ iterations (with N/P workload) in average and worst cases.

Therefore, $T = T_{seq} + T_{par}$ is

$T = O(N/P \log_2 N/P) + O(P)$ in best case,

or $T = O(N/P \log_2 N/P) + O(N/P \log^2 P)$ in average and worst cases.

Table 1 (in Section 3) illustrates the number of iterations (in the communication pattern) in T_{par} of our "dynamic" approach (DCES) and that of the "static" approach [7, 8, 11].

In our approach, the best case occurs when each PE holds the required (sorted) N/P keys (Figure 5). However, our worst case is similar to that of the "static" approach since all communication pairs ($\log_2 P(\log_2 P+1)/2$) are communicated to exchange their partial data. In our "dynamic" approach, the average iterations are derived from the best case (1 iteration) up to the worst case ($\log_2 P(\log_2 P+1)/2$ iterations), as shown in the following equation:

$[1+2+3+\ldots+\log_2 P(\log_2 P+1)/2] / \log_2 P(\log_2 P+1)/2$
$= ([\log_2 P(\log_2 P+1)/2]+1) / 2$

Compared to the LBM approach [7], the communication of processors in the group is organized to meet all possible pairs and hence the number of iterations are fixed, which are $\log_2 P(\log_2 P+1)/2$ iterations in all (best, average, worst) cases.

Moreover, in each iteration we modify the "median" computing of the LBM strategy to find the minimum data exchange and hence reduce the merging time. In addition, there is no "swap" pattern (the worst case in [7, 8, 11]) in our DCES since it is automatically converted to the "hold" pattern (the best case in our study), according to the "dynamic" communication pattern, as illustrated in Table 2.

Table 2. The amount of data exchange in each iteration of the existing strategies [7,11] and our DCES strategy.

Methods	"Hold" Pattern (Best Case)	"Swap" Pattern (Worst Case)	"Partial" Pattern (General Case)
LBM [7]	No data exchange	N/P	< N/P
CSEBS[11]	No data exchange	N/P	< N/P
Our DCES	No data exchange	No data exchange	≤ (N/P)/2

From Table 1 and 2, clearly time complexity of our DCES algorithm is improved over that of the best (LBM and CSEBS) of existing algorithms in all (best, average, worst) cases.

5. Experimental Results

In performance evaluation, we implemented our DCES (Dynamic Communication-Efficient parallel Sorting), LBM (Load-Balanced Merge), CEBS (Communication-Efficient Bitonic Sorting) and CSEBS (Communication-Space Efficient Bitonic Sorting) algorithms by using MPI library on the SGI Origin2000 system, an SMP (Symmetric Multiprocessor). In our environment, the system consists of 8 processors (MIPS R10000 and R12000) with 2 GB RAM, connected via a high speed 1.56 Gbps processor connection and 780 Mbps memory connection.

A number of experiments were performed by varying a number of processors (P = 1, 2, 4, 8, 16) and various problem sizes (N = 10, 20, 30, 40, …, 100 million elements). In each tested case, a number of sample-data in various iterations (up to stable state) were executed in average. The performance results were computed in terms of average response time, average speedup, and average efficiency, where speedup $S=T_1/T_P$ and efficiency $E=S/P$, T_1 is the time of running sequential program on one processor and T_p is the time of running parallel program on P processors.

In each experiment, evaluated results of our proposed algorithm (DCES: Dynamic Communication-Efficient parallel Sorting) were investigated and compared to those of existing approaches (LBM: Load-Balanced Parallel Merge-sort [7] and CEBS: Communication-Efficient Bitonic Sort [8], and CSEBS: Communication-Space Efficient Bitonic Sort [11]).

Table 3 showed the average response time (T_p) of our DCES strategy and the recent strategies (CSEBS, LBM and CEBS) for various problem sizes $N = 10^7$ up to 10^8 elements and various processors P = 1, 2, 4, 8, 16 processors. When fixing the problem size (N) and increasing a number of processors (P = 2, 4, 8, 16), the average response time of all methods decreased. When fixing a number of processors (P) and increasing the problem sizes (N = 10^7 - 10^8), the average response time of all methods increased.

Table 3. Response time (seconds) of 1, 2, 4, 8, 16 PE(s) and problem sizes 10^7 – 10^8 elements.

N Million	P	T1(s)	Tp(s) CEBS	Tp(s) LBM	Tp(s) CSEBS	Tp(s) DCES
10	2	2.96	1.99	1.76	1.64	1.51
	4		1.38	1.14	1.06	0.83
	8		0.54	0.46	0.42	0.45
	16		0.42	0.40	0.40	0.39
20	2	4.46	2.94	2.34	2.17	2.29
	4		1.50	1.34	1.28	1.24
	8		0.85	0.80	0.72	0.76
	16		0.76	0.62	0.60	0.53
30	2	6.62	4.34	3.87	3.72	3.35
	4		3.02	2.63	2.51	2.14
	8		1.96	1.23	1.03	0.95
	16		1.22	1.03	0.98	0.89
40	2	7.53	5.34	4.68	4.36	3.98
	4		3.59	3.12	2.92	2.74
	8		1.75	1.43	1.27	1.17
	16		1.43	1.21	1.14	0.98
50	2	11.09	6.58	6.29	5.92	5.70
	4		4.89	3.67	3.36	3.19
	8		2.05	1.98	1.85	1.68
	16		1.93	1.78	1.65	1.53
60	2	13.14	7.85	7.26	6.97	6.75
	4		5.32	4.76	4.42	4.18
	8		3.19	2.42	2.06	1.89
	16		2.41	2.13	2.07	1.93
70	2	16.51	9.49	8.90	8.76	8.49
	4		6.29	5.88	5.58	5.41
	8		4.10	3.32	3.07	2.43
	16		2.85	2.54	2.49	2.02
80	2	19.08	10.69	10.25	10.08	9.67
	4		7.16	6.79	6.60	6.23
	8		4.80	4.45	3.99	2.73
	16		3.33	3.12	3.10	3.01
90	2	21.41	11.60	11.24	11.00	10.98
	4		8.21	7.95	7.62	7.37
	8		6.16	4.87	4.12	3.37
	16		4.03	3.76	3.68	3.35
100	2	25.50	13.69	13.21	12.92	12.79
	4		9.87	8.94	8.76	6.73
	8		7.29	5.68	5.05	3.71
	16		4.72	4.08	3.89	2.73

Among all four methods (DCES, LBM, CEBS, CSEBS), our DCES strategy yielded the best result (or minimum response time), which were improved over those of the CSEBS, LBM and CEBS methods at least 23%, 24% and 31% respectively.

When fixing N = 100 million (M) elements and varying P = 2, 4, 8, 16 processors, the response time, speedup, and efficiency of our DCES and existing methods (LBM, CEBS, and CSEBS) were illustrated in Figure 7, 8, and 9 respectively.

Figure 7 presented the average response time of all four methods (DCES, LBM, CEBS, CSEBS) and our DCES performed the best (or minimum) response time, which improved over those of the CSEBS, LBM and CEBS 2%, 3% and 6% when using P = 2, 23%, 24% and 31% when using P = 4, 26%, 34% and 49% when using P = 8, and 30%, 33% and 42% when using P =16 respectively.

Figure 8 showed the average speedup of all four methods (DCES, LBM, CEBS, CSEBS) and our DCES yielded the best (or maximum) speedup in all test cases (P = 2, 4, 8, and 16). In particular, when using P = 2,

the speedup of the DCES is 1.99 closed to 2 (ideal case), while those of the CSEBS, LBM and CEBS methods were 1.97, 1.93 and 1.86 respectively. When increasing a number of processors (P = 4, 8, 16), the speedup of all methods increased but in decreasing rate because the more processors used, the more communication time required.

Figure 9 showed the average efficiency of all four methods (DCES, LBM, CEBS, CSEBS) and our DCES yielded the best (or maximum) efficiency in all test cases (P = 2, 4, 8 and 16). Especially when using P = 2, the speedup of the DCES is 0.99 closed to 1 (ideal case), while those of the CSEBS, LBM and CEBS methods were 0.98, 0.97, 0.93 respectively. When increasing a number of processors (P = 4, 8, 16), the efficiency of all methods decreased because of the more communication among processors.

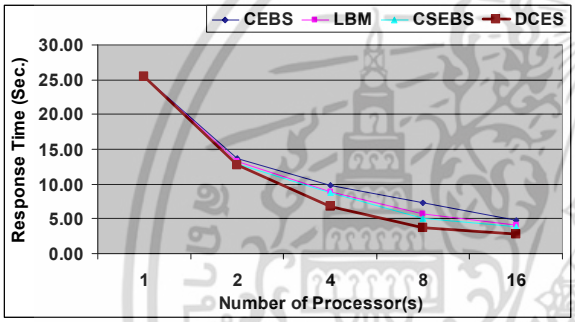


Figure 7. Response time of all methods (N = 100 M).

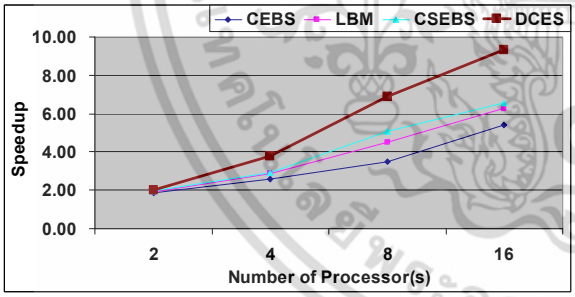


Figure 8. Speedup of all methods (N = 100 M).

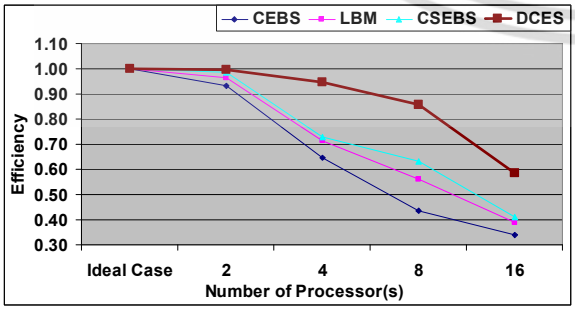


Figure 9. Efficiency of all methods (N = 100 M).

6. Conclusion

In this study, we introduce a "dynamic communication-efficient sorting (DCES) algorithm". In our new approach, the number of iterations are "dynamic" and improved over that "static" in the recently study. In addition, in each iteration the efficient "median" computing is introduced to find the minimum data-exchange and the minimum merging time between P_i and P_j. In the performance evaluation, a number of experiments were performed on the SGI Origin2000. The experimental results showed that in all tested cases, our proposed DCES algorithm yielded the improved results over those of the best of existing algorithms (LBM: Load-Balanced parallel merge- sort) at least 24% (P = 4) and at least 34% (P = 8).

7. References

[1] K. Batcher, "Sorting networks and their applications," *Proceedings of the AFIPS Spring Joint Computer Conference 32*, Reston, VA, 1968, pp.307-314.

[2] D. Bitton, D. DeWitt, D.K. Hsiao, and J. Menon, "A Taxonomy of Parallel Sorting," *in ACM Computing Surveys*, 1984, pp. 287-318.

[3] R. Cole, "Parallel merge sort," *SIAM Journal of Computing*, Vol. 17, No. 4, 1998, pp.770-785.

[4] B. Gianfranco. "Merging and sorting Networks with the Topology of the Omega Network," *IEEE Transactions on Computers*, October 1989, pp.1396-1403.

[5] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the massage passing interface*, Cambridge, MA: MIT Press, 1994.

[6] D. R. Helman, and J. JaJa, "Sorting on Cluster of SMPs," *12th International Parallel Processing Symposium,* University of Maryland, College Park, MD, USA, 1997.

[7] M. Jeon, D. Kim, "Parallel Merge Sort with Load Balancing," *International Journal of Parallel Programming*, Vol. 31, No. 1, 2003, pp.21-33.

[8] Y.Kim, M. Jeon, D. Kim, and A. Sohn, "Communication-Efficient Bitonic Sort on a Distributed Memory Parallel Computer," *International Conference on Parallel and Distributed Systems*, June 2001, pp.165-170.

[9] J. D. Lee, K. E. and Batcher, "Minimizing Communication in the Bitonic Sort," *IEEE Transactions on Parallel and Distributed Systems,* 2000, pp.459-473.

[10] J. Seiferas, "Networks for Sorting Multitonic Sequences," *Journal of Parallel and Distributed Computing*, Vol. 65, 2005, pp.1601-1606.

[11] T. Thanakulwarapas, J. Werapun, "Communication-Space Efficient Parallel Bitonic Sorting on Symmetric Multiprocessors," *Advances in Computer Science and Technology (ACST 2008),* April 2008, pp.75-79.

Proceedings of the Fourth IASTED International Conference
Advances in Computer Science and Technology (ACST 2008)
April 2-4, 2008 Langkawi, Malaysia
ISBN CD: 978-0-88986-730-7

COMMUNICATION-SPACE EFFICIENT PARALLEL BITONIC SORTING ON SYMMETRIC MULTIPROCESSORS

Tipraporn Thanakulwarapas and Jeeraporn Werapun
Department of Mathematics and Computer Science, Faculty of Science, King Mongkut's Institute of Technology at Ladkrabang (KMITL), Ladkrabang, Bangkok 10520, THAILAND
E-mail: S0067102@kmitl.ac.th, ksjeerap@kmitl.ac.th

ABSTRACT
This paper presents a new efficient data-exchange process in parallel sorting, which improves both communication time and memory space. In this communication-space efficient strategy, we proposed two new parallel sorting algorithms: 1) CSEBS (Communication-Space Efficient Bitonic Sort) algorithm and 2) CSEOE (Communication-Space Efficient Odd-Even MergeSort) algorithm. In addition, to evaluate the performance, we have implemented the CSEBS and the CSEOE algorithms on the SGI Origin2000. Investigated experimental results have been compared to those of the best of existing algorithm (CEBS: Communication Efficient Bitonic Sort). In the experiments, the proposed CSEBS algorithm yielded the improved results over those of the best of existing algorithm (CEBS) at least 11% on the system of size P = 4 and at least 31% on the system of size P = 8. Moreover our approach can save memory space up to 50%.

KEY WORDS
Communication-space efficient parallel sorting, Parallel Bitonic sorting, Parallel Odd-Even sorting, SMP (Symmetric Multiprocessor).

1. Introduction

Sorting is one of common, yet significant, operations for many applications such as information retrieval, which requires sorted data lists to operate efficiently. Those operations of data exchange can be used to solve problems in graph theory, computation geometry and image processing in optimal time. Sorting network for odd-even mergesort and bitonic sort [5] has time complexity only $O(\log^2 N)$, where N is a number of data being sorted on P processors (P = N). In the last decade, many studies were proposed in order to improve communication overhead of parallel Bitonic sorting [4], [6], [8] and were adapted to specific parallel machines. There are two main theoretical and practical approaches that propose the efficient-communication parallel Bitonic sorting: 1) one (theoretical) approach (P = N) bases on the shared-memory parallel systems [4], [6] and 2) another (practical) approach (P < N) bases on the distributed-memory parallel systems [3], [8].

In practical (P < N), the communication efficient Bitonic Sort (CEBS) with local quick sort [8] was introduced in 2001. The CEBS algorithm showed the improved results over those of Bitonic Sort with local quick sort at least 20%. In 2007, Herruzo et al.[2] introduced a new parallel sorting algorithm (P < N) called "partition and concurrent merging" (PCM), with local quick sort based on the odd-even mergesort algorithm. The results of the PCM parallel sorting were compared to the Bitonic sort (BS) with local Bitonic sort. However, the PCM algorithm showed the improved results in some cases when using two processors.

In this paper, we propose a new efficient data-exchange method in parallel sorting to improve not only communication time but also memory space. In addition, we apply this new communication-space efficient strategy to Bitonic Sort (BS) algorithm and Odd-Even mergesort (OE) algorithm. Our new parallel sorting algorithms are 1) the CSEBS (Communication-Space Efficient Bitonic Sort) algorithm and 2) the CSEOE (Communication-Space Efficient Odd-Even mergesort) algorithm. Finally in order to evaluate the performance, we have implemented our new approach on the SGI Origin2000, a SMP (Symmetric Multiprocessor) [1], by using MPI [7]. In the performance evaluation, CSEBS and CSEOE algorithms were compared to the best of existing strategy (CEBS: Communication Efficient Bitonic Sort [8]) and the recent study (OE: Odd-Even mergesort or PCM [2]). Our CSEBS algorithm show that its execution time is minimum as well as its speedup is maximum in all test cases. In particular, our CSEBS algorithm yields the improved results over those of the CEBS algorithm and the OE (or PCM) algorithm at least 11% on the system with P = 4 processors and at least 31% on the system with P = 8 processors. Moreover our approach can save more memory space up to 50%.

The remainder of this paper is organized as follows: Section 2 gives fundamental definitions of parallel sorting. Section 3 presents our new communication-space efficient method and two proposed parallel sorting (CSEBS and CSEOE) algorithms. Section 4 shows the investigated results in the system performance evaluation. Finally, Section 5 presents the conclusion of this study.

2. Parallel Sorting and Related Work

The basic idea of the Bitonic merge sorting and the odd-even merge sorting are illustrated in following sub-sections.

2.1 Parallel Bitonic Sorting

In theoretical (when P = N), the Bitonic sorting is an efficient parallel sorting algorithm since its time complexity is only O(log N) for a Bitonic sequence and $O(\log^2 N)$ for any (random) sequence.

Definition 1: A Bitonic sequence is sequence of a data set $(a_0, a_1, a_2, ..., a_i, ..., a_{N-1})$, where $a_0, a_1, ..., a_i$ is a monotonic increasing sub-sequence and $a_{i+1}, a_{i+2}, ..., a_{N-1}$ is a monotonic decreasing sub-sequence $(0 \le i \le N-1)$. An instance of a Bitonic sequence (N=16) is

4 5 6 8 10 15 20 30 28 25 20 15 12 9 3 1.

Definition 2: A Bitonic Sort is a sorting of N elements (of any sequence) on P processors ($O(\log^2 N)$) consists of two steps: 1) Transformation (of any sequence into a Bitonic sequence) and 2) Bitonic sorting (Figure 1). When P = N, there are $\log_2 N$ main stages (or $1 + 2 + 3 + \ldots + \log_2 N = \log_2 N(\log_2 N+1)/2$ sub-stages). For example, the Bitonic sorting for N = P = 16) is shown in Figure 1.

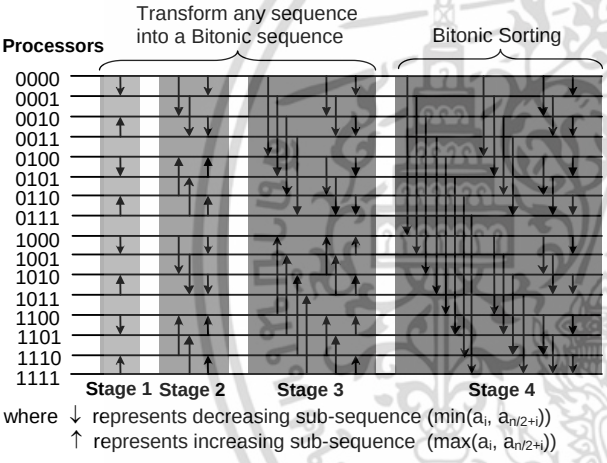


Figure 1. Bitonic sorting with 16 processors.

The inter-process communication in the Bitonic sorting (Stage 4 in Figure 1) is called a compare-exchange operation, which is compared in parallel between $(a_i, a_{N/2+i})$ for min $(a_i, a_{N/2+i})$ and max $(a_i, a_{N/2+i})$; i = 0, 1, ... N/2 – 1. Figure2 shows an example of the compare-exchange operation between any two processors (P_i, P_j). Initially (Figure 2(a)), each processor (P_i) has one element (a_i) in its local memory. Next (Figure 2(b)), each processor in the communication pair (P_i and P_j) sends its element to and receives another element from its partner. Finally (Figure 2(c)), after the compare-exchange process, P_i stores min (a_i, a_j) and P_j stores max (a_i, a_j).

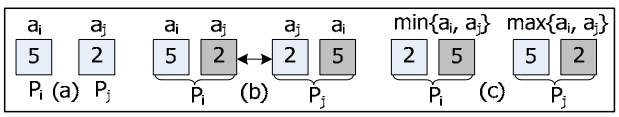


Figure 2. Compare-Exchange Operation.

In practical usage (when P < N), each processor (P_i) holds a local workload (N/P elements) and performs local sorting. In the inter-process communication between any two processors (P_i, P_j), the compare-exchange operation combined with sub-sequence mergesort is called the compare-split operation. Figure 3 illustrates an example of merge sorting between P_i and P_j with workload N/P = 5 elements in each processor, where P_i and P_j hold {6, 1, 11, 15, 8} and {16, 20, 13, 10, 12} respectively.

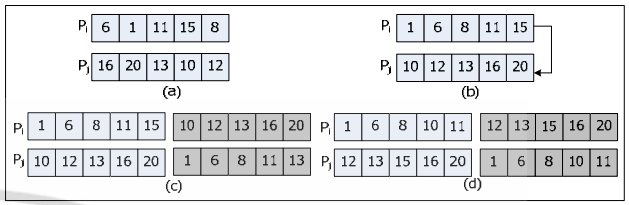


Figure 3. Compare-Split Operation.

Initially (Figure 3(a)), each processor (P_i, P_j) holds 5 elements. Therefore (Figure 3(b)), each processor (P_i) is responsible for sorting 5 elements locally. Then (Figure 3(c)), processors P_i and P_j need to communicate in order to perform Bitonic mergesort for the larger sorted sub-sequences. In particular, each processor in the communication pair (P_i and P_j) sends its workload (data) to and receives another workload from its partner. Finally (Figure 3 (d)), after performing communication Bitonic merge sort, P_i stores the minimum sub-sequence and P_j stores the maximum sub-sequence. However, the communication time of this Bitonic mergesort is not efficient because in some cases processors P_i and P_j do not need to send/receive data. Hence in 2001, the communication-efficient Bitonic Sort [8] was introduced to improve the communication time. In that study, there are three patterns of the compare-split communication: 1) hold pattern (no data sending/receiving), 2) swap pattern (sending/receiving all local data), and 3) partial pattern (sending/receiving partial data).

2.2 Parallel Odd-Even MergeSort

Odd-Even mergesort is another parallel sorting, which is simple for any sequence (of N elements). The network odd-even merge sort for P processors (P = N) consists of N/2-comparator stages. In each stage, there are two parallel (odd/even) steps: 1) Odd index compared neighbors and 2) Even index compared neighbors. Figure 4 shows the communication network of the Odd-Even mergesort with 8 processors (P = N = 8).

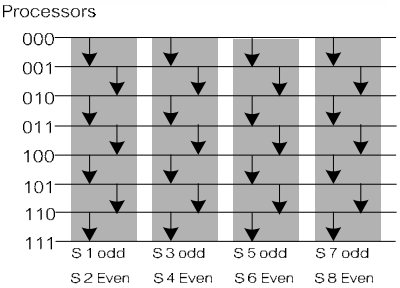


Figure 4. Odd-even merge sorting with 8 processors.

Definition 3: A comparator stage S is a composition of comparators. Sequence a_0, ..., a_{N-1} of length N >1 whose two halves a_0, ..., $a_{N/2-1}$ and $a_{N/2}$, ..., a_{N-1} are sorted. Assume N is a power of 2. Apply odd-even merge (N/2) recursively to the even subsequence a_0, a_2, ..., a_{N-2} and to the odd subsequence a_1, a_3, , ..., a_{N-1}.

In practical (when P < N), each processor (P_i) is responsible for local sorting with a workload (of N/P elements.) In 2007, a new parallel sorting algorithm (P < N) was introduced, based on the odd-even mergesort algorithm, called "partition and concurrent merging" (PCM), with local quick sort [2]. However, the PCM algorithm showed the improved results in some cases when using two processors. Therefore, the efficient communication introduced in [8] can be applied in that PCM algorithm to improve the communication time.

3. New Communication-Space Efficient Parallel Sorting

In this study, we focus on practical sorting (P < N), which consists of two parts of sorting: 1) local sorting (for N/P elements) and parallel merge-sorting (of P processors). In the 1st part (local sorting), we apply the quick sort. For the 2nd part (parallel merge-sorting), we propose the new communication-space efficient (CSE) strategy and apply in Bitonic mergeSort (BS) and Odd-Even mergesort (OE) and hence our new parallel algorithms are 1) the CSEBS (Communication-Space Efficient Bitonic Sort) algorithm and 2) the CSEOE (Communication-Space Efficient Odd-Even mergesort) algorithm. Our communication-space efficient method consists of three communication patterns for the merge-and-split operation: 1) hold pattern, 2) swap pattern, and 3) new efficient partial pattern. Our new efficient partial pattern, the last one, is proposed to reduce not only communication time but also the amount of data exchanged and memory storage. The first two patterns, which hardly occur, are adopted from the previous study [8]. Each of three patterns is illustrated as follows:

Hold Pattern

The hold pattern occurs when two communicated processors P_i and P_j hold local data sub-sets that satisfy the merging condition and hence processors P_i and P_j do not need to exchange their local data.

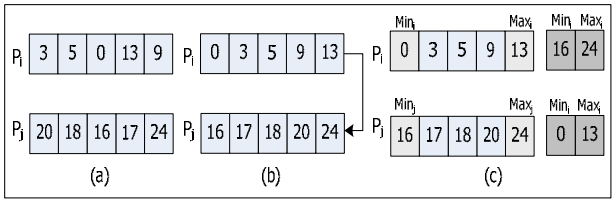


Figure 5. Hold Pattern.

For example (Figure 5), if N/P = 5 then processors P_i and P_j hold five elements {3, 5, 0,13,9} and {20,18,16,17,24}. Figure 5(a) and (b) show initial local data and local sorted data in processors P_i and P_j. Before data exchange between P_i and P_j (Figure 5(c)), processor P_i sends (Min_i, Max_i) = (0, 13) to P_j and processor P_j sends (Min_j, Max_j) = (16, 24) to P_i. Then, processor P_i is compared Min_j (16) to Max_i (13). In this case, the hold pattern is satisfied since Min_j is greater than Max_i and hence no data exchange.

Swap Pattern

The swap pattern occurs when two communicated processors P_i and P_j hold dual local data that satisfy the exchange condition with no overlapping and hence processors P_i and P_j need to exchange their local data.

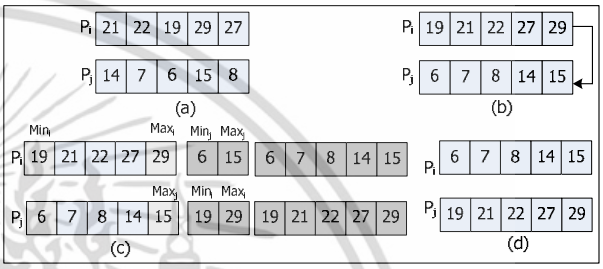


Figure 6. Swap Pattern.

For example (Figure 6), if N/P = 5 then processors P_i and P_j hold five elements {21,22,19,29,27} and {14,7,6,15,8}. Figure 6(a) and (b) show initial local data and sorted data in processors P_i and P_j. Before data exchange between P_i and P_j (Figure 6(c)), processor P_i sends (Min_i, Max_i) = (19, 29) to P_j and processor P_j sends (Min_j, Max_j) = (6, 15) to P_i. Then, processor P_i compared Max_j (15) to Min_i (19). In this case, the swap pattern is satisfied since Min_i is greater than Max_j and hence total data exchange is required.

Our New Efficient Partial Pattern

The new efficient partial pattern is activated when two communicated processors P_i and P_j hold local data that satisfy the exchange condition with some overlapping sub-sequences in the merged result and hence processors P_i and P_j need to exchange their partial data.

Then, only one of two processors (P_k = P_i or P_j) that has the minimum amount of partial (sent) data (of size n_k), satisfying the condition n_k = min (n_k, N/P - n_k), is the sender with at most $^{N}/_{(2P)}$ elements. After the corresponding receiver performs the merged result, it also sends the minimum amount of required data back.

But in the previous study [8], both P_i and P_j sent their partial data to corresponding partners with at most ($^{N}/_{P}$)-1 elements in one processor (in worse case). That pattern required twice extra space for receiving partial data in both processors. Therefore, our new efficient partial pattern can reduce both communication time (in sending minimum amount of data) and save memory space (in receiver only). Clearly, this new pattern strategy (compared to [8]) can save memory space up to 50%.

For example (Figure 7), if N/P = 5 then processors P_i and P_j hold five elements {6, 1, 11, 15, 8} and {16, 20,

13, 10, 12}. Figure 7(a) and (b) show initial local data and local sorted data in processors P_i and P_j. Before data exchange between P_i and P_j (Figure 7(c)), processor P_i sends $(Min_i, Max_i) = (1, 15)$ to P_j and processor P_j sends $(Min_j, Max_j) = (10, 20)$ to P_i. If Max_i (15) is more than Min_j (10) then in parallel processor P_i and P_j will find the minimum amount of partial (sent) data, which are (11, 15) and (10, 12, 13) respectively and hence only processor P_i, which has less amount of (sent) data, will be the sender to corresponding receiver (P_j). Finally (Figure 7(d)), processor P_j will perform the merge data and send the minimum required result (10, 11) back to P_i.

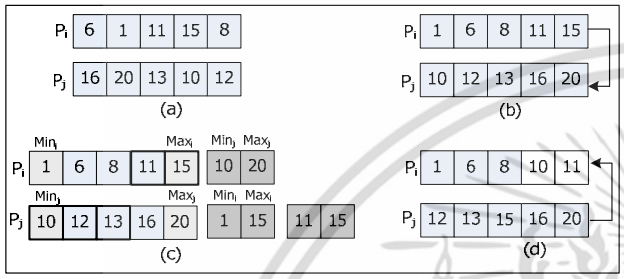


Figure 7. New Efficient Partial Pattern.

4. Experimental Results

In performance evaluation, we have implemented our communication-space efficient parallel sorting approach (the CSEBS and CSEOE algorithms) by using MPI (Massage Passing Interface) library on the SGI Origin2000 system. In our environment, the system consists of 8 processors (MIPS R10000 and R12000) with 2 GB RAM, connected via a high speed 1.56 Gbps processor connection and 780 Mbps memory connection.

A number of experiments were performed by varying a number of processors (P = 1, 2, 4, 8) and various problem sizes (N = 10, 20, 30, 40, …, 100 millions). In each tested case, a number of iterations with different (generated) data were executed and average performance results were computed in terms of average response time, average speedup, and average efficiency, where speedup $S = T_1/T_P$ and efficiency $E = S/P$, T_1 is the executive time of running sequential program on one processor and T_p is the executive time of running parallel program on P processors.

In each experiment, evaluated results of our new proposed algorithms (CSEBS: Communication-Space Efficient Bitonic Sorting and CSEOE: Communication-Space Efficient Odd-Even mergesort) were investigated and hence were compared to those of existing approaches (CEBS: Communication-Efficient Bitonic Sorting [8] and OE: Odd-Even merge sorting or PCM (Partition and Concurrent Merging) [2].)

Table 1 showed the average response time (T_p) of our CSEBS and CSEOE strategies and recently strategies (OE and CEBS) for various problem sizes $N = 10^7$ up to 10^8 elements and various processor P = 1, 2, 4, 8 processors. When fixing the problem size (N) and increasing a number of processors (P = 2, 4, 8), the average response time of all methods decreased. When fixing a number of processors (Ps) and increasing the problem sizes ($N = 10^7$ - 10^8), the average response time of all methods increased. Among all 4 methods (CSEBS, CSEOE, CEBS, OE), our CSEBS strategy yielded the best results (or minimum response time), which were improved over those of the OE and CEBS methods at least 24% and 11% respectively.

Table 1. Response Time (seconds) of 1, 2, 4, 8 Processor (s) and 10^7 – 10^8 Elements

N Million	P	T1(s)	Tp(s) OE	Tp(s) CEBS	Tp(s) CSEOE	Tp(s) CSEBS
10	2	2.96	2.79	1.99	1.95	1.64
	4		2.54	1.38	1.57	1.06
	8		1.88	0.54	0.94	0.42
20	2	4.46	3.73	2.94	2.98	2.17
	4		3.11	1.50	1.85	1.28
	8		1.92	0.85	1.02	0.72
30	2	6.62	5.64	4.34	4.25	3.72
	4		4.86	3.02	3.18	2.51
	8		2.89	1.96	1.76	1.03
40	2	7.53	7.00	5.34	5.18	4.36
	4		4.70	3.59	3.22	2.92
	8		3.68	1.75	1.75	1.27
50	2	11.09	8.80	6.58	6.76	5.92
	4		5.96	4.89	4.37	3.36
	8		4.23	2.05	2.08	1.85
60	2	13.14	9.94	7.85	7.79	6.97
	4		7.17	5.32	5.48	4.42
	8		4.95	3.19	3.10	2.06
70	2	16.51	11.08	9.49	9.95	8.76
	4		8.17	6.29	6.32	5.58
	8		6.27	4.10	4.11	3.07
80	2	19.08	12.20	10.69	11.39	10.08
	4		8.84	7.16	7.38	6.60
	8		6.27	4.80	4.75	3.99
90	2	21.41	13.75	11.60	12.77	11.00
	4		10.50	8.21	8.78	7.62
	8		8.41	6.16	5.18	4.12
100	2	25.50	16.30	13.69	13.93	12.92
	4		11.46	9.87	9.86	8.76
	8		9.59	7.29	6.19	5.05

When fixing N = 100 million elements and varying P = 2, 4, 8 processors, the response time, speedup, and efficiency of our and existing methods were illustrated in Figure 7, 8, and 9 respectively.

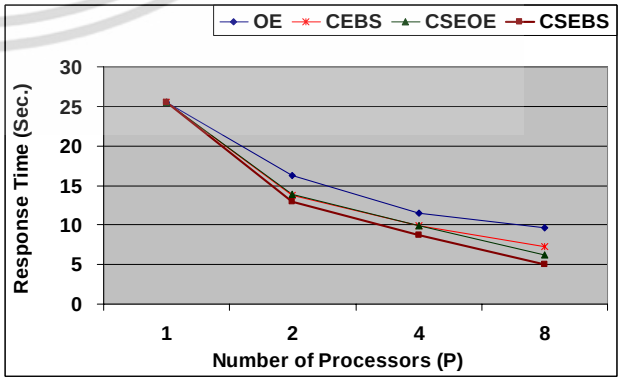


Figure 8. Response time of all methods (N = 100 M)

Figure 8 presented the average response time of all methods (CSEBS, CSEOE, CEBS, OE) and our CSEBS performed the best (or minimum) response time, which improved over those of the OE and CEBS 20% and 6% when using P = 2, 24% and 11% when using P = 4, and 47% and 31% when using P = 8, respectively.

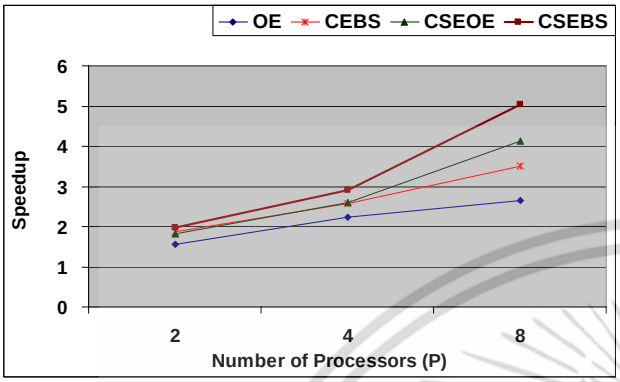


Figure 9. Speedup of all methods (N = 100 M)

Figure 9 showed the average speedup of all methods (CSEBS, CSEOE, CEBS, OE) and our CSEBS yielded the best (or maximum) speedup in all test cases (P = 2, 4, and 8). In particular, when using P = 2, the speedup of the CSEBS is 1.97 closed to 2 (ideal case), while those of the CSEOE, CEBS, and OE methods were 1.83, 1.86, 1.56 respectively. When increasing a number of processors (P = 4, 8), the speedup of all methods increased but in decreasing rate because the more processors used, the more communication time required.

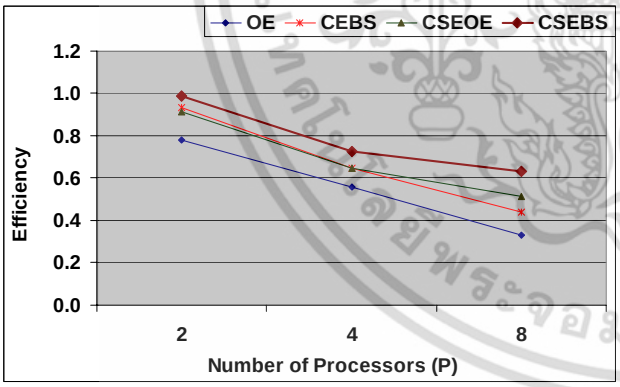


Figure 10. Efficiency of all methods (N = 100 M)

Figure 10 showed the average efficiency of all methods (CSEBS, CSEOE, CEBS, OE) and our CSEBS yielded the best (or maximum) efficiency in all test cases (P = 2, 4, and 8). Especially when using P = 2, the speedup of the CSEBS is 0.99 closed to 1 (ideal case), while those of the CSEOE, CEBS, and OE methods were 0.92, 0.93, 0.78 respectively. When increasing a number of processors (P = 4, 8), the efficiency of all methods decreased because of the more communication among processors.

5. Conclusion

In this study, we introduce a new efficient data-exchange process and applied in two popular parallel sorting algorithms (Bitonic sort (BS) and Odd-Even mergesort (OE)). Our proposed algorithms are 1) the CSEBS (Communication-Space Efficient Bitonic Sort) algorithm and 2) the CSEOE (Communication-Space Efficient Odd-Even MergeSort) algorithm.

A number of experiments were performed on the SGI Origin2000. The experimental results showed that in all tested cases, our proposed CSEBS algorithm yielded the improved results over those of the best of existing algorithm (CEBS: Communication-Efficient Bitonic Sorting) at least 11% (P = 4) and at least 31% (P = 8) and also can save memory space up to 50%.

Acknowledgement

We would like to thank the National Electronics and Computer Technology Center (in Thailand), National Science and Technology Development Agency (in Thailand) for providing computing resources that have contributed to the research results reported in this paper. URL: http://www.lsr.nectec.or.th.

References

[1] D. R. Helman, and J. JaJa, Sorting on Cluster of SMPs. *12th International Parallel Processing Symposium,* University of Maryland, College Park, MD, USA, 1997.

[2] E. Herruzo, G. Ruiz, J. I. Benavides, A New Parallel Sorting Algorithm based on Odd-Even Mergesort. *15th EUROMRICRO Int' l Conference on Parallel, Distributed and Network-Based Systems (PDP'07),* Naples, Italy, 2007.

[3] J. Brest, A. Vreze, and V. Zumer, A sorting algorithm on a pc cluster. *Proceedings 2000 ACM Symposium on Applied Computing (SAC'00)*, Como, Italy, 2000**,** 710-715.

[4] J. D. Lee, K. E. and Batcher, Minimizing Communication in the Bitonic Sort. *IEEE Transaction on Parallel and Distributed Systems,* 2000, 459-473

[5] K. E. Batcher, Sorting networks and their applications. *Proceedings Spring Joint Computing Conference AFIPS*, Washington DC, 1968, 307-314.

[6] M. F. Ionescu, and K. E. Schauser, Optimizing Parallel Bitonic Sort. *Proceedings 11th Int'l Parallel Processing Symposium*, 1997, 303-309.

[7] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the massage passing interface* (Cambridge, MA: MIT Press, 1994).

[8] Y. C. Kim, M. Jeon, D. Kim, and A. Sohn, Communication-Efficient Bitonic Sort on a Distributed Memory Parallel Computer. *Int' l Conference Parallel and Distributed Systems,* 2001, 165-170.

ประวัติผู้วิจัย

ชื่อ – สกุล	นางสาวธิปภร ธนกุลวรภาส
วัน เดือน ปีเกิด	3 พฤษภาคม 2523
ที่อยู่	24/4 หมู่ 2 ตำบลตาก้อง อำเภอเมือง จังหวัดนครปฐม 73000

ประวัติการศึกษา

2548	จบการศึกษาปริญญาวิทยาศาสตรมหาบัณฑิต สาขาวิทยาการคอมพิวเตอร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
2546	จบการศึกษาปริญญาวิทยาศาสตรบัณฑิต สาขาวิทยาการคอมพิวเตอร์ มหาวิทยาลัยมหิดล

ประวัติการทำงาน

มิถุนายน 2544	เจ้าหน้าที่ Administrator บริษัท ทรู คอร์ปอเรชั่น จำกัด (มหาชน)
ตุลาคม 2544	เจ้าหน้าที่ Consult Internet มหาวิทยาลัยมหิดล
2548-2555	ตำแหน่งผู้ช่วยนักวิจัย สังกัดห้องปฏิบัติการวิจัยวิทยาการมนุษยภาษา ศูนย์เทคโนโลยีอิเล็กทรอนิกส์และคอมพิวเตอร์แห่งชาติ (NECTEC)