ขั้นตอนวิธีการประมวลผลแบบขนานเพื่อการเข้ารหัสของฮัฟแมนที่มีประสิทธิภาพ

AN EFFICIENT PRACTICAL PARALLEL HUFFMAN CODING ALGORITHM

พัชรินทร์ บัวเย็น

PATCHARIN BUAYEN

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต

สาขาวิชาวิทยาการคอมพิวเตอร์

บัณฑิตวิทยาลัย

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2551

KMITL-2008-SC-M-002-348

ขั้นตอนวิธีการประมวลผลแบบขนานเพื่อการเข้ารหัสของฮัฟแมนที่มีประสิทธิภาพ

AN EFFICIENT PRACTICAL PARALLEL HUFFMAN CODING ALGORITHM

พัชรินทร์ บัวเย็น

PATCHARIN BUAYEN

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรวิทยาศาสตรมหาบัณฑิต

สาขาวิชาวิทยาการคอมพิวเตอร์

บัณฑิตวิทยาลัย

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2551

KMITL-2008-SC-M-002-348

AN EFFICIENT PRACTICAL PARALLEL HUFFMAN CODING ALGORITHM

PATCHARIN BUAYEN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE

SCHOOL OF GRADUATE STUDIES

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2008

KMITL-2008-SC-M-002-348

บัณฑิตวิทยาลัย

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ใบรับรองวิทยานิพนธ์

หัวข้อวิทยานิพนธ์ ขั้นตอนวิธีการประมวลผลแบบขนานเพื่อการเข้ารหัสของฮัฟแมนที่มีประสิทธิภาพ

An Efficient Practical Parallel Huffman Coding Algorithm

ชื่อนักศึกษา นางสาวพัชรินทร์ บัวเย็น

รหัสประจำตัว 49067501

ปริญญา วิทยาศาสตรมหาบัณฑิต

สาขาวิชา วิทยาการคอมพิวเตอร์

อาจารย์ที่ปรึกษาวิทยานิพนธ์ ผศ.ดร.จีรพร วีระพันธุ์

คณะกรรมการสอบวิทยานิพนธ์		ลายมือชื่อ
ผศ.ดร.ศรัณย์	อินทโกสุม	
ผศ.ดร.จีรพร	วีระพันธุ์	
รศ.ดร.วีระ	บุญจริง	
ดร.เฉลิมศักดิ์	เลิศวงศ์เสถียร	

วัน / เดือน/ ปี ที่สอบ 26 พฤษภาคม 2551 เวลา 16.00 น. เป็นต้นไป

สถานที่สอบ ณ อาคารจุฬาภรณวลัยลักษณ์ 1 ห้อง 210

บัณฑิตวิทยาลัยรับรองแล้ว

(รศ.ดร.รวีวรรณ ชินะตระกูล)

คณบดีบัณฑิตวิทยาลัย

วันที่.....๒๗.....**เดือน**.....มิถุนายน.....**พ.ศ**.....๒๕๕๑.....

หัวข้อวิทยานิพนธ์	ขั้นตอนวิธีการประมวลผลแบบขนานเพื่อการเข้ารหัสของฮัฟแมนที่มีประสิทธิภาพ
นักศึกษา	นางสาวพัชรินทร์ บัวเย็น
รหัสประจำตัว	49067501
ปริญญา	วิทยาศาสตรมหาบัณฑิต
สาขาวิชา	วิทยาการคอมพิวเตอร์
พ.ศ.	2551
อาจารย์ที่ปรึกษาวิทยานิพนธ์	ผศ.ดร.จีรพร วีระพันธุ์

บทคัดย่อ

งานวิจัยนี้นำเสนอขั้นตอนวิธีการประมวลผลแบบขนานเพื่อการเข้ารหัสของฮัฟแมนที่มีประสิทธิภาพสำหรับพีแรมโมเดลแบบซีอาร์อีดับเบิลยู (CREW PRAM model : Concurrent Read, Exclusive Write PRAM model) และนำวิธีการประมวลผลแบบบีเอสอาร์ (BSR : Broadcasting with Selective Reduction) เข้ามาประยุกต์ใช้ร่วมกับขั้นตอนวิธีการที่นำเสนอ โดยขั้นตอนวิธีการที่นำเสนอดังกล่าวเน้นการลดความซับซ้อนของขั้นตอนการประมวลผลและใช้หน่วยความจำอย่างมีประสิทธิภาพ ซึ่งมีความซับซ้อนด้านเวลาในกรณีที่ต้นไม้ฮัฟแมนมีการกระจายกิ่งแบบสมดุลหรือกรณีดีที่สุดเป็น O(log n) และมีความซับซ้อนด้านเวลาในกรณีที่ต้นไม้ฮัฟแมนมีการกระจายกิ่งไปในทิศทางเดียวหรือกรณีที่แย่ที่สุดเป็น O(n) นอกจากนี้รหัสบิตที่ได้จากการประมวลผลด้วยวิธีการที่นำเสนอในงานวิจัยนี้มีลักษณะเหมือนกับรหัสบิตที่ได้จากการประมวลผลด้วยวิธีการเข้ารหัสของฮัฟแมนแบบดั้งเดิมเนื่องจากใช้หลักการในการสร้างต้นไม้ฮัฟแมนแบบเดียวกัน ดังนั้นชุดรหัสบิตที่ได้จากวิธีการนี้สามารถนำไปถอดรหัสด้วยขั้นตอนวิธีการถอดรหัสที่มีประสิทธิภาพซึ่งรองรับชุดรหัสบิตของฮัฟแมนได้

Thesis Title	An Efficient Practical Parallel Huffman Coding Algorithm
Student	Miss Patcharin Buayen
Student ID.	49067501
Degree	Master of Science
Program	Computer Science
Year	2008
Thesis Advisor	Asst. Prof. Dr. Jeeraporn Werapun

ABSTRACT

This research proposes an efficient parallel algorithm for constructing Huffman codes on CREW-PRAM model (Concurrent Read, Exclusive Write PRAM model) using BSR (Broadcasting with Selective Reduction). This approach presents the efficient memory space as well as simplifies the coding process. Time complexity of this efficient proposed parallel algorithm is O(log n) time in the best case when the constructed Huffman tree is nearly balanced and O(n) time in the worst case when the Huffman tree is in one-sided form. In addition, the Huffman codes of this approach are generated with theoretical the same as those processed from the original Huffman algorithm. Thus the results of this coding can be decoded by any efficient algorithm that supports the original Huffman codes.

กิตติกรรมประกาศ

วิทยานิพนธ์นี้มิอาจจะสำเร็จลุล่วงไปได้ด้วยดี หากมิได้รับคำแนะนำ คำชี้แจง ความรู้ และความเอาใจใส่จาก ผศ. ดร.จีรพร วีระพันธุ์ ผู้เป็นอาจารย์ควบคุมวิทยานิพนธ์ ซึ่งท่านได้สละเวลาให้กับข้าพเจ้าอย่างเต็มที่ จึงใคร่ขอขอบพระคุณเป็นอย่างสูง

ขอขอบพระคุณ รศ. ดร.วีระ บุญจริง และ ผศ.ดร.ศรัณย์ อินทโกสุม สำหรับคำแนะนำและการให้คำปรึกษาต่างๆ จนในที่สุดทำให้วิทยานิพนธ์ฉบับนี้สำเร็จลุล่วงไปได้ด้วยดี และยังให้ความกรุณามาเป็นตัวแทนกรรมการจากภาควิชา

ขอขอบพระคุณ ดร.เฉลิมศักดิ์ เลิศวงศ์เสถียร ซึ่งให้ความกรุณามาเป็นตัวแทนกรรมการจากบุคคลภายนอก ทำให้ได้รับคำแนะนำต่างๆ ในการสอบวิทยานิพนธ์

ขอขอบพระคุณคณาจารย์ประจำภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบังทุกท่าน ที่ได้ประสิทธิ์ประสาทวิชาความรู้ต่างๆ ให้กับข้าพเจ้าในระดับปริญญาโท

ขอขอบพระคุณโครงการทุนการศึกษาสำหรับบุคคลทั่วไป ของมหาวิทยาลัยศรีปทุม วิทยาเขตชลบุรีที่ให้ทุนการศึกษาตลอดหลักสูตร

ขอขอบพระคุณบิดา มารดาและพี่ๆ ที่ให้การสนับสนุนด้านการเรียนมาโดยตลอด อีกทั้งยังเป็นกำลังใจอันยิ่งใหญ่ในการเรียนและการทำวิทยานิพนธ์ฉบับนี้ให้ลุล่วงไปได้ด้วยดี

ขอขอบคุณ นางสาวจิราภรณ์ ทัดรัตน์ นางสาวจารุณี แซ่หลี ที่ได้ให้คำแนะนำเกี่ยวกับการทำวิทยานิพนธ์ การเขียนบทความและคอยให้ความช่วยเหลือและให้กำลังใจมาโดยตลอด

ขอขอบคุณ นายไพโรจน์ สมุทรักษ์ นางสาวจุรีพร บุญนิยม และพี่ๆทุกคนในระดับปริญญาเอกที่ได้ให้คำแนะนำในด้านต่างๆ และคอยให้กำลังใจมาโดยตลอด

ขอขอบคุณ นางสาวปรัศนี แจ้งสกุล ที่คอยให้ความช่วยเหลือในด้านต่างๆ และคอยให้กำลังใจมาโดยตลอด ทั้งในด้านการเรียนและการทำวิทยานิพนธ์

ขอขอบคุณ นางสาวสุวารี แก้วปรารถนา นางสาวรสลิน เพตะกร รวมถึงพี่ๆ เพื่อนๆ และน้องๆ ในระดับปริญญาโททุกคนที่คอยให้กำลังใจในการทำวิทยานิพนธ์

สำหรับคุณงามความดีและประโยชน์อันใดที่เกิดขึ้นจากวิทยานิพนธ์ฉบับนี้ ข้าพเจ้าขอมอบให้แก่บิดา มารดา อาจารย์ทุกท่านซึ่งเป็นที่เคารพรักยิ่ง ตลอดจนญาติพี่น้อง และเพื่อน ๆ ทุกคน

พัชรินทร์ บัวเย็น

สารบัญ

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

สารบัญรูป

สารบัญรูป (ต่อ)

สารบัญรูป (ต่อ)

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

จากอดีตจนถึงปัจจุบันได้มีการคิดค้นและพัฒนาวิธีการบันทึกข้อมูลข่าวสาร ตั้งแต่การใช้วัสดุจากธรรมชาติมาประยุกต์เพื่อการจดบันทึก จนกระทั่งพัฒนามาเป็นยุคของการใช้เทคโนโลยีซึ่งเทคโนโลยีในปัจจุบันทำให้อุปกรณ์ที่ใช้ในการเก็บข้อมูลในคอมพิวเตอร์มีขนาดเล็กลงในขณะที่มีความจุมากขึ้น โดยอุปกรณ์ดังกล่าวอาจจะเป็นเทป ฮาร์ดดิสก์ ซีดีรอม หรือแฟลช-ไดรฟ์ แต่อุปกรณ์จัดเก็บข้อมูลของคอมพิวเตอร์ต่างๆ ก็ยังมีข้อจำกัดด้านขนาดเนื้อที่ในการจัดเก็บข้อมูล โดยเฉพาะปัจจุบันเป็นยุคที่ข้อมูลข่าวสารมีความสำคัญและจำเป็นต่อชีวิตมนุษย์อย่างขาดไม่ได้ ทำให้แนวโน้มของข้อมูลสารสนเทศมีขนาดเพิ่มมากขึ้นอยู่ตลอดเวลา ซึ่งไม่สอดคล้องกับขนาดของอุปกรณ์จัดเก็บข้อมูลที่พัฒนาขึ้นในขณะนั้น นักวิจัยจึงได้คิดค้นและพัฒนาขั้นตอนวิธีการลดขนาดข้อมูลให้เล็กลงเพื่อสามารถนำไปบันทึกข้อมูลได้อย่างมีประสิทธิภาพ แต่ขั้นตอนการประมวลผลดังกล่าวมีความซับซ้อนมากและเมื่อนำมาใช้ในการลดขนาดข้อมูลที่มีขนาดใหญ่จะทำให้ใช้เวลาในการประมวลผลมากขึ้นตามไปด้วย วิธีการหนึ่งที่จะช่วยลดปัญหาดังกล่าวได้คือการประยุกต์ใช้การประมวลผลแบบขนานเข้ามาร่วมกับขั้นตอนการลดขนาดข้อมูล

ขั้นตอนวิธีการหนึ่งที่นิยมนำมาพัฒนาเพื่อลดขนาดข้อมูลคือวิธีเข้ารหัสของฮัฟแมน [5][7] [17][20] ซึ่งเป็นวิธีที่ใช้ชุดรหัสบิตที่สั้นที่สุดแทนสัญลักษณ์ที่เกิดขึ้นบ่อยที่สุดในข้อมูล โดยใช้โครงสร้างต้นไม้สองทาง (Binary Tree) ที่มีชื่อเรียกว่า "ต้นไม้ฮัฟแมน (Huffman Tree)" สำหรับสร้างรหัส ขั้นตอนวิธีการเข้ารหัสของฮัฟแมนนี้มีความซับซ้อนด้านเวลาเท่ากับ O(n log n) [9][10] ซึ่งจะใช้เวลามากในการประมวลผลเมื่อข้อมูลนำเข้ามีขนาดใหญ่มากๆ ด้วยเหตุนี้จึงได้มีการนำขั้นตอนการประมวลผลแบบขนานสำหรับสถาปัตยกรรมคอมพิวเตอร์ที่หน่วยประมวลผลหลายๆ หน่วยสามารถเข้าถึงหน่วยความจำในเวลาเดียวกันมาใช้ร่วมกับวิธีการเข้ารหัสของฮัฟแมนเพื่อให้ประมวลผลได้เร็วขึ้น โดย Milidiu และทีมงาน [18] ได้พัฒนาแนวคิดดังกล่าวด้วยการสร้างต้นไม้ฮัฟแมนบนกระบวนการประมวลผลแบบขนานด้วยความซับซ้อนด้านเวลาเป็นO(log(1/p1)log(log n)) เมื่อ p1 คือผลหารระหว่างความถี่ของสัญลักษณ์แรกกับความถี่ของสัญลักษณ์ทั้งหมด แต่ไม่ได้พัฒนาในส่วนการแสดงรหัสที่สร้างจากการค้นหาเส้นทางตั้งแต่โหนดราก (Root Node) จนถึงโหนดใบ (Leaf Node) ทั้งหมด ต่อมา Ostadzadeh และทีมงาน [19] ได้นำขั้นตอนวิธีที่นำเสนอในเอกสารอ้างอิงที่ [18] มาพัฒนาต่อ เพื่อให้สามารถสร้างรหัสของฮัฟแมนได้ โดยใช้โครงสร้างอะเรย์แทนข้อมูลในต้นไม้ฮัฟแมนสำหรับใช้ในการประมวลผล ซึ่งได้แบ่งขั้นตอนการประมวลผล

ออกเป็น 2 ส่วน คือ ส่วนสำหรับคำนวณหาความยาวของชุดรหัสบิต (CLGeneration : Codes Length Generation) และส่วนการสร้างรหัสบิต (CWGeneration : Codeword Generation) ซึ่งจะนำค่าความยาวของชุดรหัสบิตที่ได้จากขั้นตอนแรกมาคำนวณเพื่อแปลงเป็นรหัสบิตสำหรับนำไปใช้แทนสัญลักษณ์แต่ละสัญลักษณ์ในชุดข้อมูล ข้อดีของวิธีนี้คือมีความซับซ้อนด้านเวลาเป็น O(log((log(n-1))!)) ซึ่งน้อยกว่าวิธีของ Milidiu [18] เมื่อใช้หน่วยประมวลผลเท่ากันซึ่งเท่ากับขนาดของข้อมูล (n) แต่วิธีการนี้มีข้อจำกัดคือมีขั้นตอนการประมวลผลที่ซับซ้อนและใช้อะเรย์ขนาดใหญ่หลายอะเรย์ในการประมวลผลข้อมูลทำให้ใช้เนื้อที่ในหน่วยความจำเป็นจำนวนมาก

งานวิจัยนี้ได้นำเสนอขั้นตอนวิธีการเข้ารหัสของฮัฟแมนด้วยการประมวลผลแบบขนานที่ลดความซับซ้อนของขั้นตอนในการประมวลผลลง และมีความซับซ้อนด้านเวลาในกรณีที่ดีที่สุดลดลง โดยรหัสบิตที่ได้จะมีลักษณะตามทฤษฎีเหมือนการเข้ารหัสของฮัฟแมนและเมื่อนำขั้นตอนวิธีการดังกล่าวมาใช้กับโครงสร้างอะเรย์ พบว่ามีการใช้หน่วยความจำในระหว่างการประมวลผลน้อยกว่าวิธีการเดิม [19] นอกจากนี้ได้ประยุกต์ใช้วิธีการรวมข้อมูลแบบบีเอสอาร์ (Merging on BSR : Merging on Broadcasting with Selective Reduction) ที่มีความซับซ้อนด้านเวลาเท่ากับ O(1) ในขั้นตอนย่อยของการสร้างรหัสบิตซึ่งช่วยลดความซับซ้อนด้านเวลาในการประมวลผลเมื่อเปรียบเทียบกับ 2 วิธีที่มีผู้เสนอไว้แล้วข้างต้น [18][19] ที่ได้ใช้วิธีการรวมข้อมูลแบบขนาน (Parallel Merging) ที่มีความซับซ้อนด้านเวลาเป็น O(log(log n)) จึงทำให้ขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอนี้มีความซับซ้อนด้านเวลาเป็น O(log n) ในกรณีที่ต้นไม้ฮัฟแมนมีการกระจายกิ่งแบบสมดุล และมีความซับซ้อนด้านเวลาเป็น O(n) ในกรณีที่ต้นไม้ฮัฟแมนมีการกระจายกิ่งไปในทิศทางเดียว

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา

วิทยานิพนธ์นี้มีวัตถุประสงค์เพื่อพัฒนาขั้นตอนวิธีการเข้ารหัสของฮัฟแมนบนสถาปัตยกรรมแบบขนานที่ใช้หน่วยความจำร่วม (Shared Memory Parallel Architecture) บนพีแรมโมเดลแบบซีอาร์อีดับเบิลยู (CREW PRAM model) ที่มีผู้ได้นำเสนอไว้ในเอกสารอ้างอิงที่ [18][19] โดยออกแบบและปรับเปลี่ยนขั้นตอนวิธีการสร้างรหัสบิตตามลักษณะของฮัฟแมนให้มีขั้นตอนกระบวนการประมวลผลที่ง่ายขึ้นและใช้หน่วยความจำในระหว่างการประมวลผลน้อยกว่าวิธีการเดิม อีกทั้งขั้นตอนวิธีที่นำเสนอในงานวิจัยนี้ได้นำกระบวนการประมวลผลแบบบีเอสอาร์ (BSR : Broadcasting with Selective Reduction) และการรวมข้อมูลแบบบีเอสอาร์ (Merging on BSR) มาประยุกต์ใช้เพื่อลดเวลาในการประมวลผลซึ่งทำให้ได้ค่าความซับซ้อนด้านเวลาในกรณีที่ดีที่สุดเป็น O(log n) ซึ่งน้อยกว่าวิธีการเดิมที่มีนักวิจัยได้นำเสนอไว้

1.3 สมมุติฐานของการศึกษา

1) การลดขั้นตอนการคำนวณความยาวของรหัสบิตก่อนการสร้างรหัสบิตที่ประมวลผลในแบบขนาน ทำให้ลดความซับซ้อนในขั้นตอนการประมวลผลและลดการใช้พื้นที่ในหน่วยความจำในการประมวลผล
2) การนำการประมวลผลแบบบีเอสอาร์ (BSR : Broadcasting with Selective Reduction) และการรวมข้อมูลแบบบีเอสอาร์ (Merging on BSR) มาประยุกต์ใช้ในขั้นตอนการสร้างรหัสของฮัฟแมนทำให้ลดความซับซ้อนด้านเวลา

1.4 ขอบเขตการวิจัย

ขอบเขตของวิทยานิพนธ์ฉบับนี้ มีดังต่อไปนี้

1) งานวิจัยนี้ครอบคลุมเฉพาะส่วนการเข้ารหัสข้อมูลเท่านั้น ไม่ครอบคลุมส่วนของการถอดรหัสข้อมูล
2) ชุดรหัสบิตที่ได้จากการเข้ารหัสของฮัฟแมนแบบขนานในงานวิจัยนี้สามารถนำไปถอดรหัสด้วยวิธีการถอดรหัสที่รองรับชุดรหัสบิตของฮัฟแมน
3) งานวิจัยนี้เปรียบเทียบค่าความซับซ้อนด้านเวลาของขั้นตอนวิธีการที่นำเสนอกับวิธีการเข้ารหัสของฮัฟแมนแบบขนานแบบเดิมที่ได้เสนอในเอกสารอ้างอิงที่ [18] และ [19]

1.5 ขั้นตอนการศึกษาและดำเนินงานวิจัย

วิทยานิพนธ์นี้มีขั้นตอนการศึกษาและการดำเนินงานวิจัย ดังนี้

1) ศึกษาขั้นตอนวิธีการเข้ารหัสของฮัฟแมนและงานวิจัยที่เกี่ยวข้อง ดังนี้
   - ศึกษาขั้นตอนวิธีการประมวลผลแบบขนาน
   - ศึกษางานวิจัยเกี่ยวกับการพัฒนาขั้นตอนวิธีบนการประมวลผลแบบขนาน
   - ศึกษางานวิจัยเกี่ยวกับการพัฒนาขั้นตอนวิธีการเข้ารหัสของฮัฟแมนบนกระบวนการแบบลำดับ
   - ศึกษางานวิจัยเกี่ยวกับการพัฒนากระบวนการเข้ารหัสของฮัฟแมนแบบขนานทั้งในแบบจำลองฮาร์ดแวร์และแบบจำลองหน่วยความจำร่วม
   - ศึกษางานวิจัยเกี่ยวกับการประมวลผลแบบบีเอสอาร์
   - ศึกษางานวิจัยเกี่ยวกับการรวมข้อมูลแบบบีเอสอาร์

2) ทำการตั้งสมมุติฐาน โดยคาดว่าขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอนี้จะสามารถลดความซับซ้อนของขั้นตอนในการประมวลผลและลดขนาดหน่วยความจำที่ใช้ในการประมวลผลลง และคาดว่าการนำวิธีการรวมข้อมูลแบบบีเอส-อาร์มาประยุกต์ใช้ในขั้นตอนการสร้างรหัสบิตของวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอนี้ทำให้ได้ความซับซ้อนด้านเวลาในกรณีที่ดีที่สุดเป็น O(log n)
3) วิเคราะห์ความซับซ้อนด้านเวลาของขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอ และเปรียบเทียบค่าความซับซ้อนด้านเวลาของขั้นตอนวิธีดังกล่าวกับค่าความซับซ้อนด้านเวลาของขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่ได้เสนอในเอกสารอ้างอิงที่ [18] และ [19] และเปรียบเทียบจำนวนและขนาดของหน่วยความจำที่ใช้ในการประมวลผลในขั้นตอนวิธีการดังกล่าวกับขั้นตอนวิธีการที่ได้มีผู้เสนอไว้ในเอกสารอ้างอิงที่ [19]
4) สรุปผล พร้อมเสนอแนวทางการพัฒนางานวิจัย
5) เขียนวิทยานิพนธ์และบทความวิจัย

1.6 ประโยชน์ที่คาดว่าจะได้รับ

วิทยานิพนธ์นี้มีประโยชน์ที่คาดว่าจะได้รับจากขั้นตอนวิธีการเข้ารหัสของฮัฟแมนบนกระบวนการแบบขนานที่นำเสนอในงานวิจัย ดังนี้

1) ลดความซับซ้อนของขั้นตอนในการประมวลผลสำหรับการลดขนาดข้อมูล
2) นำมาใช้เป็นแนวทางในการพัฒนาขั้นตอนวิธีการลดขนาดข้อมูลด้วยวิธีอื่น
3) นำมาใช้เป็นแนวทางในการพัฒนาโปรแกรมแบบขนานบนหน่วยความจำร่วมและการประมวลผลแบบบีเอสอาร์เพื่อให้สามารถใช้งานได้สะดวกและเข้าใจง่ายขึ้น
4) นำไปประยุกต์ในการรักษาความปลอดภัยของข้อมูลเพราะผลการเข้ารหัสที่ได้มีลักษณะเป็นรหัสบิตที่สอดคล้องกับข้อมูลโดยตรงซึ่งสามารถนำไปถอดรหัสได้ง่าย จึงควรมีการเพิ่มกระบวนการด้านความปลอดภัยให้กับรหัสบิตในขั้นตอนการประมวลผล

บทที่ 2
ทฤษฎีและงานวิจัยที่เกี่ยวข้อง

หัวข้อนี้จะกล่าวถึงทฤษฎีพื้นฐานต่างๆ ที่เกี่ยวข้องในการวิจัย ประกอบด้วย การลดขนาดข้อมูลด้วยวิธีการเข้ารหัสของฮัฟแมน สถาปัตยกรรมคอมพิวเตอร์แบบขนาน การลดขนาดข้อมูลด้วยการเข้ารหัสของฮัฟแมนบนกระบวนการแบบขนาน และขั้นตอนวิธีการรวมข้อมูลแบบบีเอสอาร์

2.1 การบีบอัดข้อมูล

การบีบอัดข้อมูล (Data Compression) [5][17][20] คือ กระบวนการลดขนาดข้อมูลให้มีขนาดเล็กลงทำให้สามารถเก็บข้อมูลได้มากขึ้น ซึ่งเทคโนโลยีนี้ถูกนำไปใช้ในงานด้านต่างๆ อาทิ การรักษาพื้นที่ว่างสำหรับดิสก์ การสำรองข้อมูล รวมถึงการลดเวลาที่ใช้ในการสื่อสารหรือเวลาที่ใช้ในการถ่ายโอนข้อมูล

การบีบอัดข้อมูลสามารถแบ่งออกเป็น 2 แบบ [17] คือ

1) การบีบอัดข้อมูลแบบที่มีการสูญเสียข้อมูล (Lossy Compression) เป็นการบีบอัดที่จะได้ข้อมูลที่เพี้ยนหรือสูญเสียข้อมูลบางส่วนไปจากข้อมูลต้นฉบับหลังจากการคลายข้อมูล ซึ่งการสูญเสียข้อมูลบางส่วนนี้เพื่อให้สามารถบีบอัดข้อมูลได้เล็กลง ข้อมูลที่เหมาะสำหรับการใช้วิธีการบีบอัดประเภทนี้ ได้แก่ รูปภาพ เสียง และวีดิโอ

2) การบีบอัดข้อมูบแบบไม่สูญเสียข้อมูล (Lossless Compression) เป็นการบีบอัดที่จะได้ข้อมูลที่เหมือนต้นฉบับทุกประการหลังจากการคลายข้อมูล ดังนั้นการบีบอัดข้อมูลประเภทนี้จึงถูกนำไปใช้กับข้อมูลที่จะสูญเสียส่วนใดๆ ในข้อมูลไม่ได้ อาทิ ข้อมูลประเภทข้อความหรือแฟ้มข้อมูล เป็นต้น การบีบอัดข้อมูลประเภทรูปภาพก็สามารถใช้แนวทางการบีบอัดแบบนี้ได้ โดยอาศัยหลักการลดความซ้ำซ้อนของข้อมูล (Redundancy Reduction) และการใช้จำนวนบิตที่เก็บค่าข้อมูลไม่คงที่ (Variable-Length Coding) เช่น การเข้ารหัสของฮัฟแมน

2.2 ขั้นตอนวิธีการเข้ารหัสของฮัฟแมน

วิธีการเข้ารหัสของฮัฟแมน (Huffman Coding Algorithm) [5][7][17][20] เป็นการแปลงความถี่ของการเกิดสัญลักษณ์ในชุดข้อมูลนำเข้าให้เป็นรหัสบิต โดยอาศัยหลักการกระจายสัญลักษณ์แต่ละตัวไปกับข้อมูลโหนดใบของโครงสร้างต้นไม้สองทาง (Binary Tree) ซึ่งความยาวของรหัสบิตที่ได้จะสั้นกว่ารหัสบิตที่แปลงจากสัญลักษณ์โดยตรงทำให้วิธีนี้ถูกนำมาพัฒนาเพื่อเพิ่ม

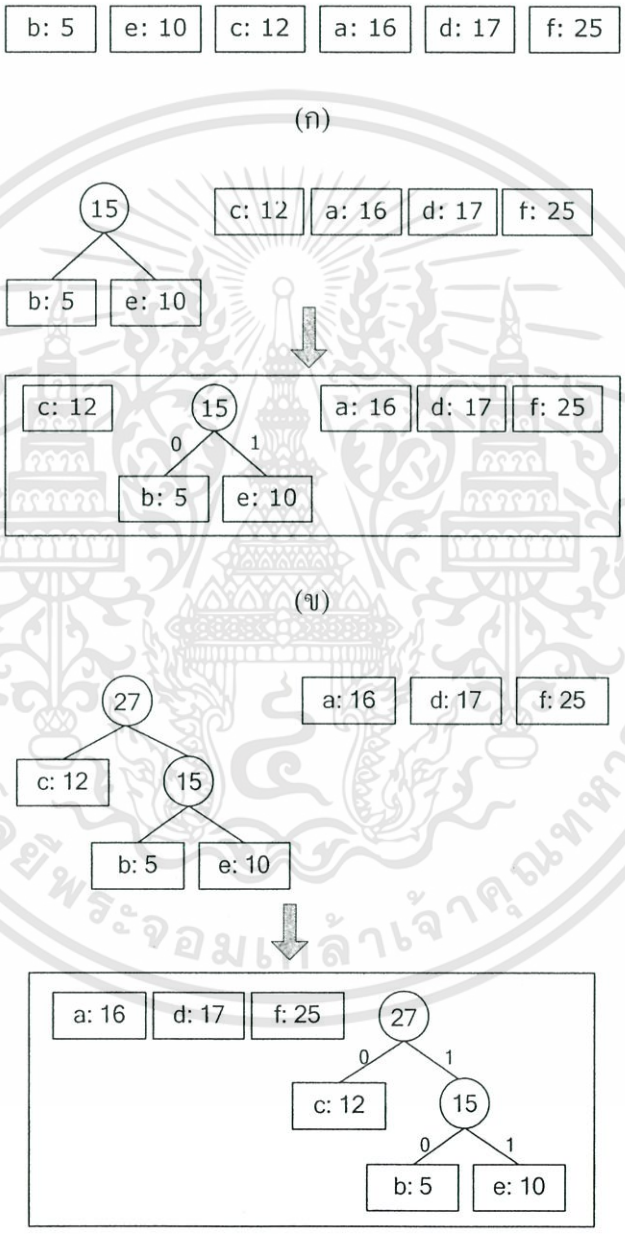
ประสิทธิภาพการทำงาน อาทิ การพัฒนาวิธีการเพื่อลดความซับซ้อนด้านเวลา [9] การพัฒนาเพื่อลดการใช้พื้นที่ของหน่วยความจำในการประมวลผล [4][10][23] ซึ่งบางขั้นตอนวิธีจะปรับโครงสร้างของต้นไม้ฮัฟแมนให้มีการกระจายกิ่งไปในทิศทางเดียว (SGH-Tree : Single Side Growing Huffman Tree) แล้วจึงนำมาสร้างเป็นตารางที่เรียกว่าเอสจีเอชที (SGHT : Single Side Growing Huffman Table) เพื่อลดการใช้พื้นที่ในหน่วยความจำ [4][23] นอกจากนั้นยังมีการพัฒนาวิธีการเข้ารหัสของฮัฟแมนให้มีรูปแบบของบิต '0' และบิต '1' ที่สมดุลกัน [15]

ในการสร้างรหัสบิตด้วยวิธีการเข้ารหัสของฮัฟแมนแทนสัญลักษณ์ภายในชุดข้อมูลจะกระทำโดยการกำหนดรหัสบิตบนต้นไม้ฮัฟแมนเพื่อให้ได้รหัสบิตที่นำมาแทนสัญลักษณ์ของข้อมูล โดยกำหนดให้กิ่งซ้ายของต้นไม้มีค่าบิตเป็น '0' และกิ่งขวามีค่าบิตเป็น '1' โดยในโครงสร้างต้นไม้ของฮัฟแมนจะประกอบด้วยโหนดที่แทนสัญลักษณ์ซึ่งจะอยู่ที่ปลายกิ่งของโครงสร้างต้นไม้เรียกว่า โหนดใบ (Leaf Node) และโหนดพ่อแม่ (Parent Node) หรือเรียกอีกชื่อว่า โหนดภายใน (Internal Node) เป็นโหนดที่สร้างมาจากการรวมความถี่ของโหนด 2 โหนด ซึ่งโหนดพ่อแม่นี้จะอยู่ในระดับที่สูงกว่าโหนดลูก (Child Node) และโหนดพ่อแม่ที่อยู่บนสุดของโครงสร้างต้นไม้จะเรียกว่า โหนดราก (Root Node)

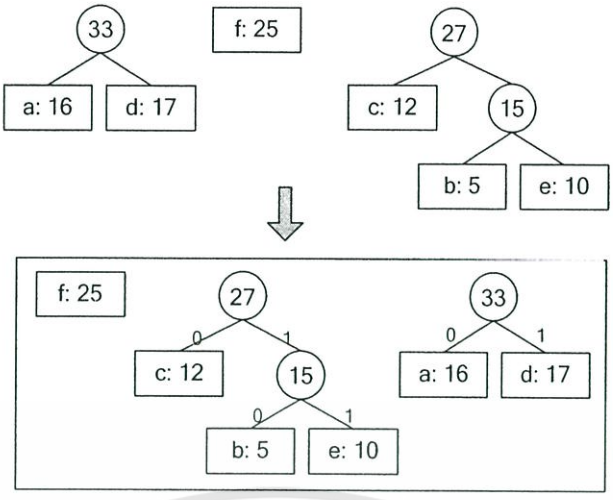
การสร้างรหัสบิตด้วยวิธีการเข้ารหัสของฮัฟแมน เริ่มจากการเลือกสมาชิกที่มีค่าความถี่ของการปรากฏสัญลักษณ์ในชุดข้อมูลที่น้อยที่สุด 2 สมาชิกมารวมกันเพื่อสร้างโหนดพ่อแม่ซึ่งจะได้ต้นไม้ย่อย (Sub Tree) ในโครงสร้างต้นไม้ของฮัฟแมน จากนั้นลบสมาชิกที่นำมารวมกันออกจากกลุ่มสมาชิกเดิม แล้วนำค่าของโหนดพ่อแม่ใหม่ที่สร้างขึ้นเพิ่มเข้าไปในกลุ่มสมาชิกนั้น โดยขั้นตอนการจับคู่และสร้างโหนดพ่อแม่ใหม่นี้จะกระทำจนกระทั่งได้โหนดราก เมื่อได้โครงสร้างต้นไม้แล้ว จากนั้นจะกำหนดบิต '0' ให้กับกิ่งซ้ายและกำหนดบิต '1' ให้กับกิ่งขวาของแต่ละกิ่งในต้นไม้ แล้วทำการค้นหาเส้นทางจากโหนดรากไปยังโหนดใบแต่ละโหนดซึ่งจะทำให้ได้รหัสบิตแทนสัญลักษณ์ที่ตรงกับโหนดใบนั้นๆ

ตัวอย่างเช่น รูปที่ 2.1 แสดงการทำงานในแต่ละรอบของขั้นตอนวิธีการสร้างโครงสร้างต้นไม้ของฮัฟแมน ในที่นี้สมมุติข้อมูลนำเข้าประกอบด้วยคู่ลำดับของสัญลักษณ์กับความถี่ดังนี้ b: 5, e: 10, c: 12, a: 16, d: 17 และ f: 25 โดยเริ่มจากการจัดเรียงสัญลักษณ์ตามความถี่ของการปรากฏสัญลักษณ์จากน้อยไปมาก ซึ่งจะได้ผลดังรูปที่ 2.1(ก) จากนั้นทำการรวมความถี่ที่น้อยที่สุด 2 ค่าเพื่อสร้างโหนดพ่อแม่ ในที่นี้ได้โหนดพ่อแม่ใหม่ที่มีความถี่ 15 ซึ่งถูกสร้างจากการรวมความถี่ของ b ที่มีค่า 5 กับ ค่าความถี่ 10 ของสัญลักษณ์ e จากนั้นทำการลบค่า b: 5 กับ e: 10 ออกจากชุดข้อมูล และแทรกโหนดพ่อแม่ใหม่เข้าไปในชุดข้อมูล พร้อมทั้งกำหนดบิต '0' ให้กับกิ่งซ้าย และกำหนดบิต '1' ให้กับกิ่งขวาของต้นไม้ ดังแสดงในรูปที่ 2.1 (ข) การประมวลผลรอบถัดไปจะทำการรวมความถี่ของ c: 12 กับ โหนดพ่อแม่ที่มีความถี่ 15 เพื่อสร้างโหนดพ่อแม่ใหม่เพราะเป็น 2 ความถี่ที่น้อยที่สุดในชุดข้อมูลซึ่งทำให้ได้ความถี่รวมเท่ากับ 27 แล้วทำการลบสมาชิก c:12 และ

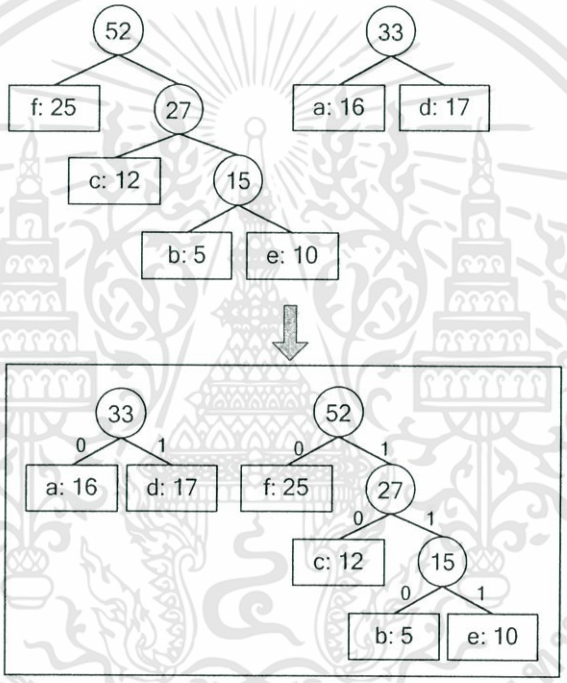
โหนดพ่อแม่ที่มีความถี่ 15 ออกจากชุดข้อมูลและแทรกโหนดพ่อแม่ใหม่เข้าไปพร้อมทั้งกำหนดค่าบิตให้กับแต่ละกิ่งในโครงสร้างต้นไม้ ดังแสดงในรูปที่ 2.1 (ค) ในรอบที่สามทำการรวมค่าความถี่ของ a : 16 กับ d : 17 ซึ่งเป็นค่าความถี่ที่น้อยที่สุดในชุดข้อมูล ซึ่งทำให้ได้โหนดพ่อแม่ใหม่ที่มีค่าความถี่ 33 แล้วนำไปแทรกในชุดข้อมูลพร้อมกับลบ 2 สมาชิกดังกล่าวออกจากชุดข้อมูล (รูปที่ 2.1 (ง)) ในรอบที่ 4 (รูปที่ 2.1 (จ)) ทำการประมวลผลเช่นเดียวกับรอบที่ 2 และ 3 จนกระทั่งเหลือสมาชิกในข้อมูลเพียง 1 สมาชิกคือโหนดรากดังแสดงในรูป 2.1 (ฉ)



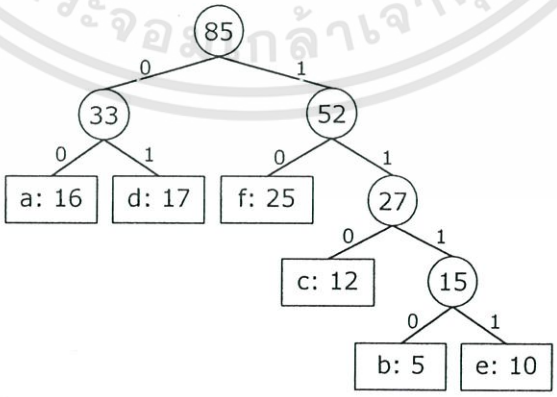
รูปที่ 2.1 ตัวอย่างแสดงขั้นตอนการสร้างต้นไม้ฮัฟแมน



(ง)



(จ)



(ฉ)

รูปที่ 2.1 (ต่อ)

หลังจากได้ต้นไม้ฮัฟแมนที่สมบูรณ์แล้ว จะทำการเข้ารหัสข้อมูลโดยการค้นหาเส้นทางในโครงสร้างต้นไม้จากโหนดรากจนถึงโหนดใบแต่ละโหนด ซึ่งจะได้รหัสบิตที่แทนสัญลักษณ์แต่ละสัญลักษณ์ ดังแสดงในตารางที่ 2.1

ตารางที่ 2.1 แสดงผลของรหัสบิตที่แทนสัญลักษณ์แต่ละสัญลักษณ์ในชุดข้อมูล

สัญลักษณ์	ความถี่ของการเกิดสัญลักษณ์	รหัสบิต
b	5	1110
e	10	1111
c	12	110
a	16	00
d	17	01
f	25	10

2.3 สถาปัตยกรรมคอมพิวเตอร์แบบขนาน

สถาปัตยกรรมคอมพิวเตอร์แบบขนาน (Parallel Computer Architecture) [6][8][11][21][28] เป็นการออกแบบระบบการทำงานที่ทำให้หน่วยประมวลผลหลายๆ หน่วยสามารถประมวลผลไปพร้อมๆ กันได้ เพื่อเพิ่มประสิทธิภาพด้านความเร็ว ความถูกต้องแม่นยำ และลดความซับซ้อนในการประมวลผล ในปี 1972 Flynn [11] ได้จำแนกสถาปัตยกรรมคอมพิวเตอร์ออกเป็น 4 แบบ คือ

1. สถาปัตยกรรมแบบเอสไอเอสดี (SISD: Single Instruction over Single Data) คือ เครื่องคอมพิวเตอร์แบบทั่วไปที่ใช้หน่วยประมวลผลเพียงหนึ่งหน่วยประมวลผลเท่านั้น ซึ่งมีลักษณะการประมวลผลแบบทีละหนึ่งคำสั่งต่อหนึ่งข้อมูล

2. สถาปัตยกรรมแบบเอสไอเอ็มดี (SIMD: Single Instruction over Multiple Data) คือ ระบบคอมพิวเตอร์แบบขนานสำหรับใช้งานเฉพาะด้าน ซึ่งมีลักษณะการประมวลผลแบบทีละหนึ่งคำสั่งต่อหลายข้อมูลด้วยหน่วยประมวลผลหลายหน่วยพร้อมกัน

3. สถาปัตยกรรมแบบเอ็มไอเอสดี (MISD: Multiple Instructions over Single Data) คือ ระบบคอมพิวเตอร์แบบขนานสำหรับใช้งานเฉพาะด้าน ซึ่งมีลักษณะการประมวลผลแบบหลายคำสั่งต่อหนึ่งข้อมูลด้วยหน่วยประมวลผลหลายหน่วยพร้อมกัน

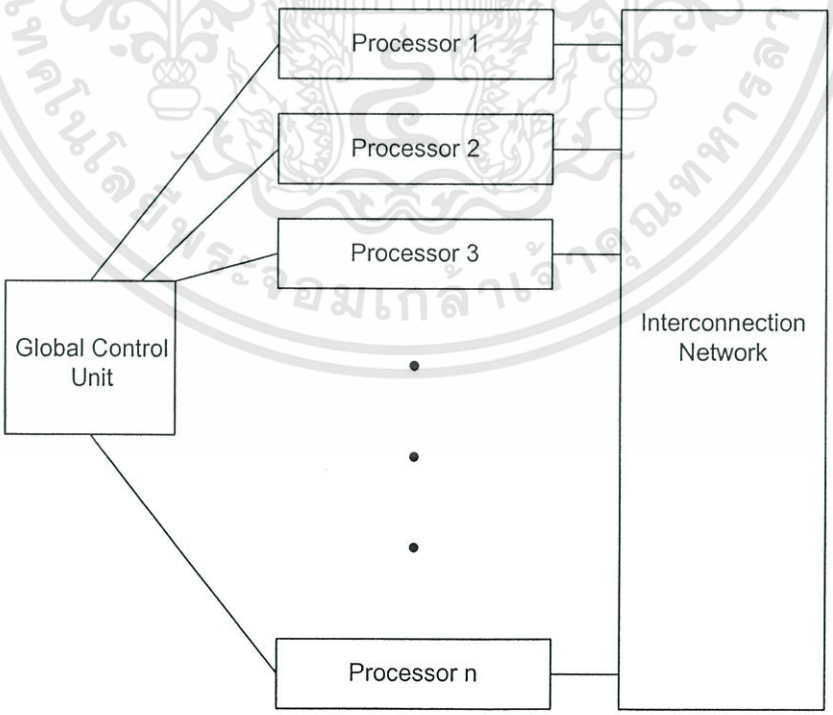
4. สถาปัตยกรรมแบบเอ็มไอเอ็มดี (MIMD: Multiple Instructions over Multiple Data) คือ ระบบคอมพิวเตอร์แบบขนานสำหรับใช้ในงานทั่วๆ ไป ซึ่งมีลักษณะการประมวลผลแบบหลายคำสั่งต่อหลายข้อมูลด้วยหน่วยประมวลผลหลายหน่วยพร้อมกัน ซึ่งได้รับความนิยมอย่างสูงในปัจจุบัน

การสร้างระบบคอมพิวเตอร์แบบขนานสามารถทำได้หลายวิธีซึ่งแบ่งตามรูปแบบวิธีการพื้นฐาน 3 รูปแบบ [21] คือ กลไกการควบคุม (Control Mechanism) องค์ประกอบของพื้นที่การใช้งาน (Address-Space Organization) และเครือข่ายการเชื่อมต่อระหว่างหน่วยประมวลผล (Interconnection Network)

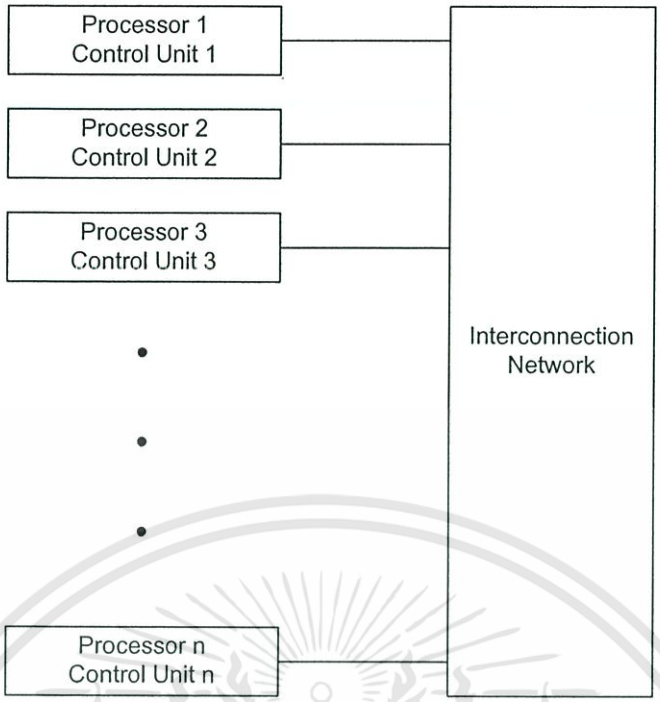
2.3.1 ระบบคอมพิวเตอร์แบบขนานที่แบ่งตามกลไกการควบคุม

แต่ละหน่วยประมวลผลในระบบคอมพิวเตอร์แบบขนานที่แบ่งตามกลไกการควบคุม (Control Mechanism) นี้สามารถทำงานภายใต้การควบคุมของหน่วยควบคุมกลาง (Control Unit) หรือทำงานภายใต้การควบคุมแบบอิสระด้วยหน่วยควบคุมของแต่ละหน่วยประมวลผลเอง ซึ่งสถาปัตยกรรมชนิดแรกเรียกว่าเอสไอเอ็มดี (SIMD : Single Instruction Stream, Multiple Data Stream) ดังแสดงในรูปที่ 2.2 และเอ็มไอเอ็มดี (MIMD : Multiple Instruction Stream, Multiple Data Stream) ดังแสดงในรูปที่ 2.3

กลไกการควบคุมในสถาปัตยกรรมแบบเอสไอเอ็มดีเป็นการยอมให้คำสั่งเดียวกันสามารถถูกประมวลผลด้วยหน่วยประมวลผลทั้งหมดในจังหวะเดียวกัน (Synchronously) แต่มีข้อมูลที่ใช้ประมวลผลต่างกันโดยควบคุมจากหน่วยควบคุมกลาง ดังแสดงในรูปที่ 2.2 สำหรับกลไกการควบคุมในสถาปัตยกรรมแบบเอ็มไอเอ็มดีนั้น แต่ละหน่วยประมวลผลจะมีหน่วยควบคุมรวมอยู่ด้วย ดังแสดงในรูปที่ 2.3 ซึ่งทำให้ระบบนี้สามารถเก็บได้ทั้งระบบปฏิบัติการและโปรแกรมในแต่ละหน่วยประมวลผล



รูปที่ 2.2 โครงสร้างสถาปัตยกรรมแบบเอสไอเอ็มดี



รูปที่ 2.3 โครงสร้างสถาปัตยกรรมแบบเอ็มไอเอ็มดี

2.3.2 ระบบคอมพิวเตอร์แบบขนานที่แบ่งตามองค์ประกอบของพื้นที่ใช้งาน

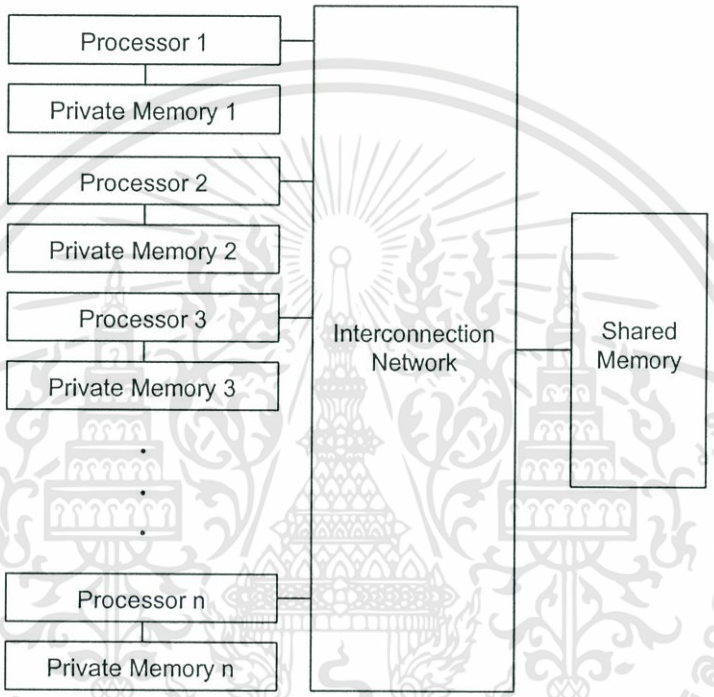
ระบบคอมพิวเตอร์แบบขนานที่แบ่งตามองค์ประกอบของพื้นที่ใช้งาน (Address Space Organization) นี้ หน่วยประมวลผลสามารถติดต่อซึ่งกันและกันด้วยวิธีการใดวิธีการหนึ่งใน 2 วิธี คือ 1) แก้ไขข้อมูลในตำแหน่งพื้นที่ที่ว่างอยู่ (Shared Address Space) หรือ 2) โดยการส่งข้อความถึงกัน (Message Passing)

1) สถาปัตยกรรมในการใช้ที่ว่างร่วมกัน

สถาปัตยกรรมในการใช้ที่ว่างร่วมกัน (Shared Address Space Architecture) เป็นสถาปัตยกรรมคอมพิวเตอร์แบบขนานที่ถูกออกแบบมาให้มีการใช้หน่วยความจำร่วมกัน [21] โดยที่หน่วยประมวลผลทั้งหมดในระบบสามารถเข้าถึงข้อมูลในหน่วยความจำ (Memory) ของระบบได้ทุกที่พร้อมๆ กัน ทำให้การพัฒนาโปรแกรมทำได้สะดวกมากขึ้น อย่างไรก็ตามจำนวนของหน่วยประมวลผลที่สามารถเข้าถึงข้อมูลในหน่วยความจำนั้นได้จะถูกจำกัดด้วยขนาดของเส้นทางเครือข่ายการเชื่อมต่อ จากปัญหาดังกล่าวได้มีความพยายามที่จะแก้ไข โดยกำหนดให้แต่ละหน่วยประมวลผลมีหน่วยความจำส่วนตัว (Local Memory) และมีหน่วยความจำกลาง (Global Memory) ที่ใช้เก็บข้อมูลร่วมกัน ซึ่งโดยทั่วไปแล้วสามารถแบ่งสถาปัตยกรรมแบบใช้หน่วยความจำร่วมกันได้ 3 โมเดล คือ

- **ยูเอ็มเอโมเดล**

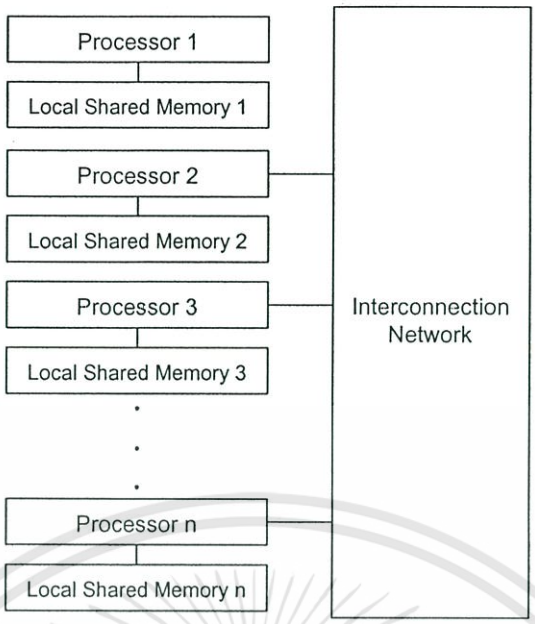
สถาปัตยกรรมแบบที่ใช้หน่วยความจำร่วมกันแบบยูเอ็มเอโมเดล (UMA Model : Uniform Memory Access Model) [8] นี้เป็นโมเดลที่ทุกๆ หน่วยประมวลผลจะมีหน่วยความจำส่วนตัวของตนเองส่วนหนึ่งซึ่งไม่ยอมให้หน่วยประมวลผลอื่นเข้าถึงได้ และสามารถติดต่อหน่วยความจำอื่นที่เป็นหน่วยความจำร่วมซึ่งทุกหน่วยประมวลผลสามารถเข้าถึงหน่วยความจำนี้ได้ในเวลาเดียวกัน และจำนวนของหน่วยประมวลผลไม่จำเป็นต้องเท่ากับจำนวนของหน่วยความจำ ดังรูปที่ 2.4



รูปที่ 2.4 โครงสร้างการใช้หน่วยความจำร่วมกันแบบยูเอ็มเอโมเดล

- **เอ็นยูเอ็มเอโมเดล**

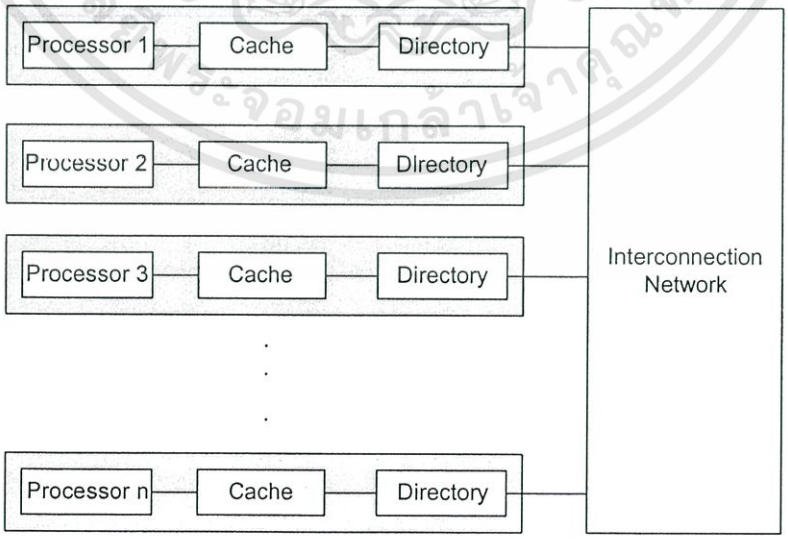
สถาปัตยกรรมแบบที่ใช้หน่วยความจำร่วมกันแบบเอ็นยูเอ็มเอโมเดล (NUMA Model : Non-Uniform Memory Access Model) [8] นี้เป็นโมเดลที่มีหน่วยความจำส่วนตัวเป็นแบบหน่วย-ความจำร่วม (LM : Local Shared Memory) อยู่ในแต่ละหน่วยประมวลผลดังแสดงในรูปที่ 2.5 ซึ่งสถาปัตยกรรมแบบนี้ยอมให้หน่วยประมวลผลอื่นสามารถเข้าถึงได้ โดยที่การเข้าถึงข้อมูลในหน่วยความจำส่วนตัวที่ใช้ร่วมกันของตนเองจะทำได้เร็วกว่าการเข้าถึงข้อมูลในหน่วยความจำส่วนตัวที่ใช้ร่วมกันของหน่วยประมวลผลอื่น



รูปที่ 2.5 โครงสร้างการใช้หน่วยความจำร่วมกันแบบเอ็นยูเอ็มเอโมเดล

- **ซีโอเอ็มเอโมเดล**

สถาปัตยกรรมแบบที่ใช้หน่วยความจำร่วมกันแบบซีโอเอ็มเอโมเดล (COMA Model : Cache Only Memory Access Model) [8] เป็นโมเดลที่มีการนำแคช (Cache) เข้ามาใช้แทนหน่วยความจำ (Memory) ดังแสดงในรูปที่ 2.6 ซึ่งทำให้เข้าถึงข้อมูลได้เร็วขึ้น โมเดลนี้เป็นรูปแบบพิเศษของเครื่องเอ็นยูเอ็มเอ ซึ่งได้มีการกระจายหน่วยความจำหลักไปเปลี่ยนเป็นแบบแคช โดยไม่มีลำดับชั้นของหน่วยความจำในแต่ละหน่วยประมวลผล การเข้าถึงหน่วยความจำนี้จากเครือข่ายอื่นจะเข้าถึงโดยผ่านไดเรกทอรี่แคช ซึ่งมีเครือข่ายการเชื่อมต่อภายในเครือข่ายด้วยการคัดลอกบล๊อกของแคชเป็นลำดับชั้น

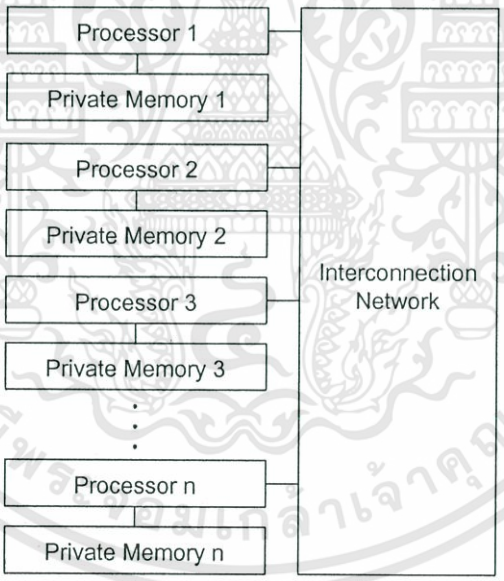


รูปที่ 2.6 โครงสร้างการใช้หน่วยความจำร่วมกันแบบซีโอเอ็มเอโมเดล

นอกจากยูเอ็มเอโมเดล (UMA Model) เอ็นยูเอ็มเอโมเดล (NUMA Model) และ ซีโอเอ็มเอโมเดล (COMA Model) แล้ว ยังมีสถาปัตยกรรมคอมพิวเตอร์แบบอื่นที่ใช้สำหรับหน่วยประมวลผลหลายๆ หน่วย อาทิ ซีซี-เอ็นยูเอ็มเอโมเดล (CC-NUMA Model : Cache-Coherent Non-Uniform Memory Access Model) ซึ่งเป็นโมเดลที่มีการคัดลอกรูปแบบของสถาปัตยกรรมแบบซีโอเอ็มเอโมเดล (COMA Model) มาสร้างบนสถาปัตยกรรมแบบเอ็นยูเอ็มเอโมเดล (NUMA Model) [8]

2) สถาปัตยกรรมการส่งผ่านข้อความ

สถาปัตยกรรมแบบการส่งผ่านข้อความ (Message-Passing Architecture) นี้ ได้ถูกออกแบบมาให้แต่ละหน่วยประมวลผลมีหน่วยความจำส่วนตัวอยู่ภายใน ซึ่งหน่วยความจำนี้จะยอมให้เฉพาะหน่วยประมวลผลของตนเองเท่านั้นเข้าถึงได้ หากหน่วยประมวลผลอื่นต้องการจะติดต่อถึงกันจะต้องอาศัยการส่งผ่านข้อมูล(Message-Passing) หรือถ้าหน่วยประมวลผลใดต้องการใช้หน่วยความจำในหน่วยประมวลผลอื่นจะต้องทำการส่งข้อความร้องขอไปยังหน่วยประมวลผลนั้นๆ เพื่อขอให้หน่วยประมวลผลนั้นส่งข้อมูลมาให้ตามที่ร้องขอ



รูปที่ 2.7 โครงสร้างสถาปัตยกรรมการส่งผ่านข้อความ

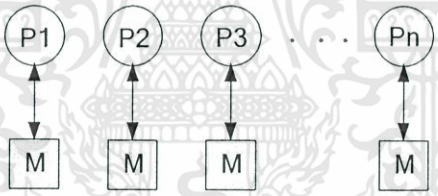
2.3.3 ระบบคอมพิวเตอร์แบบขนานที่แบ่งตามเครือข่ายการเชื่อมต่อระหว่างหน่วยประมวลผล

เครือข่ายการเชื่อมต่อระหว่างหน่วยประมวลผล (Interconnection Network) [21] แบ่งออกเป็น 2 แบบ คือ แบบสแตติก (Static Interconnection) และแบบไดนามิก (Dynamic Interconnection) ซึ่งเครือข่ายการเชื่อมต่อแบบสแตติกจะนิยมใช้ในการสร้างสถาปัตยกรรมคอมพิวเตอร์

แบบส่งผ่านความ (Message-Passing Architecture) ในขณะที่เครือข่ายการเชื่อมต่อแบบไดนามิกจะนำไปใช้ในการสร้างสถาปัตยกรรมคอมพิวเตอร์แบบการใช้หน่วยความจำร่วมกัน (Shared Address Space Architecture)

2.4 พีแรมโมเดล

พีแรมโมเดล (PRAM Models : Parallel Random Access Machine Models) [6][21] เป็นโมเดลของระบบคอมพิวเตอร์แบบขนานตามทฤษฎีหรือระบบคอมพิวเตอร์ในอุดมคติที่ได้รับการยอมรับและนำไปใช้เป็นโมเดลของระบบพื้นฐานเพื่อความสะดวกในการออกแบบขั้นตอนวิธีแบบขนานแบบทั่วไปอย่างกว้างขวาง ซึ่งรูปแบบการเข้าถึงหน่วยความจำในโมเดลนี้จะมีการใช้หน่วยประมวลผลจำนวนมากโดยสามารถเข้าถึงหน่วยความจำร่วมที่มีขนาดใหญ่ได้อย่างไม่มีรูปแบบจำกัด และแต่ละหน่วยประมวลผลสามารถประมวลผลคำสั่งที่แตกต่างกันได้ในแต่ละรอบของการประมวลผล ดังนั้น พีแรมโมเดลจึงเป็นโมเดลระบบคอมพิวเตอร์ตามทฤษฎีหรือระบบคอมพิวเตอร์ในอุดมคติที่ผสมผสานระหว่างระบบคอมพิวเตอร์แบบเอ็มไอเอ็มดีโมเดลและยูเอ็มเอโมเดล ซึ่งโดยปกติแล้วพีแรมโมเดลจะสามารถอ่านหรือเขียนข้อมูลในหน่วยความจำที่ตำแหน่งต่างกันในเวลาเดียวกันได้ดังรูปที่ 2.8

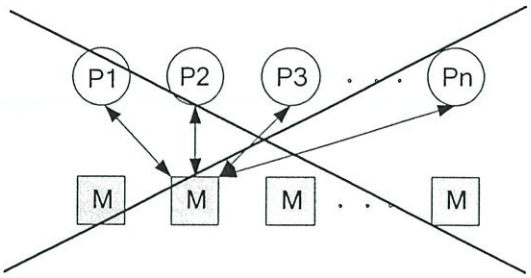


รูปที่ 2.8 การอ่านหรือเขียนในหน่วยความจำที่ตำแหน่งต่างกันในแบบขนานของพีแรมโมเดล

เครื่องคอมพิวเตอร์ที่มีการใช้หน่วยความจำร่วมลักษณะเดียวกันกับพีแรมโมเดลอาจจะยอมให้หน่วยประมวลผลมากกว่าหนึ่งหน่วยสามารถอ่านและเขียนในหน่วยความจำตำแหน่งเดียวกันได้ในเวลาเดียวกัน ซึ่งลักษณะการเข้าถึงหน่วยความจำร่วมของพีแรมโมเดลสามารถแบ่งได้ 4 รูปแบบ ดังนี้

2.4.1 พีแรมโมเดลแบบอีอาร์อีดับเบิลยู

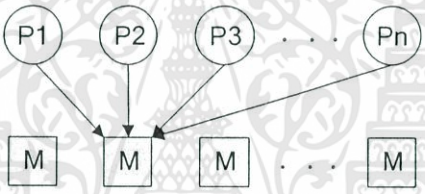
การเข้าถึงหน่วยความจำร่วมแบบขนานของพีแรมโมเดลแบบอีอาร์อีดับเบิลยู (EREW PRAM Model : Exclusive-Read, Exclusive-Write PRAM Model) ไม่ยอมให้หน่วยประมวลผลหลายหน่วยอ่านหรือเขียนข้อมูลในหน่วยความจำเดียวกันในเวลาเดียวกันได้ นั่นคือในเวลาหนึ่งๆ จะมีเพียงหนึ่งหน่วยประมวลผลเท่านั้นที่สามารถเข้าถึงหน่วยความจำหนึ่งได้ ดังรูปที่ 2.9



รูปที่ 2.9 พีแรมโมเดลแบบอีอาร์อีดับเบิลยู

2.4.2 พีแรมโมเดลแบบอีอาร์ซีดับเบิลยู

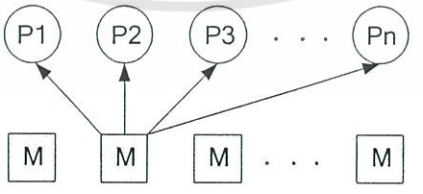
การเข้าถึงหน่วยความจำร่วมแบบขนานของพีแรมโมเดลแบบอีอาร์ซีดับเบิลยู (ERCW PRAM Model : Exclusive-Read, Concurrent-Write PRAM Mode) ยอมให้หน่วยประมวลผลเดียวเท่านั้นที่เข้าถึงหน่วยความจำร่วมเพื่ออ่านข้อมูล และยอมให้หน่วยประมวลผลหลายหน่วยเข้าถึงหน่วยความจำร่วมเพื่อเขียนได้ในเวลาเดียวกันได้ ดังแสดงในรูปที่ 2.10



รูปที่ 2.10 พีแรมโมเดลแบบอีอาร์ซีดับเบิลยู

2.4.3 พีแรมโมเดลแบบซีอาร์อีดับเบิลยู

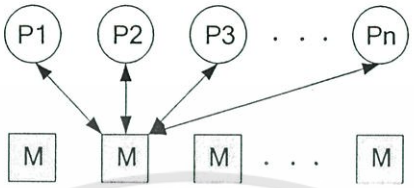
การเข้าถึงหน่วยความจำร่วมแบบขนานของพีแรมโมเดลแบบซีอาร์อีดับเบิลยู (ERCW PRAM Model: Concurrent-Read, Exclusive-Write PRAM Model) ยอมให้หน่วยประมวลผลหลายหน่วยสามารถเข้าถึงหน่วยความจำร่วมเพื่ออ่านข้อมูลพร้อมกันได้ แต่จะยอมให้เพียงหนึ่งหน่วยประมวลผลเท่านั้นที่เข้าถึงหน่วยความจำร่วมเพื่อเขียน ดังแสดงในรูปที่ 2.11



รูปที่ 2.11 พีแรมโมเดลแบบซีอาร์อีดับเบิลยู

2.4.4 พีแรมโมเดลแบบซีอาร์ซีดับเบิลยูพี

การเข้าถึงหน่วยความจำร่วมแบบขนานของพีแรมโมเดลแบบซีอาร์ซีดับเบิลยู (CREW PRAM Model : Concurrent-Read,Concurrent-Write PRAM Model) ยอมให้มีหน่วยประมวลผลหลายหน่วยสามารถเข้าถึงหน่วยความจำร่วมเพื่ออ่านหรือเขียนข้อมูลพร้อมกันได้ ดังแสดงในรูปที่ 2.12 ซึ่งวิธีนี้จะช่วยให้สามารถประมวลผลข้อมูลได้รวดเร็วมากกว่ารูปแบบอื่น



รูปที่ 2.12 พีแรมโมเดลแบบซีอาร์ซีดับเบิลยู

ด้วยคุณลักษณะต่างๆ ของการประมวลผลแบบขนานจึงได้ถูกนำไปประยุกต์ใช้ในงานด้านต่างๆ เพื่อเพิ่มความเร็วในการประมวลผล อาทิ การลดความซับซ้อนด้านเวลาสำหรับการค้นหา การรวมข้อมูลและเรียงลำดับข้อมูลในแบบขนาน [13] การสร้างโครงสร้างต้นไม้บนการประมวลผลแบบขนาน [2][12] การนำวิธีการประมวลผลแบบขนานมาประยุกต์ในการเข้ารหัสของฮัฟแมน[18][19] และการถอดรหัสของฮัฟแมน [14] เพื่อลดความซับซ้อนด้านเวลา

2.5 งานวิจัยที่เกี่ยวข้องกับการเข้ารหัสข้อมูลของฮัฟแมนแบบขนาน

ขั้นตอนวิธีการเข้ารหัสของฮัฟแมนได้ถูกนำมาพัฒนาให้สามารถประมวลผลด้วยกระบวนการแบบขนาน [18][19] เพื่อลดเวลาในการประมวลผล โดยชุดรหัสบิตที่ได้หลังจากการประมวลผลจะมีลักษณะเหมือนกับชุดรหัสบิตที่ได้จากการเข้ารหัสของฮัฟแมนแบบดั้งเดิม

2.5.1 ขั้นตอนวิธีอีเอส-พาร์ฮัฟ

ในปี ค.ศ.1999 Milidiu, Laber และ Pessoa [18] ได้คิดค้นขั้นตอนวิธีการสร้างต้นไม้ฮัฟแมนบนกระบวนการประมวลผลแบบขนานเพื่อลดเวลาในการประมวลผล โดยขั้นตอนวิธีการนี้มีชื่อว่าขั้นตอนวิธีอีเอส-พาร์ฮัฟ (ES-ParHuff Algorithm) ซึ่งมีความซับซ้อนด้านเวลาในกรณีที่ดีที่สุดหรือในกรณีที่ต้นไม้ฮัฟแมนมีการกระจายกิ่งในลักษณะที่ใกล้สมดุลเท่ากับ O(log(1/p1) log(log n)) เมื่อ p1 คือผลหารของความถี่ของสัญลักษณ์แรกกับความถี่ทั้งหมดของทุกสัญลักษณ์ในข้อมูลและได้ความซับซ้อนด้านเวลาเป็น O(n) ในกรณีที่ต้นไม้ฮัฟแมนมีการกระจายกิ่งไปในทิศทางเดียว โดยขั้นตอนวิธีของอีเอส-พาร์ฮัฟ แสดงดังขั้นตอนวิธีที่ 2.1

ขั้นตอนวิธีที่ 2.1 ขั้นตอนวิธีอีเอส-พาร์ฮัฟ

```
S ← sorted list of leaves;
Q ← nil;
While length(S) > 0 or length(Q) > 1
    Select the two nodes a and b with smallest weights in S or Q;
    Remove a and b from their corresponding lists;
    Create t1 as a parent of a and b; w(t1) ← w(a)+w(b);
    k ← Select(S, w(t1))
    If k+length(Q) is even then U ← Merge(S^k,Q);
    Else
        If w(S^k) ≤ w(q_length(Q)) then U ← Merge(S^k,Q^length(Q)-1);
        Else U ← Merge(S^k-1,Q);
    End If
    Insert t1 in the queue Q;
    For i=1 to length(U)/2 pardo
        Create ti as a parent of u2i-1 and u2i;
        w(ti) ← w(u2i-1)+w(u2i);
        Insert ti in the queue Q
    End For
End While
```

ขั้นตอนการทำงานของอีเอส-พาร์ฮัฟ สามารถอธิบายได้ดังนี้

ขั้นตอนที่ 1 นำความถี่ของสัญลักษณ์ทั้งหมดในข้อมูลเก็บไว้ในชุดรายการ S (ชุดรายการสำหรับเก็บค่าความถี่ของสัญลักษณ์) และกำหนดค่าเริ่มต้นให้ชุดรายการ Q (ชุดรายการสำหรับเก็บค่าน้ำหนักของโหนดพ่อแม่) เป็น nil

ขั้นตอนที่ 2 เลือกค่าความถี่ที่น้อยที่สุดในชุดรายการ S หรือชุดรายการ Q มาเก็บไว้ในตัวแปร a และ b ตามลำดับ

ขั้นตอนที่ 3 รวมค่าในตัวแปร a และ b เพื่อสร้างโหนดพ่อแม่ใหม่ (t_1)

ขั้นตอนที่ 4 นำค่าน้ำหนักของ t_1 กับชุดรายการ S เข้าฟังก์ชัน Select เพื่อเลือกสมาชิกในรายการ S ที่มีค่าน้อยกว่าหรือเท่ากับค่า t_1 แล้วนำค่าของจำนวนสมาชิกที่เลือกจากชุดรายการ S เก็บไว้ในตัวแปร k

ขั้นตอนที่ 5 ตรวจสอบเงื่อนไข

ถ้านำค่า k บวกด้วยจำนวนสมาชิกในชุดรายการ Q แล้วได้ผลลัพธ์เป็นเลขคู่

จริง : ให้รวมสมาชิกในชุดรายการ S จำนวน k สมาชิกกับชุดรายการ Q เก็บไว้ในชุดรายการ U

ไม่จริง : ให้ทำการตรวจสอบเงื่อนไข โดยถ้าค่าน้ำหนักของสมาชิกตัวที่ k น้อยกว่าหรือเท่ากับสมาชิกตัวสุดท้ายของชุดรายการ Q

จริง : ให้ทำการรวมสมาชิกในชุดรายการ S จำนวน k สมาชิกกับสมาชิกในชุดรายการ Q จำนวน length(Q)-1 สมาชิกไว้ในชุดรายการ U

ไม่จริง : ให้รวมสมาชิกในชุดรายการ S จำนวน k-1 สมาชิกกับสมาชิกในชุดรายการ Q ทั้งหมดไว้ในชุดรายการ U

ซึ่งการรวมชุดรายการในขั้นตอนนี้จะใช้ขั้นตอนวิธีการรวมข้อมูลแบบขนานของ Kruskal [13] ซึ่งมีความซับซ้อนด้านเวลาเท่ากับ O(log(log n))

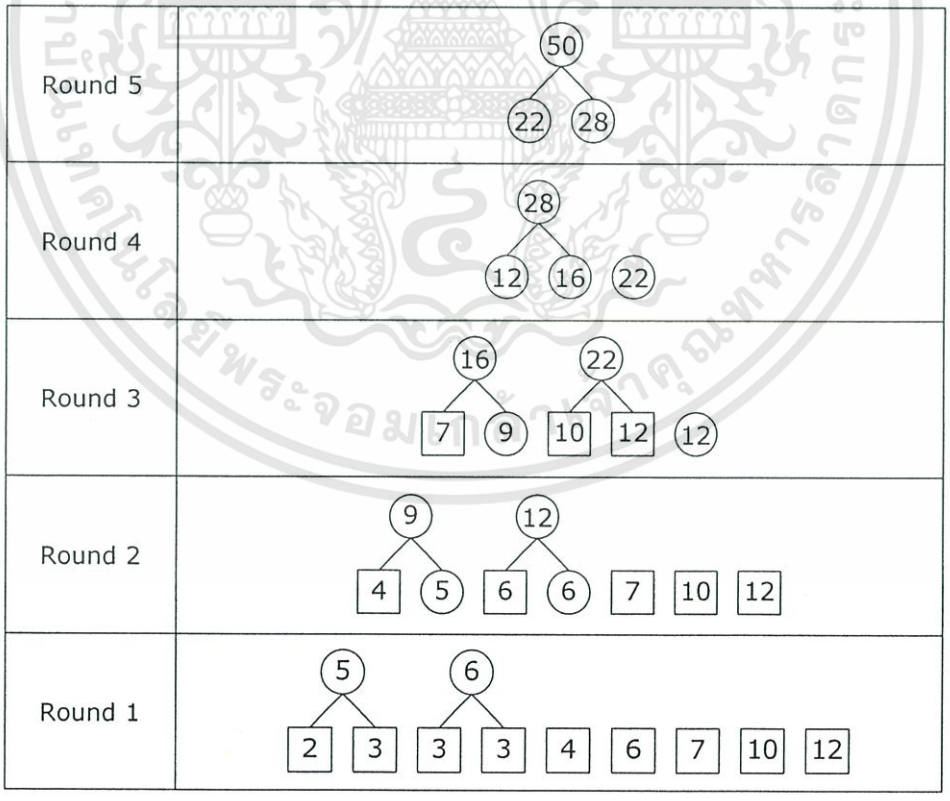
ขั้นตอนที่ 6 นำโหนดพ่อแม่ t_1 เก็บไว้ในชุดรายการ Q

ขั้นตอนที่ 7 ทำการจับคู่สมาชิกในชุดรายการ U เพื่อสร้างโหนดพ่อแม่ใหม่ แล้วนำโหนดพ่อแม่ที่ได้เก็บไว้ในชุดรายการ Q

ทำซ้ำในขั้นตอนที่ 2 ถึง 7 จนกระทั่งความยาวของรายการ S น้อยกว่าเท่ากับ 0 หรือความยาวของรายการ Q น้อยกว่าเท่ากับ 1

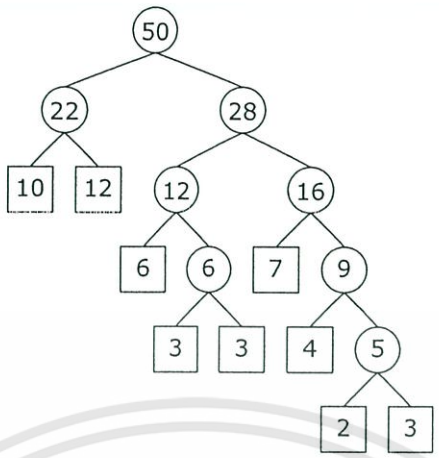
ตัวอย่างที่ 1 ตัวอย่างการสร้างต้นไม้ฮัฟแมนด้วยขั้นตอนวิธีอีเอส-พาร์ฮัฟ

สมมุติชุดข้อมูลประกอบด้วยสัญลักษณ์ S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8 และ S_9 โดยมีความถี่การปรากฏสัญลักษณ์แต่ละสัญลักษณ์คือ 2, 3, 3, 3, 4, 6, 7, 10 และ 12 ตามลำดับ ซึ่งจะต้องประมวลผลทั้งหมด 5 รอบในรูปแบบจากล่างขึ้นบน (Bottom-Up) ดังรูปที่ 2.13



รูปที่ 2.13 แสดงตัวอย่างในแต่ละรอบของการสร้างต้นไม้ฮัฟแมนด้วยขั้นตอนวิธีอีเอส-พาร์ฮัฟ

เมื่อนำโครงสร้างของต้นไม้ในแต่ละรอบมาประกอบกันจะได้ต้นไม้ฮัฟแมน ดังรูปที่ 2.14



รูปที่ 2.14 แสดงต้นไม้ฮัฟแมนซึ่งเป็นผลลัพธ์จากการประมวลผลด้วยขั้นตอนวิธีอีเอส-พาร์ฮัฟ

ข้อดีและข้อจำกัดของขั้นตอนวิธีอีเอส-พาร์ฮัฟ เมื่อเทียบกับวิธีการดั้งเดิม มีดังนี้

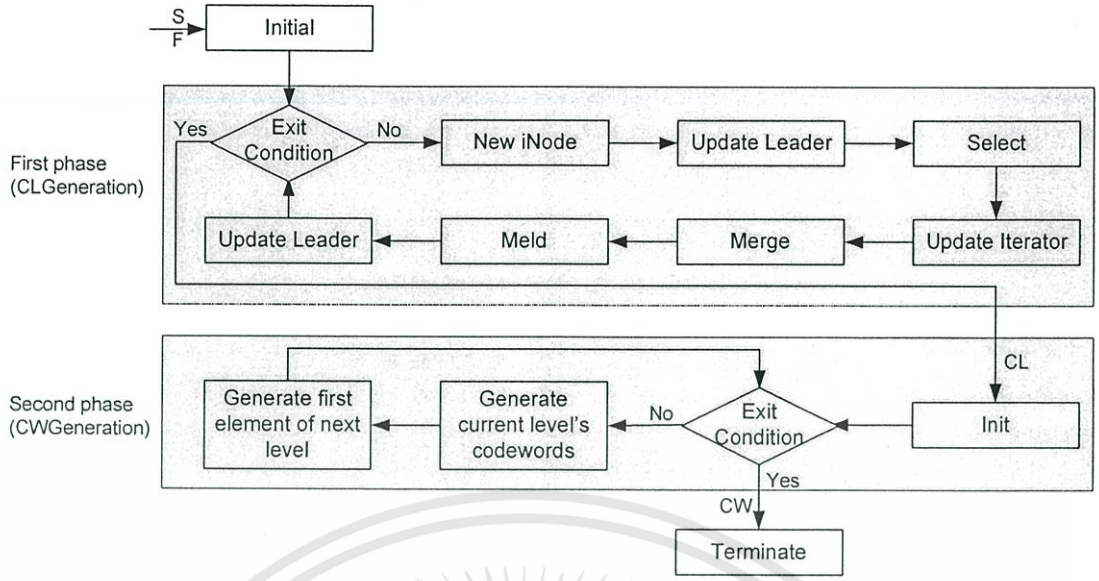
ข้อดี	ใช้เวลาในการสร้างต้นไม้ฮัฟแมนน้อยกว่าวิธีการดั้งเดิม
ข้อจำกัด	ส่วนการเข้ารหัสได้ใช้วิธีการสร้างรหัสบิตด้วยวิธีการเข้ารหัสของฮัฟแมนแบบดั้งเดิม ทำให้ไม่ได้ลดเวลาในการประมวลผลในส่วนการสร้างรหัสบิต

2.5.2 ขั้นตอนวิธีของอราส

ขั้นตอนวิธีของอราส ซึ่งมีชื่อเรียกว่า "Two-phasePractical Parallel Algorithm" ได้ถูกพัฒนาขึ้นโดย Ostadzadeh และคณะ [19] ในปี .ศ.2006 เพื่อแก้ไขข้อจำกัดของวิธีอีเอส-พาร์ฮัฟ โดยการนำหลักการสร้างต้นไม้ฮัฟแมนในแบบขนานของ Milidiu และคณะมาพัฒนาเพื่อให้สามารถสร้างรหัสบิตของฮัฟแมนได้ โดยใช้โครงสร้างอะเรย์เก็บข้อมูลโหนดในต้นไม้ฮัฟแมนแทนการสร้างต้นไม้ฮัฟแมน ซึ่งมีความซับซ้อนด้านเวลาในกรณีที่ดีที่สุดเท่ากับ O(log((log(n-1))!))

โดยขั้นตอนวิธีนี้ได้แบ่งขั้นตอนในการทำงานออกเป็น 2 ส่วนดังรูปที่ 2.15 ดังนี้

- ส่วนคำนวณความยาวของรหัสบิต (CLGeneration : Code Length Generation)
- ส่วนการสร้างชุดรหัสบิต (CWGeneration : Codeword Generation)



S : สัญลักษณ์ในชุดข้อมูลนำเข้า
F : ความถี่ของแต่ละสัญลักษณ์ในชุดข้อมูล
CL : ความยาวของรหัสบิตของแต่ละสัญลักษณ์
CW : รหัสบิตของแต่ละสัญลักษณ์

รูปที่ 2.15 โครงสร้างการทำงานของขั้นตอนวิธีของอราส

ส่วนที่ 1 ส่วนคำนวณหาความยาวของชุดรหัสบิต

ขั้นตอนนี้ได้ใช้อะเรย์ดังต่อไปนี้

lNodes	เป็นอะเรย์สำหรับเก็บข้อมูลเกี่ยวกับโหนดใบซึ่งประกอบด้วยขอบเขต freq และ leader
iNodes	เป็นอะเรย์สำหรับเก็บข้อมูลเกี่ยวกับโหนดพ่อแม่หรือโหนดภายใน (Internal Node) ซึ่งมีขอบเขตเช่นเดียวกับอะเรย์ lNodes
CL	เป็นอะเรย์สำหรับเก็บความยาวของแต่ละสัญลักษณ์
Copy	เป็นอะเรย์สำหรับเก็บข้อมูลของโหนดใบเฉพาะที่มีความถี่น้อยกว่าหรือเท่ากับค่าในตัวแปร MinFreq ซึ่งตัวแปร MinFreq จะเก็บค่าความถี่ของโหนดภายในโหนดแรกที่สร้างขึ้นในแต่ละรอบการประมวลผล ซึ่งประกอบด้วยขอบเขต freq, isLeaf และ index
Temp	เป็นอะเรย์สำหรับเก็บข้อมูลที่คัดลอกมาจากอะเรย์ Copy โดยจำนวนสมาชิกในอะเรย์ Temp จะเป็นจำนวนคู่ เพื่อจะนำไปสร้างโหนดพ่อแม่ใหม่พร้อมกัน
mid	เป็นอะเรย์ที่ประกอบด้วย 2 ความถี่ที่น้อยที่สุดของอะเรย์ lNodes และ 2 ความถี่ที่น้อยที่สุดของอะเรย์ iNodes

ขั้นตอนการคำนวณความยาวของชุดรหัสบิต (CLGeneration) ประกอบด้วย 6 ขั้นตอนย่อย ดังนี้

ขั้นตอนที่ 1 กำหนดค่าเริ่มต้นให้กับตัวแปรและอะเรย์ที่ใช้ในการคำนวณ

ขั้นตอนที่ 2 สร้างโหนดพ่อแม่ใหม่แรก

ขั้นตอนที่ 3 เลือกสมาชิกสำหรับสร้างโหนดพ่อแม่ใหม่ในระดับเดียวกัน

ขั้นตอนที่ 4 สร้างกลุ่มสมาชิกใหม่ของอะเรย์ Temp

ขั้นตอนที่ 5 สร้างโหนดพ่อแม่ใหม่แบบขนาน

ขั้นตอนที่ 6 ปรับค่าความยาวของรหัสบิตสำหรับแต่ละสัญลักษณ์

ซึ่งแต่ละขั้นตอนมีรายละเอียดดังต่อไปนี้

1. **กำหนดค่าเริ่มต้น** ซึ่งขั้นตอนนี้จะกำหนดค่าเริ่มต้นให้กับตัวแปรและอะเรย์ที่ใช้ในการคำนวณความยาวของรหัสบิตโดยจะประมวลผลแบบขนานด้วยจำนวนหน่วยประมวลผลที่เท่ากับจำนวนสัญลักษณ์ในข้อมูลนำเข้า ดังแสดงในขั้นตอนวิธีที่ 2.2

ขั้นตอนวิธีที่ 2.2 ขั้นตอนวิธีในส่วนกำหนดค่าเริ่มต้น

```
Forall Processors P_i(1 ≤ i ≤ n) pardo
    lNodes[i].freq ← F[i]
    lNodes[i].leader ← -1
    CL[i] ← 0
End Forall

P_1 sets
    iNodesFront ← 0
    iNodesRear ← 0
    lNodesCur ← 0
```

2. **สร้างโหนดพ่อแม่ใหม่แรก** ซึ่งเป็นขั้นตอนสำหรับสร้างโหนดพ่อแม่ใหม่ซึ่งเป็นโหนดแรกในรอบการประมวลผลนั้นๆ โดยใช้หน่วยประมวลผล 1 หน่วยในการประมวลผล ดังแสดงในขั้นตอนวิธีที่ 2.3 โดยเริ่มจากการนำค่าที่น้อยที่สุด 2 ค่าจากอะเรย์ lNodes และ iNodes มาเก็บไว้ในอะเรย์ mid แล้วทำการเลือกค่าที่น้อยที่สุด 2 ค่าจากอะเรย์ mid มารวมกันเพื่อสร้างเป็นโหนดพ่อแม่ใหม่ พร้อมทั้งเก็บผลรวมที่ได้ไว้ในอะเรย์ iNodes.freq และปรับค่า lNodes.leader ของสมาชิกที่ได้นำมารวมกันเพิ่มขึ้นเพื่อบ่งบอกว่าโหนดใบนั้นๆ มีลำดับของโหนดภายในที่เป็นพ่อแม่กี่ลำดับ จากนั้นทำการเพิ่มค่าความยาวของสมาชิกที่เป็นโหนดใบที่ได้นำมารวมกันในอะเรย์ CL

ขั้นตอนวิธีที่ 2.3 ขั้นตอนวิธีในส่วนการสร้างโหนดพ่อแม่ใหม่แรก

```
P1 sets
    mid ← {∞, ∞, ∞, ∞}
    If (lNodesCur≤n-1)
        mid[1] ← lNodes[lNodesCur+1].freq
    If (lNodesCur≤n-2)
        mid[2] ← lNodes[lNodesCur+2].freq
    If (iNodesRear>iNodesFront)
        mid[3] ← iNodes[iNodesFront+1].freq
    If (iNodesRear>iNodesFront+1)
        mid[4] ← iNodes[iNodesFront+2].freq
    SelectMinimums(mid)
    MinFreq ← mid[1]+mid[2]
    iNodes[iNodesRear+1].freq ← MinFreq
    iNodes[iNodesRear+1].leader ← -1
    If (isLeaf(mid[1]))
        lNodes[lNodesCur+1].leader ← iNodesRear+1
        CL[lNodesCur+1]++, lNodesCur++
    Else
        iNodes[iNodesFront+1].leader ← iNodesRear+1
        iNodesFront++
    End If
    If (isLeaf(mid[2]))
        lNodes[lNodesCur+1].leader ← iNodesRear+1
        CL[lNodesCur+1]++, lNodesCur++
    Else
        iNodes[iNodesFront+1].leader ← iNodesRear+1
    End If
```

3. เลือกสมาชิก โดยเลือกสมาชิกที่มีค่าใน lNodes.freq ที่น้อยกว่าหรือเท่ากับความถี่ของโหนดพ่อแม่ใหม่ โดยสมาชิกที่ตรงตามเงื่อนไขจะถูกคัดลอกไว้ในอะเรย์ Copy

ขั้นตอนวิธีที่ 2.4 ขั้นตอนวิธีในส่วนการเลือกสมาชิก

```
Forall Processors Pi(lNodesCur<i≤n) pardo
    If (lNodes[i].freq≤MinFreq)
        Copy[i-lNodesCur].freq ← lNodes[i].freq
        Copy[i-lNodesCur].index ← i
        Copy[i-lNodesCur].isLeaf ← true
        If (i=n || lNode[i+1].freq>MinFreq)
            CurLeavesNum ← i-lNodesCur
        End If
    End If
End Forall
```

4. สร้างกลุ่มสมาชิกใหม่ของอะเรย์ Temp โดยรวมสมาชิกของอะเรย์ Copy และ iNodes แล้วเก็บไว้ในอะเรย์ Temp โดยใช้ขั้นตอนวิธีการรวมชุดข้อมูลของ Kruskal [13] ดังแสดงในขั้นตอนวิธีที่ 2.5 โดยจำนวนสมาชิกในอะเรย์ Temp ที่ได้จะเป็นเลขคู่เพื่อนำไปจับคู่สร้างโหนดพ่อแม่ใหม่ในขั้นตอนถัดไป

ขั้นตอนวิธีที่ 2.5 ขั้นตอนวิธีในส่วนการสร้างกลุ่มสมาชิกใหม่ของอะเรย์ Temp

```
P1 sets
    mergeRear ← iNodesRear
    mergeFront ← iNodesFront

    If ((CurLeavesNum+iNodesRear-iNodesFront)% 2=0)
        iNodesFront ← iNodesRear
    Else
        If ((iNodesRear-iNodesFront!=0)&&
          (F[lNodesCur+CurLeavesNum]iNodes[iNodesRear].freq))
            mergeRear--
            iNodesFront ← iNodesRear-1
        Else
            iNodesFront ← iNodesRear
            CurLeavesNum--
        End If
    End If
    lNodesCur ← lNodesCur+ CurLeavesNum
    iNodesRear++
```

5. สร้างโหนดพ่อแม่ใหม่แบบขนาน โดยการรวมความถี่ของคู่สมาชิกในอะเรย์ Temp เพื่อสร้างโหนดพ่อแม่ใหม่ด้วยการประมวลผลแบบขนานด้วยจำนวนหน่วยประมวลผล n หน่วย

ขั้นตอนวิธีที่ 2.6 ขั้นตอนวิธีในส่วนการสร้างโหนดพ่อแม่ใหม่แบบขนาน

```
Forall Processors Pi(lNodesCur<i≤n) pardo
    ind ← iNodesRear+i
    iNodes[ind].freq ← temp[2*i-1].freq+temp[2*i].freq
    iNodes[ind].leader ← -1

    If (temp[2*i-1].isleaf)
        lNodes[temp[2*i-1].index].leader ← ind
        CL[temp[2*i-1].index]++
    Else
        iNodes[temp[2*i-1].index].leader ← ind

    End If
    If (temp[2*i].isleaf)
        lNodes[temp[2*i].index].leader ← ind
        CL[temp[2*i].index]++
    Else
        iNodes[temp[2*i].index].leader ← ind
    End If
End Forall
P1 sets
    iNodesRear ← iNodesRear+(TempLength/2)
```

6. ปรับค่าความยาวของรหัสบิตสำหรับแต่ละสัญลักษณ์ เป็นการเพิ่มค่าความยาวของรหัสบิตสำหรับสมาชิกที่แทนโหนดใบ โดยค่าความยาวจะถูกเพิ่มขึ้นตามลำดับของโหนดภายในที่อยู่เหนือโหนดใบนั้นๆ

ขั้นตอนวิธีที่ 2.7 ขั้นตอนวิธีในส่วนการปรับค่าความยาวของรหัสบิตสำหรับแต่ละสัญลักษณ์

```
Forall Processors Pi(lNodesCur<i≤n) pardo
   If (lNodes[i].leader != -1)
      If (iNodes[lNodes[i].leader].leader != -1)
         lNodes[i].leader ← iNodes[lNodes[i].leader].leader
         CL[i]++
      End If
   End If
End Forall
```

<u>ส่วนที่ 2</u> **ส่วนการสร้างชุดรหัสบิต**

ขั้นตอนนี้เป็นการนำค่าความยาว (CWGeneration) ของแต่ละสัญลักษณ์ซึ่งเก็บไว้ในอะเรย์ CL มาคำนวณเพื่อเปลี่ยนค่าความยาวที่ได้ให้เป็นรหัสบิตซึ่งจะนำมาใช้แทนสัญลักษณ์แต่ละสัญลักษณ์ในข้อมูล ดังแสดงในขั้นตอนวิธีที่ 2.8

ขั้นตอนวิธีที่ 2.8 ขั้นตอนวิธีในส่วนการสร้างรหัสบิต

```
Forall Processors Pi(1≤i≤n/2) pardo
    LocalVariable ← CL[i]
    CL[n-i+1] ← LocalVariable
End Forall
P1 sets
    CCL← CL[1]
    CW[1] ← bit string of CCL zeros
    CDPI ← 1
While (CDPI<n)
    Forall processors Pi(1≤i≤n) pardo
        If (i>CDPI && CL[i]−CCL)
            CW[i]←CW[CDPI]+(i-CDPI)
        If (i<n && CL[i+1]!=CCL)
            CDPI ← i
        Else be idle
        End If
    End Forall
End While
P1 sets
    CLDiff ← CL[CDPI+1]-CL[CDPI]
    CW[CDPI+1] ←(CW[CDPI]+1)*2^(CLDiff)
    CCL ← CL[CDPI+1]
    CDPI ← CDPI +1
Forall Processors Pi(1≤i≤n/2) pardo
    LocalVariable ← CW[i]
    CW[n-i+1] ← LocalVariable
End Forall
```

1. สร้างรหัสบิตให้กับค่าความยาวที่น้อยที่สุด โดยกำหนดให้เป็นบิต '0' ทั้งหมด
2. เปรียบเทียบค่าความยาวของสัญลักษณ์ถัดไปเพื่อหาค่าความต่างของความยาวระหว่างชุดรหัสบิตแรกกับชุดรหัสบิตถัดไป
3. นำค่าความต่างที่ได้มาคำนวณเพื่อสร้างชุดรหัสบิตแรกของกลุ่มสมาชิกที่มีความยาวเท่ากัน โดยคำนวณดังนี้

   รหัสบิตของสมาชิกปัจจุบัน = (รหัสบิตของสมาชิกก่อนหน้า+1)*$2^{\text{ค่าความต่างของความยาว}}$
4. สร้างรหัสบิตที่มีความยาวเท่ากันโดยเพิ่มค่าของรหัสบิตนั้นๆ ขึ้น 1 ค่าจากรหัสบิตก่อนหน้าด้วยการประมวลผลแบบขนาน
5. ทำซ้ำในขั้นตอนที่ 2 ถึง 4 จนกระทั่งแปลงค่าความยาวเป็นรหัสบิตครบทุกสมาชิก

ตัวอย่างที่ 2 ตัวอย่างการสร้างต้นไม้ฮัฟแมนด้วยขั้นตอนวิธีของอราส

สมมุติข้อมูลประกอบด้วยสัญลักษณ์ $S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}, S_{11}$ โดยมีความถี่ของการปรากฏสัญลักษณ์ในชุดข้อมูลตามลำดับดังนี้ 9, 9, 11, 11, 13, 13, 13, 16, 18, 20, 38

ส่วนที่ 1 ขั้นตอนเตรียมข้อมูลเบื้องต้น

lNodes.freq	9	9	11	11	13	13	13	16	18	20	38
lNodes.leader	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

CL	0	0	0	0	0	0	0	0	0	0	0

ขั้นตอนการคำนวณความยาวของชุดรหัสบิต (CL Generation)

การประมวลผลในส่วนการคำนวณความยาวของรหัสบิตสำหรับข้อมูลข้างต้นจะประมวลผลทั้งหมด 4 รอบดังนี้

รอบที่ 1

1. เลือกสมาชิกที่น้อยที่สุด 2 ค่าจากอะเรย์ lNodes ซึ่งในที่นี้คือ 9 และ 9 เก็บไว้ในอะเรย์ mid แต่สำหรับในรอบนี้ไม่มีสมาชิกใดในอะเรย์ iNodes ดังนั้นจึงได้ค่าในอะเรย์ mid 2 สมาชิก

2. รวมค่าของสมาชิกที่น้อยที่สุด 2 ค่าในอะเรย์ mid ซึ่งในที่นี้คือ 9 และ 9 รวมกันได้ผลลัพธ์คือ 18 เก็บไว้ในตัวแปร MinFreq

lNodes.freq	9	9	11	11	13	13	13	16	18	20	38
lNodes.leader	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

mid	9	9	-	-

MinFreq = 18

3. นำค่า MinFreq ไปเปรียบเทียบกับค่าใน INodes.freq ที่เหลือ ว่ามีสมาชิกใดที่มีค่าน้อยกว่าหรือเท่ากับค่าของ MinFreq (= 18) ซึ่งในที่นี้ได้สมาชิกที่มีความถี่ 11 ถึง 18 จากนั้นทำการคัดลอกค่าดังกล่าวไว้ในอะเรย์ Copy.freq พร้อมทั้งกำหนดค่าใน Copy.isLeaf ให้เป็น 1 เนื่องจากสมาชิกดังกล่าวเป็นโหนดใบ และกำหนดลำดับของสมาชิกใน Copy.index ดังนี้

Copy.freq	11	11	13	13	13	16	18
Copy.isLeaf	1	1	1	1	1	1	1
Copy.index	1	2	3	4	5	6	7

4. คัดลอกสมาชิกในอะเรย์ Copy ให้ได้จำนวนคู่ เก็บไว้ในอะเรย์ Temp เพื่อนำมาจับคู่สร้างโหนดพ่อแม่ใหม่ ดังรูป

Temp	11	11	13	13	13	16
Temp.isLeaf	1	1	1	1	1	1
Temp.index	1	2	3	4	4	6

22 26 29

เก็บข้อมูลของโหนดพ่อแม่ใหม่ไว้ในอะเรย์ iNodes และทำการปรับค่า INodes.leader ให้ตรงกับลำดับการสร้างโหนดพ่อแม่ใหม่ ซึ่งได้ผลดังนี้

iNodes.freq	18	22	26	29
iNodes.leader	-1	-1	-1	-1

INodes.freq	9	9	11	11	13	13	13	16	18	20	38
INodes.leader	1	1	2	2	3	3	4	4	-1	-1	-1

5. ปรับค่าในอะเรย์ CL เฉพาะสมาชิกช่องที่ได้นำมาสร้างโหนดพ่อแม่ใหม่ให้มีความยาวเท่ากับ 1

CL	1	1	1	1	1	1	1	1	0	0	0

<u>รอบที่ 2</u>

1. เนื่องจากค่าตั้งแต่ 9 ถึง 16 ในอะเรย์ INodes ได้ถูกนำไปสร้างโหนดพ่อแม่ในรอบที่ 1 แล้ว ดังนั้นในรอบที่ 2 นี้จึงนำค่า 18 และ 20 ในอะเรย์ INodes และ ค่า 18 และ 22 ในอะเรย์ iNodes ซึ่งเป็นค่าที่น้อยที่สุดใน 2 กลุ่มดังกล่าวมาเก็บไว้ในอะเรย์ mid

2. เลือกค่าที่น้อยที่สุด 2 ค่าในอะเรย์ mid มารวมกันเก็บไว้ในตัวแปร MinFreq ซึ่งได้ 36

lNodes.freq	9	9	11	11	13	13	13	16	18	20	38
lNodes.leader	1	1	2	2	3	3	4	4	-1	-1	-1

iNodes.freq	18	22	26	29
iNodes.leader	-1	-1	-1	-1

mid	18	20	18	22

MinFreq = 36

3. นำค่า 18 ในตัวแปร MinFreq ไปเปรียบเทียบกับค่าใน lNodes.freq และ iNodes.freq ว่ามีสมาชิกใดที่มีค่าน้อยกว่าหรือเท่ากับค่าของ MinFreq (= 36) ซึ่งในอะเรย์ lNodes ได้สมาชิกที่มีความถี่ 20 จึงทำการคัดลอกไว้ในอะเรย์ Copy และได้สมาชิกในอะเรย์ iNodes ที่มีความถี่ 22, 26 และ 29 จึงทำการรวมสมาชิกในอะเรย์ Copy และอะเรย์ iNodes ที่มีจำนวนคู่เก็บไว้ในอะเรย์ Temp เพื่อทำการจับคู่สร้างโหนดพ่อแม่ใหม่ดังรูป

Copy.freq	20
Copy.isLeaf	1
Copy.index	1

Temp.freq	20	22	26	29
Temp.isLeaf	1	0	0	0
Temp.index	1	2	3	4

เก็บข้อมูลของโหนดพ่อแม่ใหม่ไว้ในอะเรย์ iNodes และทำการปรับค่า lNodes.leader ให้ตรงกับลำดับการสร้างโหนดพ่อแม่ใหม่ ซึ่งได้ผลดังนี้

lNodes.freq	9	9	11	11	13	13	13	16	18	20	38
lNodes.leader	1	1	2	2	3	3	4	4	5	6	-1

iNodes.freq	36	42	55
iNodes.leader	-1	-1	-1

4. ปรับค่าในอะเรย์ CL เฉพาะสมาชิกช่องที่ได้นำมาสร้างโหนดพ่อแม่ใหม่ให้เป็น 1 และปรับค่าความยาวเพิ่มขึ้น 1 ค่าให้กับสมาชิกที่เป็นโหนดลูกของโหนดพ่อแม่ที่นำมารวมเพื่อสร้างโหนดพ่อแม่ใหม่ในรอบนี้

CL	2	2	2	2	2	2	2	2	1	1	0

รอบที่ 3 ประมวลผลเช่นเดียวกับรอบที่ 2 ซึ่งได้ผลของแต่ละอะเรย์ดังนี้

lNodes.freq	9	9	11	11	13	13	13	16	18	20	38
lNodes.leader	1	1	2	2	3	3	4	4	5	6	-1

iNodes.freq	36	42	55
iNodes.leader	-1	-1	-1

mid	38	36	42	-

MinFreq = 74

Copy.freq	-
Copy.isLeaf	-
Copy.index	-

Temp.freq	42	55
Temp.isLeaf	0	0
Temp.index	1	2

97

ผลการประมวลผลในรอบที่ 3

lNodes.freq	9	9	11	11	13	13	13	16	18	20	38
lNodes.leader	1	1	2	2	3	3	4	4	5	6	7

iNodes.freq	74	97
iNodes.leader	-1	-1

CL	3	3	3	3	3	3	3	3	2	2	1

รอบที่ 4 ประมวลผลเช่นเดียวกับรอบที่ 2 และ 3 ซึ่งได้ผลของแต่ละอะเรย์ดังนี้

lNodes.freq	9	9	11	11	13	13	13	16	18	20	38
lNodes.leader	1	1	2	2	3	3	4	4	5	6	7

iNodes.freq	74	97
iNodes.leader	-1	-1

mid	74	97	-	-

MinFreq = 171

Copy	-

Temp	-

ผลการประมวลผลในรอบที่ 4

lNodes.freq	9	9	11	11	13	13	13	16	18	20	38
lNodes.leader	1	1	2	2	3	3	4	4	5	6	7

iNodes.freq	171
iNodes.leader	-1

CL	4	4	4	4	4	4	4	4	3	3	2

เมื่อได้สมาชิกใน lNodes.leader มีค่าไม่เท่ากับ -1 ทั้งหมดและสมาชิกใน iNodes มี 1 สมาชิกแสดงว่ามีการรวมสมาชิกเพื่อสร้างต้นไม้ฮัฟแมนจนครบแล้ว จะทำการจัดเรียงตำแหน่งสมาชิกในอะเรย์ CL แบบย้อนกลับ ซึ่งได้ผลดังนี้

CL	2	3	3	4	4	4	4	4	4	4	4

ส่วนที่ 2 ขั้นตอนการสร้างชุดรหัสบิต (CW Generation)

การประมวลผลในส่วนการสร้างรหัสบิตจะประมวลผลทั้งหมด 3 รอบดังนี้

รอบที่ 1 สร้างรหัสบิตให้กับสมาชิกแรกของอะเรย์ CW โดยเริ่มด้วยบิต '0' จำนวน 2 บิตตามความยาวของชุดรหัสบิตในสมาชิกช่องแรกของอะเรย์ CL ดังรูป

CL	**2**	3	3	4	4	4	4	4	4	4	4

CW	00										

ปรับค่าตัวแปรดังนี้

ค่าในตัวแปร CCL (เก็บค่าความยาวของรหัสบิตปัจจุบัน) = 2

ค่าในตัวแปร CDPI (เก็บจำนวนของชุดรหัสบิตที่แปลงเสร็จแล้ว) = 1

ค่าในตัวแปร Level (เก็บระดับของโหนดในโครงสร้างต้นไม้) = 1

รอบที่ 2 สร้างรหัสบิตให้กับอะเรย์ CW สำหรับสมาชิกที่มีความยาวเท่ากับ 3

CL	2	3	**3**	4	4	4	4	4	4	4	4

คำนวณเพื่อสร้างรหัสบิตให้กับสมาชิกแรกของกลุ่มสมาชิกที่มีความยาวเท่ากับ 3 ตามลำดับการคำนวณดังนี้

CLDiff <-- CL[CDPI+1] – CL[CDPI] ได้ผลลัพธ์คือ CLDiff <-- 1= 3-2

CW[CDPI+1] <-- (CW[CDPI]+1) * 2^(CLDiff) ได้ผลลัพธ์คือ CW[2]<--"**10**" = 2 =("00"+1)* 2^(1)

CCL <-- CL[CDPI+1] ได้ผลลัพธ์คือ CCL <-- 3

CDPI <-- CDPI+1 ได้ผลลัพธ์คือ CDPI <-- 2

CW	00	010									

ทำการเพิ่มค่าบิตให้กับสมาชิกตัวถัดไปของกลุ่ม

CW	00	010	**011**								

รอบที่ 3 สร้างรหัสบิตให้กับอะเรย์ CW สำหรับสมาชิกที่มีความยาวเท่ากับ 4

CL	2	3	3	4	4	4	4	4	4	4	4

คำนวณเพื่อสร้างรหัสบิตให้กับสมาชิกแรกของกลุ่มสมาชิกที่ความยาวเท่ากับ 4 ตามลำดับการคำนวณดังนี้

CLDiff <-- CL[CDPI+1] – CL[CDPI] ได้ผลลัพธ์คือ CLDiff <-- 1= 4-3

CW[CDPI+1] <-- (CW[CDPI]+1) * 2^(CLDiff) ได้ผลลัพธ์คือ CW[2]<-- “1000” = 8 =(“011”+1)* 2^(1)

CCL <-- CL[CDPI+1] ได้ผลลัพธ์คือ CCL <-- 4

CDPI <-- CDPI+1 ได้ผลลัพธ์คือ CDPI <--4

CW	00	010	011	1000							

ทำการเพิ่มค่าบิตให้กับสมาชิกตัวถัดไปของกลุ่ม

CW	00	010	011	1000	**1001**	**1010**	**1011**	**1100**	**1101**	**1110**	**1111**

หลังจากได้รหัสบิตในอะเรย์ CW ครบทุกสมาชิก ก็จะทำการจัดเรียงตำแหน่งของสมาชิกในอะเรย์แบบย้อนกลับ ซึ่งจะได้ผลลัพธ์ดังนี้

CW	1111	1110	1101	1100	1011	1010	1001	1000	011	010	00

ดังนั้นจะได้ชุดรหัสบิตที่แทนแต่ละสัญลักษณ์ดังนี้

สัญลักษณ์	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}
ชุดรหัสบิต	1111	1110	1101	1100	1011	1010	1001	1000	011	010	00

ข้อดีและข้อจำกัดของขั้นตอนวิธีของอราสเมื่อเทียบกับขั้นตอนวิธีอีเอส-พาร์ฮัฟมีดังนี้

ข้อดี	สามารถสร้างรหัสบิตของแต่ละสัญลักษณ์ได้เมื่อเสร็จสิ้นการประมวลผล
ข้อจำกัด	วิธีนี้ใช้โครงสร้างอะเรย์จำนวนมากในการประมวลผล ทำให้ต้องจองพื้นที่ในหน่วยความจำเพื่อใช้ในการประมวลผลเป็นจำนวนมากตามไปด้วย

2.6 ขั้นตอนวิธีการรวมข้อมูลแบบขนานด้วยบีเอสอาร์

การประมวลผลแบบบีเอสอาร์ (BSR : Broadcasting with Selective Reduction) [1][16] ได้นำเสนอในปี ค.ศ.1989 โดย Akl และ Stojmenović [1] ซึ่งบีเอสอาร์เป็นรูปแบบการประมวลผลที่ประยุกต์กระบวนการกระจายข้อมูล (Data Broadcasting) มาใช้กระจายคำสั่งเข้าไปประมวลผลในหน่วยประมวลผลทั้งหมดในเวลาเดียวกันที่เรียกว่าการกระจายคำสั่ง (BI: Broadcasting Instruction) พร้อมทั้งเขียนผลลัพธ์ของการประมวลผลที่ได้ลงในหน่วยความจำร่วมซึ่งทำงานแบบ many-to-many ร่วมกับการประมวลผลแบบซีอาร์ซีดับเบิลยูพีแรมโมเดล (CRCW PRAM Model) จึงทำให้การประมวลผลแบบบีเอสอาร์มีประสิทธิภาพด้านความเร็วมากกว่ารูปแบบการประมวลผลแบบซีอาร์ซีดับเบิลยูพีแรมโมเดล

การประมวลผลแบบบีเอสอาร์ มีดังนี้

$$u_j := \mathop{\Re}_{1 \le i \le n} d_i \,|\, t_i \sigma l_j$$

โดยที่ n คือ จำนวนหน่วยประมวลผล

d_i คือ ข้อมูลย่อยๆ ที่กระจายโดยหน่วยประมวลผล i ซึ่ง $1 \le i \le n$

t_i คือ ข้อมูลที่กระจายในทุกหน่วยประมวลผล i ซึ่ง $1 \le i \le n$

l_i คือ ค่าจำกัดที่ใช้โดยหน่วยประมวลผล j

σ คือ ตัวดำเนินการ โดยที่ $\sigma \in \{<, \le, =, >, \ge, \ne\}$

$\Re$ คือ กระบวนการคำนวณเพื่อสรุปผลการเปรียบเทียบตามการดำเนินการ σ ที่เป็นจริง โดยที่ $\Re \in \{\Sigma, \Pi, \wedge, \vee, \oplus, \bigcup, \bigcap\}$

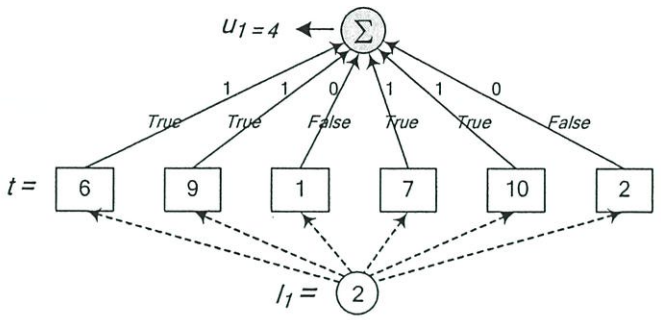
u_j คือ ผลลัพธ์ที่ได้จากกระบวนการ $\Re$ ซึ่งประมวลผลด้วยหน่วยประมวลผล j

การประมวลผลของบีเอสอาร์ เริ่มจากการเปรียบเทียบค่าระหว่างค่าใน t_i กับค่าใน l_j ด้วยตัวดำเนินการ σ ตั้งแต่ i=1 ถึง i=n ในแบบขนาน หากผลการเปรียบเทียบของคู่ลำดับ t_i, l_jใดเป็นจริง จะกระทำตามกระบวนการ $\Re$ ด้วยค่า d_i แล้วเก็บผลของกระบวนการที่ได้ไว้ในตัวแปร u_j

ตัวอย่างเช่น $u_1 := \sum_{1 \le i \le n} 1 \,|\, t_i > l_1$ มีความหมายว่า ถ้าค่าของ t ตัวที่ i ใดมีค่ามากกว่า 1 ตัวที่ 1 แล้ว ให้ทำการบวกค่า 1 ให้กับตัวแปร u ตัวที่ 1 หากสมมุติค่าของ t และ l_1 ดังนี้

$$t = 6, 9, 1, 7, 10, 2 \quad \text{และ} \quad l_1 = 2$$

กระบวนการจะทำการกระจายค่า l_1 ที่มีค่าเท่ากับ 2 ไปเปรียบเทียบกับค่าใน t ทุกค่า หากค่า t ใดมีค่ามากกว่า ก็ให้ทำการเพิ่มค่า 1 ให้กับตัวแปร u_1 ดังนั้นผลลัพธ์ที่เก็บในตัวแปร u_1 คือ 4 ดังแสดงในรูปที่ 2.16



รูปที่ 2.16 แสดงการทำงานของขั้นตอนวิธีบีเอสอาร์

จากคุณลักษณะของการประมวลผลแบบบีเอสอาร์ จึงได้มีการนำการประมวลผลแบบบีเอสอาร์ มาพัฒนาในงานต่างๆ อาทิ การพัฒนาขั้นตอนวิธีในการค้นหารูปทรงเรขาคณิตและจุดล้อมรอบรูปร่างด้วยบีเอสอาร์ [22] และการพัฒนาขั้นตอนวิธีการสร้างต้นไม้สองทางและการค้นหาเส้นทางจากโหนดรากไปยังโหนดใบด้วยบีเอสอาร์ ซึ่งทำให้ได้ความซับซ้อนด้านเวลาเป็น O(1) [25][26][27] นอกจากนี้ Xiang และ Ushijima [24] ได้นำการประมวลผลแบบบีเอสอาร์มาพัฒนาในการรวมชุดข้อมูลซึ่งทำให้ได้ค่าความซับซ้อนด้านเวลาเป็น O(1) ซึ่งขั้นตอนวิธีในการรวมชุดข้อมูลแบบบีเอสอาร์ที่ Xiang และ Ushijima ได้พัฒนาขึ้น [24] มีดังนี้

2.6.1 เมิร์จ-บีเอสอาร์-1

ขั้นตอนวิธีการรวมชุดข้อมูลแบบบีเอสอาร์ แบบที่ 1 มีรายละเอียดดังนี้

ขั้นตอนที่ 1 นำข้อมูลครึ่งแรกของข้อมูลชุดที่หนึ่ง (A) เก็บไว้ในอะเรย์ C ตั้งแต่สมาชิกช่องที่ 1 เป็นต้นไปจนครบข้อมูลที่นำมาวาง

ขั้นตอนที่ 2 นำข้อมูลครึ่งหลังของข้อมูลชุดที่หนึ่ง (A) เก็บไว้ในอะเรย์ C ในสมาชิกช่องที่ N/2 + 1 เป็นต้นไปจนครบข้อมูลชุดแรก

ขั้นตอนที่ 3 นำข้อมูลครึ่งแรกของข้อมูลชุดที่สอง (B) เก็บไว้ในอะเรย์ C ในสมาชิกช่องที่ถัดจากชุดสมาชิกส่วนแรกของชุดข้อมูลที่หนึ่ง (A) ที่นำมาวางในขั้นตอนแรก

ขั้นตอนที่ 4 นำข้อมูลครึ่งหลังของข้อมูลชุดที่สอง (B) เก็บไว้ในอะเรย์ C ในสมาชิกช่องที่ถัดจากชุดสมาชิกส่วนหลังของชุดข้อมูลที่หนึ่ง (A) ที่นำมาวางในขั้นตอนที่สอง

ขั้นตอนที่ 5 จัดเรียงข้อมูลในอะเรย์ C ส่วนครึ่งแรกตามลำดับจากน้อยไปมาก

ขั้นตอนที่ 6 จัดเรียงข้อมูลในอะเรย์ C ส่วนครึ่งหลังตามลำดับจากน้อยไปมาก

ขั้นตอนที่ 7 จัดเรียงข้อมูลในอะเรย์ C ส่วนกลางคือตั้งแต่สมาชิกช่องที่ N/2 + 1 จนถึงสมาชิกช่องที่ N/2 +1+ N ตามลำดับจากน้อยไปมาก

ซึ่งกระบวนการในขั้นตอนที่ 1 ถึง 4 รวมถึงการจัดเรียงข้อมูลเป็นการประมวลผลแบบขนานซึ่งมีค่าความซับซ้อนด้านเวลาคือ O(1) ทำให้วิธีนี้มีค่าความซับซ้อนด้านเวลาเท่ากับ O(1)

ขั้นตอนวิธีที่ 2.9 การรวมชุดข้อมูลด้วยบีเอสอาร์แบบที่ 1 มีชื่อว่าเมิร์จ-บีเอสอาร์-1 (Merge_BSR_1)

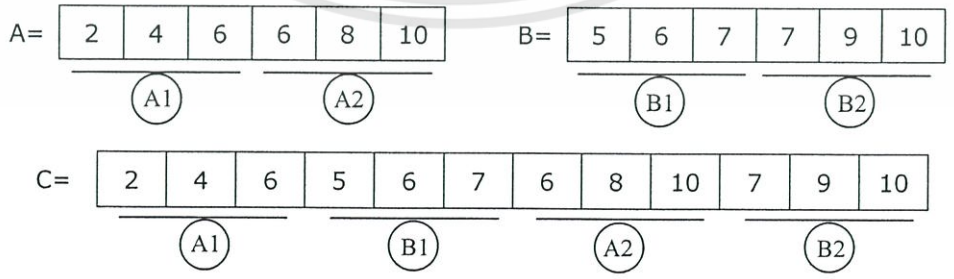
```
Procedure Merge_BSR_1(A, B: S_N; C:S_2N);
    var j : integer;
    procedure sort(k: integer);
        var  a : array[1..N] of integer;
             s : array[1..N] of real;
             j : integer;
    begin
        Forall Processors Pj(j :=1 to N) pardo
            a[j] := 0;
            a[j] := Σ 1 | C[k+i] < C[k+j];
            s[j] := a[j] + (j-1)/j;
            a[j] := Σ 1 | s[i] < s[j];
            C[k+a[j]] := C[k+j];
        End Forall
    end;
Begin
    Forall Processors Pj(j :=1 to N) pardo
        If j ≤ n/2  Then C[j] := A[j]; C[N+j] := A[N/2 +j];
        Else  C[j] := B[j- N/2]; C[N+j] := B[j];
        End If
    End Forall
    sort(0); sort(N); sort(N/2);
End;
```

โดยสามารถอธิบายขั้นตอนการรวมข้อมูลโดยสรุป ได้ดังนี้

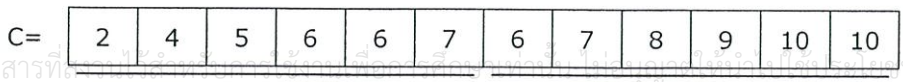
ตัวอย่างที่ 3 ตัวอย่างแสดงขั้นตอนการรวมชุดข้อมูลด้วยขั้นตอนวิธีการเมิร์จ-บีเอสอาร์-1 (Merge_BSR_1)

สมมุติชุดข้อมูลดังนี้ A = {2, 4, 6, 6, 8, 10} และ B = {5, 6, 7, 7, 9, 10}

ขั้นตอนที่ 1 ถึง 4 สามารถแสดงการทำงานได้ดังนี้



ขั้นตอนที่ 5 และ 6 เรียงลำดับข้อมูลภายในอะเรย์ C โดยแบ่งช่วงการเรียงลำดับข้อมูลเป็น 2 ส่วนคือส่วนครึ่งแรกและส่วนครึ่งหลัง



ขั้นตอนที่ 7 เรียงลำดับข้อมูลส่วนกลางของข้อมูลภายในอะเรย์ C

C=	2	4	5	6	6	6	7	7	8	9	10	10

ดังนั้นการรวมชุดข้อมูลระหว่าง A กับ B จึงได้ผลลัพธ์ดังนี้

C=	2	4	5	6	6	6	7	7	8	9	10	10

2.6.2 เมิร์จ-บีเอสอาร์-2

สำหรับขั้นตอนวิธีการรวมชุดข้อมูลแบบขนานด้วยการประมวลผลแบบบีเอสอาร์ แบบที่ 2 มีรายละเอียดดังนี้

ขั้นตอนวิธีที่ 2.10 การรวมชุดข้อมูลด้วยบีเอสอาร์ แบบที่ 2 มีชื่อว่าเมิร์จ-บีเอสอาร์-2 (Merge_BSR_2)

```
Procedure Merge_BSR_2(A, B: S_N; C:S_2N);
    var  Ind : array[1..N] of integer;
         j : integer;
Begin
    Forall Processors (j :=1 to N) pardo
        Ind[j] := 0;
        Ind[j] := Σ 1 | B[i] < A[j];
        C[j + Ind[j]] := A[j];
        Ind[j] := 0;
        Ind[j] := Σ 1 | A[i] ≤ B[j];
        C[j+Ind[j]] := B[j];
    End Forall
End;
```

รายละเอียดขั้นตอนการทำงานมีดังนี้

1. กระจายสมาชิกของ B แต่ละค่าไปยังทุกๆ หน่วยประมวลผล (ซึ่งมีสมาชิกของ A อยู่ตามลำดับของสมาชิก) โดยเปรียบเทียบค่าสมาชิกของ B ตัวที่ i กับ สมาชิกของ A ทุกค่าด้วยจำนวนหน่วยประมวลผลเท่ากับ A ซึ่งหากค่าสมาชิกของ B ตัวที่ i น้อยกว่าค่าสมาชิกของ A ตัวที่ j ให้ทำการเพิ่มค่าขึ้นหนึ่งค่า โดยจำนวนของข้อมูลที่ตรงตามเงื่อนไขจะเก็บไว้ที่ตัวแปร Ind ช่องที่ j (ด้วยหน่วยประมวลผล j)

2. นำสมาชิกของ A ตัวที่ j มาเก็บไว้ในอะเรย์ C ช่องที่ j+Ind[j] (เป็นช่องที่นำจำนวนของ B ที่ตรงตามเงื่อนไขกับ A ในแต่ละหน่วยประมวลผล มาบวกกับค่าหมายเลขของหน่วยประมวลผล)

3. กระจายสมาชิกของ A แต่ละค่าไปยังทุกๆ หน่วยประมวลผล (ซึ่งจะมีสมาชิกของ B อยู่ตามลำดับของสมาชิก) แล้วทำการเปรียบเทียบค่าของ A ตัวที่ i กับ B ตัวที่ j (ในหน่วยประมวลผล j) หากค่า A น้อยกว่าหรือเท่ากับค่า B ในแต่ละหน่วยประมวลผล ก็ให้เพิ่มค่า Ind ของหน่วยประมวลผล j นั้นๆ ขึ้น 1 ค่า

4. นำสมาชิกของ B ตัวที่ j มาเก็บไว้ในอะเรย์ C ช่องที่ j+Ind[j] (เป็นช่องที่นำจำนวนของ Ind ที่ตรงตามเงื่อนไข มาบวกกับค่าหมายเลขของหน่วยประมวลผล)

<u>ตัวอย่างที่ 4</u> ตัวอย่างแสดงขั้นตอนการรวมชุดข้อมูลด้วยขั้นตอนวิธีการเมิร์จ-บีเอสอาร์-2 (Merge_BSR_2)

สมมุติชุดข้อมูลดังนี้ A = {2, 4, 6, 6, 8, 10} และ B = {5, 6, 7, 7, 9, 10}

Ind[j] := 0;

Ind[j] := Σ 1 | *B[i]* < *A[j]*;

j	1	2	3	4	5	6
A[j] / B[i]	2	4	6	6	8	10
5	0	0	1	1	1	1
6	0	0	0	0	1	1
7	0	0	0	0	1	1
7	0	0	0	0	1	1
9	0	0	0	0	0	1
10	0	0	0	0	0	0
Ind[j]	0	0	1	1	4	5

C[j + Ind[j]] := *A[j]*; ได้ผลลัพธ์ดังนี้

C[1+0]= ***C*[1]** := *A*[1]=**2**;
C[2+0]= ***C*[2]** := *A*[2]=**4**;
C[3+1]= ***C*[4]** := *A*[3]=**6**;
C[4+1]= ***C*[5]** := *A*[4]=**6**;
C[5+4]= ***C*[9]** := *A*[5]=**8**;
C[6+5]= ***C*[11]** := *A*[6]=**10**;

Ind[j] := 0;

Ind[j] := Σ 1 | *A[i]* ≤ *B[j]*;

j	1	2	3	4	5	6
B[j] / A[i]	5	6	7	7	9	10
2	1	1	1	1	1	1
4	1	1	1	1	1	1
6	0	1	1	1	1	1
6	0	1	1	1	1	1
8	0	0	0	0	1	1
10	0	0	0	0	0	1
Ind[j]	2	4	4	4	5	6

C[j + Ind[j]] := *B[j]*; ได้ผลลัพธ์ดังนี้

C[1+2]= ***C*[3]** := *B*[1]=**5**;
C[2+4]= ***C*[6]** := *B*[2]=**6**;
C[3+4]= ***C*[7]** := *B*[3]=**7**;
C[4+4]= ***C*[8]** := *B*[4]=**7**;
C[5+5]= ***C*[10]** := *B*[5]=**9**;
C[6+6]= ***C*[12]** := *B*[6]=**10**;

เมื่อนำค่าในอะเรย์ C ที่ได้จากการประมวลผลมาแสดงจะได้ผลลัพธ์ดังนี้

	1	2	3	4	5	6	7	8	9	10	11	12
C=	2	4	5	6	6	6	7	7	8	9	10	10
	A[1]	A[2]	B[1]	A[3]	A[4]	B[2]	B[3]	B[4]	A[5]	B[5]	A[6]	B[6]

บทที่ 3
การเข้ารหัสของฮัฟแมนแบบขนาน

งานวิจัยนี้เป็นงานวิจัยที่นำเสนอขั้นตอนวิธีแบบขนาน ซึ่งผู้วิจัยได้ทำการศึกษาขั้นตอนวิธีการเข้ารหัสของฮัฟแมนบนกระบวนการแบบขนาน โดยได้ปรับปรุงขั้นตอนวิธีการเข้ารหัสของฮัฟแมนบนกระบวนการแบบขนานที่เคยมีผู้ได้เสนอไว้ [18][19] ให้มีประสิทธิภาพมากขึ้น โดยเน้นเรื่องการแก้ไขข้อจำกัดในส่วนการสร้างรหัสบิตที่เสนอในเอกสารอ้างอิงที่ [18] และเสนอการปรับปรุงขั้นตอนวิธีให้มีความซับซ้อนน้อยลง อีกทั้งใช้เนื้อที่ในการประมวลผลน้อยกว่าขั้นตอนวิธีที่เสนอในเอกสารอ้างอิงที่ [19]

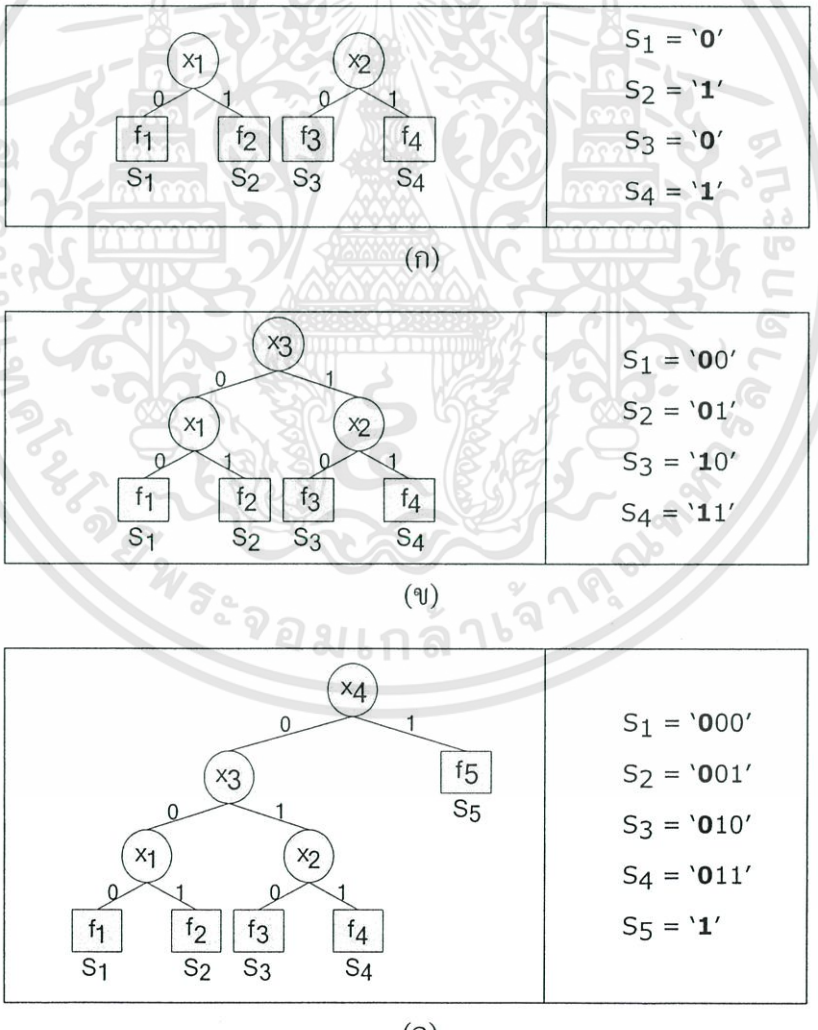
ขั้นตอนวิธีการแบบขนานที่นำเสนอนี้เป็นการสร้างรหัสบิตของแต่ละสัญลักษณ์โดยใช้หลักการเดียวกับขั้นตอนวิธีการเข้ารหัสโดยใช้โครงสร้างต้นไม้ฮัฟแมน แต่ในงานวิจัยที่นำเสนอนี้มีข้อดีคือ จะสร้างชุดรหัสบิตไปพร้อมกับการสร้างต้นไม้ฮัฟแมน ดังนั้นเมื่อต้นไม้ฮัฟแมนถูกสร้างเสร็จก็จะทำให้ได้ชุดรหัสบิตของแต่ละสัญลักษณ์ที่สมบูรณ์ นอกจากนี้ผู้วิจัยได้นำขั้นตอนวิธีการรวมข้อมูลแบบบีเอสอาร์ (Merging on BSR) [24] ซึ่งได้กล่าวไปแล้วในบทที่ 2 มาประยุกต์ใช้ในขั้นตอนวิธีที่ได้นำเสนอเพื่อลดความซับซ้อนด้านเวลา โดยขั้นตอนวิธีการที่ได้นำเสนอในงานวิจัยนี้เป็นการสร้างรหัสบิตแทนสัญลักษณ์ของข้อมูลตามลักษณะการเข้ารหัสของฮัฟแมนโดยไม่มีการคำนวณความยาวของรหัสบิตก่อนการสร้างรหัสบิตจริง อีกทั้งขั้นตอนวิธีที่นำเสนอดังกล่าวสามารถนำไปพัฒนาได้สะดวกมากขึ้น และด้วยการประยุกต์ขั้นตอนวิธีการรวมข้อมูลแบบบีเอสอาร์ [24] ซึ่งมีค่าความซับซ้อนด้านเวลาเท่ากับ O(1) มาใช้ในขั้นตอนวิธีที่นำเสนอทำให้ขั้นตอนวิธีที่นำเสนอในงานวิจัยนี้ได้ค่าความซับซ้อนด้านเวลาที่น้อยที่สุดเป็น $O(\log\ n)$ เมื่อโครงสร้างต้นไม้ฮัฟแมนมีการกระจายกิ่งภายในแบบสมดุล และได้ค่าความซับซ้อนด้านเวลาที่มากที่สุดคือ O(n) เมื่อโครงสร้างต้นไม้ฮัฟแมนมีการกระจายกิ่งไปในทิศทางเดียว สำหรับกรณีที่โครงสร้างต้นไม้ฮัฟแมนมีการกระจายที่ไม่สมดุลและไม่ได้กระจายไปในทิศทางเดียว ซึ่งเป็นกรณีที่สามารถเกิดได้ทั่วไปจะมีค่าความซับซ้อนด้านเวลาเป็น $O\left(\frac{n+\log n}{2}\right) = O(n)$ ซึ่งจะแสดงการพิสูจน์ในบทที่ 4

3.1 หลักการของขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอ

ขั้นตอนวิธีการเข้ารหัสของฮัฟแมนในแบบขนานที่นำเสนอนี้มีหลักการคือ จะทำการสร้างรหัสบิตของแต่ละสัญลักษณ์ไปพร้อมกับการสร้างต้นไม้ย่อยในต้นไม้ฮัฟแมนแบบขนาน ซึ่งทำให้ได้รหัสบิตของแต่ละสัญลักษณ์ที่สมบูรณ์ เมื่อโครงสร้างต้นไม้ฮัฟแมนถูกสร้างจนกระทั่งถึงโหนดราก (Root Node) นั่นคือเมื่อสัญลักษณ์ใดๆ ที่ถูกนำไปสร้างโหนดพ่อแม่ใหม่สัญลักษณ์นั้น

จะได้รหัสบิตเริ่มต้นทันที และเมื่อโหนดพ่อแม่ของสัญลักษณ์นั้นถูกนำไปรวมเพื่อสร้างโหนดพ่อแม่ใหม่ในลำดับถัดไป จะทำให้สัญลักษณ์ทุกสัญลักษณ์ที่เป็นโหนดลูกของโหนดพ่อแม่เดิมมีรหัสบิตเพิ่มด้านหน้าบิตเดิมไปด้วย

ตัวอย่างเช่นในรูปที่ 3.1 เมื่อสัญลักษณ์ S_1, S_2 และ S_3, S_4 ถูกนำมารวมกันตามลำดับคู่เพื่อสร้างโหนดพ่อแม่ x_1 และ x_2 ในแบบขนานดังในรูปที่ 3.1 (ก) จะทำให้สัญลักษณ์ S_1, S_2, S_3, S_4 มีบิตเริ่มต้นเป็น '0', '1', '0', '1' ตามลำดับพร้อมกัน และถ้านำโหนดพ่อแม่ x_1 และโหนดพ่อแม่ x_2 รวมกันเพื่อสร้างโหนดพ่อแม่ x_3 ดังแสดงในรูปที่ 3.1 (ข) จะทำให้สัญลักษณ์ S_1, S_2 มีบิต '0' เพิ่มขึ้นด้านหน้าบิตเดิมของแต่ละสัญลักษณ์ เนื่องจากโหนดพ่อแม่ x_1 ถูกนำไปรวมเป็นโหนดลูกฝั่งซ้ายของโหนดพ่อแม่ x_3 ทำให้โหนดใบหรือสัญลักษณ์ทั้งหมดที่อยู่ภายใต้โหนดพ่อแม่ x_1 จะต้องได้รหัสบิต '0' เพิ่มขึ้นด้านหน้าบิตเดิม และสัญลักษณ์ S_3, S_4 มีบิต '1' เพิ่มด้านหน้าเพราะโหนดพ่อแม่ x_2 ถูกนำไปรวมเป็นโหนดลูกฝั่งขวาของโหนดพ่อแม่ x_3 ดังรูปที่ 3.1(ข)



รูปที่ 3.1 หลักการขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอ

ในทำนองเดียวกันหากโหนดพ่อแม่ x3 ถูกนำไปรวมกับโหนดอื่นเพื่อสร้างโหนดพ่อแม่ x4 จะทำให้สัญลักษณ์ S_1, S_2, S_3, S_4 ถูกเพิ่มบิต ดังรูปที่ 3.1(ค) โหนดพ่อแม่ x3 ถูกนำไปรวมเป็นโหนดลูกฝั่งซ้ายของโหนดพ่อแม่ x4 ทำให้สัญลักษณ์ที่เป็นโหนดใบภายใต้โหนดพ่อแม่ x3 ซึ่งมีโหนดใบที่แทนสัญลักษณ์ S_1, S_2, S_3, S_4 อยู่ ถูกเพิ่มบิต '0' ไว้ด้านหน้าบิตเดิมของแต่ละสัญลักษณ์พร้อมกัน และสัญลักษณ์ที่แทนด้วยโหนดใบที่อยู่ในกิ่งขวาจะถูกกำหนดด้วยบิต '1' ประมวลผลเช่นนี้จนกระทั่งได้โหนดรากของต้นไม้ฮัฟแมน ซึ่งเมื่อถึงการประมวลผลรอบสุดท้ายจะทำให้ได้รหัสบิตที่สมบูรณ์สำหรับใช้แทนสัญลักษณ์แต่ละสัญลักษณ์

สรุปขั้นตอนการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอในงานวิจัย มีขั้นตอนดังนี้

ขั้นตอนที่ 1 หน่วยประมวลผล P_1 คำนวณความถี่รวมของความถี่ที่น้อยที่สุด 2 ค่า

ขั้นตอนที่ 2 ทุ กหน่วยประมวลผล (P_i) ที่มีค่าความถี่ไม่เกินความถี่รวมในขั้นที่ 1 จะดำเนินการต่อไปนี้พร้อมๆ กัน

1. สร้างโหนดพ่อแม่ใหม่จากโหนดที่มีเงื่อนไขตรงตามที่กำหนดไว้ โดยการจับคู่โหนดที่เลือกไว้ตามลำดับความถี่พร้อมกันเพื่อสร้างโหนดพ่อแม่ใหม่
2. เพิ่มบิตให้ทุกสัญลักษณ์ที่เป็นโหนดใบภายใต้โหนดพ่อแม่ใหม่ โดยเพิ่มบิต '0' ด้านหน้าบิตเดิมให้กับแต่ละสัญลักษณ์ที่เป็นโหนดใบในกิ่งฝั่งซ้ายของโหนดพ่อแม่ใหม่ และเพิ่มบิต '1' ด้านหน้าบิตเดิมให้กับแต่ละสัญลักษณ์ที่เป็นโหนดใบในกิ่งฝั่งขวาของโหนดพ่อแม่ใหม่

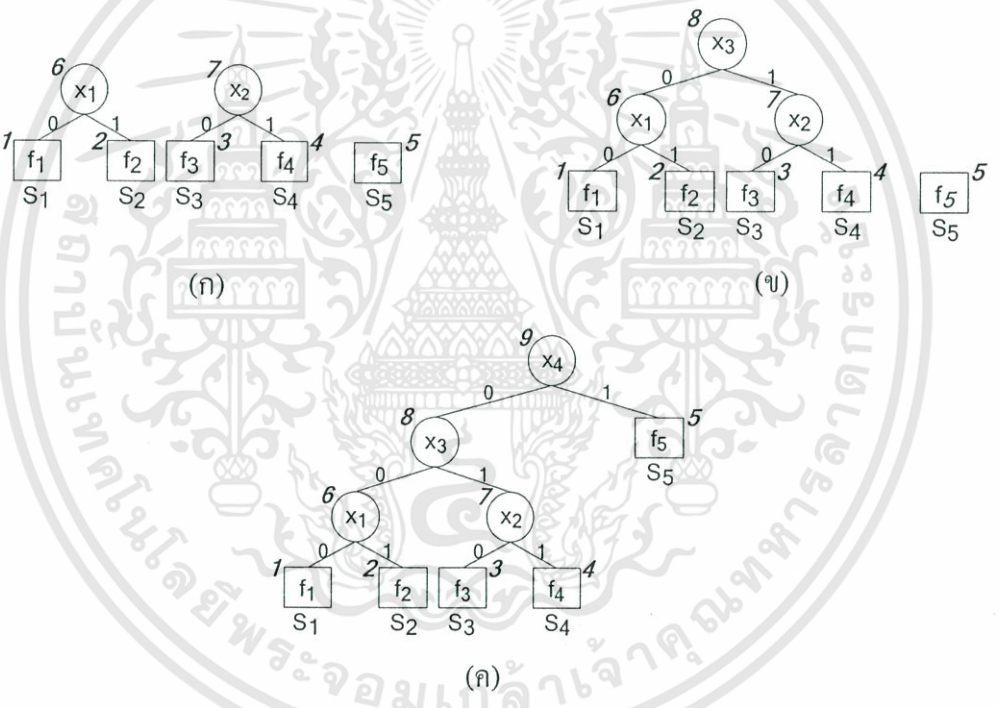
ขั้นตอนที่ 3 ลบโหนดที่ถูกนำไปรวม แล้วออกจากชุดรายการโหนดสำหรับเลือกและเพิ่มโหนดพ่อแม่ใหม่ที่สร้างขึ้นเข้าไปในชุดรายการโหนดสำหรับเลือกในรอบถัดไป

ทำซ้ำในขั้นตอนที่ 1 ถึง 3 จนกระทั่งได้โหนดราก

3.2 การนำหลักการของขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอมาใช้กับโครงสร้างอะเรย์

หลักการขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอซึ่งได้อธิบายในหัวข้อที่ 3.1 สามารถนำไปปรับปรุงใช้ร่วมกับโครงสร้างข้อมูลที่มีประสิทธิภาพได้ ซึ่งในที่นี้ผู้วิจัยได้นำหลักการขั้นตอนวิธีการดังกล่าวมาใช้ร่วมกับโครงสร้างอะเรย์ เพื่อเปรียบเทียบการใช้หน่วยความจำในการประมวลผลระหว่างขั้นตอนวิธีการที่นำเสนอกับขั้นตอนวิธีการเดิมที่นำเสนอในเอกสารอ้างอิงที่ [19] ซึ่งใช้โครงสร้างอะเรย์จำนวน 7 อะเรย์ในการนำเสนอขั้นตอนวิธี

สำหรับการเข้ารหัสของฮัฟแมนด้วยขั้นตอนวิธีที่นำเสนอโดยใช้โครงสร้างอะเรย์นี้จะใช้เพียง 3 อะเรย์ ซึ่งจะมีการกำหนดหมายเลขประจำโหนดให้กับทุกโหนดในต้นไม้ฮัฟแมนเพื่อใช้ในการพิจารณาสำหรับเพิ่มบิต '0' หรือ '1' ให้กับแต่ละสัญลักษณ์ โดยการกำหนดหมายเลขประจำโหนดนี้จะกำหนดให้กับโหนดใบทุกโหนดก่อน ซึ่งมีหมายเลขที่กำหนดตั้งแต่ 1 ถึง n เมื่อ n คือจำนวนสัญลักษณ์ในชุดข้อมูล และเมื่อโหนดพ่อแม่ใหม่ถูกสร้างขึ้น โหนดพ่อแม่ใหม่เหล่านั้นก็จะถูกกำหนดหมายเลขประจำโหนดเช่นกัน โดยหมายเลขของโหนดพ่อแม่จะมีค่าเพิ่มจาก n ขึ้นโหนดละ 1 ค่าตามลำดับการสร้าง ดังรูปที่ 3.2 สมมุติมีสัญลักษณ์ในชุดข้อมูลนำเข้าจำนวน 5 สัญลักษณ์ ดังนั้นหมายเลขประจำโหนดของโหนดใบจะมีตั้งแต่ 1 ถึง 5 และเมื่อโหนดพ่อแม่ x_1 และ x_2 ถูกสร้างขึ้น ก็จะถูกกำหนดด้วยหมายเลข 6 และ 7 ตามลำดับการสร้างดังแสดงในรูปที่ 3.2 (ก) และในรอบถัดไป เมื่อโหนดพ่อแม่ใหม่ถูกสร้างขึ้นอีก โหนดพ่อแม่แม่ใหม่นั้นก็จะถูกกำหนดหมายเลขประจำโหนดเพิ่มขึ้นดังแสดงในรูปที่ 3.2 (ข) และ 3.2 (ค) ตามลำดับ



รูปที่ 3.2 การกำหนดหมายเลขประจำโหนดให้กับแต่ละโหนดในต้นไม้ฮัฟแมน

ในการใช้โครงสร้างอะเรย์ในการเก็บข้อมูลและประมวลผลจำเป็นต้องอาศัยการคำนวณพื้นฐานซึ่งมีตัวแปรดังนี้

n	สำหรับเก็บจำนวนสัญลักษณ์ทั้งหมดในชุดข้อมูลนำเข้า
c_index	สำหรับกำหนดหมายเลขประจำโหนดพ่อแม่ที่ถูกสร้างขึ้นมาใหม่
t	สำหรับเก็บความถี่รวมของคู่สมาชิกคู่แรกในอะเรย์ Temp ที่มีความถี่น้อยที่สุด

a	สำหรับเก็บจำนวนสมาชิกของอะเรย์ Temp ที่เหมาะสมในการนำไปจับคู่เพื่อสร้างโหนดพ่อแม่ใหม่ ซึ่งสมาชิกเหล่านั้นจะต้องมีค่าความถี่ (Temp.index) น้อยกว่าหรือเท่ากับค่าในตัวแปร t

และอะเรย์ที่ใช้ในขั้นตอนวิธีดังนี้

Temp	สำหรับเก็บข้อมูลของสมาชิกที่แทนโหนดซึ่งจะนำไปจับคู่เพื่อสร้างโหนดพ่อแม่ใหม่ โดยสามารถประกอบไปด้วย สมาชิกที่แทนโหนดใบและสมาชิกที่แทนโหนดพ่อแม่ อะเรย์ Temp ประกอบด้วยขอบเขต (field) ดังนี้	
	index	สำหรับเก็บหมายเลขประจำโหนดของสมาชิกนั้นๆ โดยหมายเลขประจำโหนดจะเริ่มต้นตั้งแต่ 1 ถึง n (n คือจำนวนของสัญลักษณ์ในชุดข้อมูลนั้นๆ) ให้แก่สมาชิกที่แทนโหนดใบหรือสัญลักษณ์ข้อมูล เช่น หากสัญลักษณ์นำเข้ามี 10 สัญลักษณ์ หมายเลขประจำโหนดของโหนดใบจะมีตั้งแต่ 1 ถึง 10 แต่หากเป็นสมาชิกที่แทนโหนดพ่อแม่ หมายเลขประจำโหนดของสมาชิกนั้นจะเป็น 11 และ 12 ตามลำดับ
	freq	สำหรับเก็บค่าความถี่หรือน้ำหนักของแต่ละโหนด

IntTemp	สำหรับเก็บรายละเอียดของสมาชิกที่แทนโหนดพ่อแม่ที่ถูกสร้างขึ้นใหม่ในรอบประมวลผลปัจจุบัน ซึ่งประกอบด้วยขอบเขตดังนี้	
	index	สำหรับเก็บหมายเลขประจำโหนดพ่อแม่ใหม่ที่สร้างขึ้น โดยคำนวณจาก หมายเลขประจำโหนดล่าสุดในรอบประมวลผลก่อนหน้าบวกด้วยลำดับการสร้างโหนดพ่อแม่ใหม่ในรอบประมวลผลปัจจุบัน โดยใช้ตัวแปร c_index
	freq	สำหรับเก็บค่าความถี่หรือน้ำหนักของโหนดพ่อแม่ใหม่ ซึ่งได้จากการรวมค่าความถี่ของสมาชิกที่แทนโหนดลูกในฝั่งซ้าย กับสมาชิกที่แทนโหนดลูกในฝั่งขวา

S	สำหรับเก็บรายละเอียดของสัญลักษณ์ ซึ่งประกอบด้วยขอบเขตดังนี้	
	index	สำหรับเก็บหมายเลขประจำโหนดของพ่อแม่ปัจจุบัน โดยค่าในขอบเขต index ของแต่ละสมาชิกจะถูกปรับค่าให้ตรงกับหมายเลขประจำโหนดพ่อแม่ใหม่ของสมาชิกนั้นที่ถูกสร้างขึ้นในแต่ละรอบ
	cw	สำหรับเก็บรหัสบิตของแต่ละสมาชิกที่ถูกสร้างขึ้นในแต่ละรอบของการประมวลผล

โดยมีขั้นตอนย่อยตามลำดับต่อไปนี้

ขั้นตอนที่ 1 หน่วยประมวลผล P_1 รวมความถี่ที่น้อยที่สุด 2 ค่าของอะเรย์ Temp (อะเรย์สำหรับเก็บโหนดที่จะนำมาสร้างโหนดพ่อแม่ใหม่) เก็บไว้ในตัวแปร t

ขั้นตอนที่ 2 ทุกหน่วยประมวลผล (P_i) อ่านค่า t พร้อมกัน เพื่อนำไปเปรียบเทียบกับค่าความถี่ของสมาชิกในอะเรย์ Temp ในหน่วยประมวลผลของตนเอง และเก็บจำนวนโหนดที่มีความถี่ใน Temp น้อยกว่าหรือเท่ากับค่า t ไว้ในตัวแปร a ด้วยวิธีการประมวลผลแบบบีเอสอาร์ (BSR) จากนั้นทุกหน่วยประมวลผลจะดำเนินการต่อไปนี้พร้อมๆ กัน

1. จับคู่สมาชิกในอะเรย์ Temp ตามลำดับ จำนวน $\lfloor a/2 \rfloor$ คู่เพื่อสร้างโหนดพ่อแม่ใหม่เก็บข้อมูลโหนดพ่อใหม่ใหม่ที่ได้ไว้ในอะเรย์ IntTemp
2. เพิ่มบิต '0' ให้กับทุกสัญลักษณ์ (โหนดใบ) ที่อยู่ภายใต้ต้นไม้ย่อยของโหนดลูกฝั่งซ้ายและเพิ่มบิต '1' ให้กับทุกสัญลักษณ์ (โหนดใบ) ที่อยู่ภายใต้ย่อยของโหนดลูกฝั่งขวา

ขั้นตอนที่ 3 ลบโหนดในอะเรย์ Temp ที่ได้นำไปรวมเพื่อสร้างโหนดพ่อแม่ใหม่

ขั้นตอนที่ 4 รวมโหนดที่เหลือในอะเรย์ Temp กับโหนดพ่อแม่ใหม่ในชุดรายการ IntTemp ด้วยวิธีการรวมชุดข้อมูลแบบบีเอสอาร์ (Merging on BSR) [24] เก็บไว้ในชุดรายการ Temp

ทำซ้ำในขั้นตอนที่ 1 ถึง 4 จนกระทั่งได้โหนดราก

ซึ่งได้แบ่งขั้นตอนการประมวลผลออกเป็น 2 ส่วนใหญ่ๆ คือ

1) ส่วนการเตรียมการ
2) ส่วนการสร้างรหัสบิตแบบฮัฟแมน

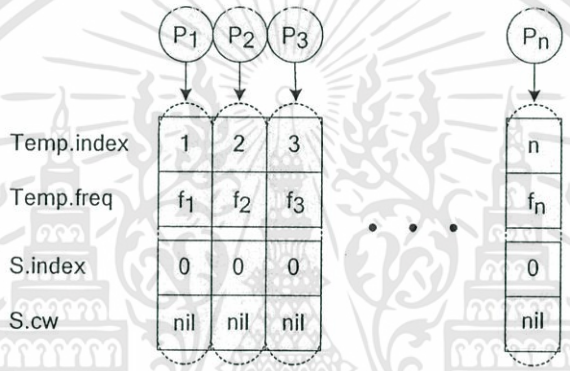
3.2.1 ส่วนการเตรียมการ

เป็นส่วนสำหรับกำหนดค่าเริ่มต้นให้กับตัวแปร n และ c_index ให้มีค่าเท่ากับจำนวนสัญลักษณ์ในข้อมูลนำเข้า จากนั้นกำหนดค่าเริ่มต้นให้แก่สมาชิกในอะเรย์ Temp และอะเรย์ S เพื่อจะนำไปใช้ในการสร้างรหัสบิตในขั้นตอนต่อไป ซึ่งขั้นตอนการเตรียมการมีดังนี้

ขั้นตอนวิธีที่ 3.1 ขั้นตอนวิธีการเตรียมการ

```
n, c_index ← number of symbols
Forall Processors Pi(1 ≤ i ≤ n) pardo
    Temp.index[i] ← i
    Temp.freq[i] ← frequency of symbol[i]
    S.index[i] ← 0
    S.cw[i] ← nil
    Create IntTemp array which its length equal ⌊n/2⌋
End Forall
```

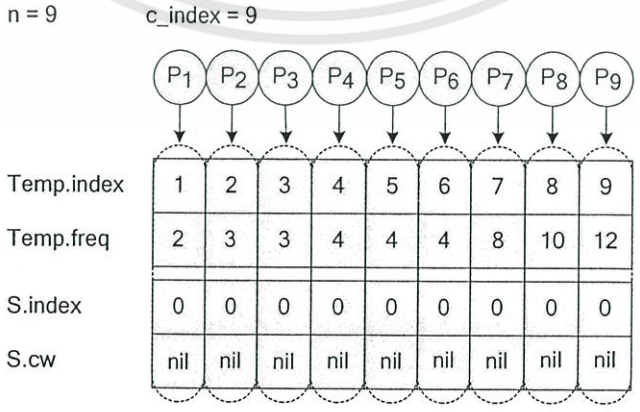
ในขั้นตอนนี้จะกำหนดค่าเริ่มต้นให้กับตัวแปร n และตัวแปร c_index ให้มีค่าเท่ากันคือเท่ากับจำนวนสัญลักษณ์ภายในชุดข้อมูลนำเข้า และกำหนดค่าเริ่มต้นให้กับอะเรย์ด้วยการประมวลผลแบบขนานโดยใช้หน่วยประมวลผล n หน่วย โดยกำหนดค่าให้กับขอบเขต index ของอะเรย์ Temp ด้วยค่าที่ตรงกับลำดับของหน่วยประมวลผลนั้นๆ นั่นคือจะเรียงลำดับจาก 1, 2, 3, ..., n และเก็บค่าความถี่ของการปรากฏสัญลักษณ์แต่ละสัญลักษณ์ในข้อมูลไว้ในขอบเขต freq ของอะเรย์ Temp โดยสมมุติว่าข้อมูลเริ่มต้นในงานวิจัยนี้ได้มีการเรียงลำดับความถี่ของสัญลักษณ์จากน้อยไปมากก่อนการประมวลผล นอกจากนั้นในแต่ละหน่วยประมวลผลจะกำหนดค่า 0 ให้กับขอบเขต index และ ค่า nil ให้กับขอบเขต cw ของทุกสมาชิกในอะเรย์ S ซึ่งมี n สมาชิก จากนั้นสร้างอะเรย์ IntTemp ซึ่งมีขนาด $\lfloor n/2 \rfloor$ สมาชิกโดยมีขอบเขตเช่นเดียวกับอะเรย์ Temp จะใช้เก็บข้อมูลของโหนดพ่อแม่ใหม่ที่ได้ในแต่ละรอบของการประมวลผล ซึ่งแสดงการทำงานดังรูปที่ 3.3



รูปที่ 3.3 การกำหนดค่าเริ่มต้นให้อะเรย์ Temp และ S

<u>ตัวอย่างการเตรียมการ</u>

สมมุติข้อมูลนำเข้าประกอบด้วยสัญลักษณ์ดังนี้ S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8 และ S_9 ซึ่งแต่ละสัญลักษณ์มีค่าความถี่ของการปรากฏสัญลักษณ์ในชุดข้อมูลตามลำดับดังนี้ 2, 3, 3, 4, 4, 4, 8, 10 และ 12 ดังนั้นจะได้ค่าเริ่มต้นในตัวแปร n และตัวแปร c_index เท่ากับ 9 และค่าในแต่ละสมาชิกของอะเรย์ Temp และ S ดังรูปที่ 3.4 โดยใช้หน่วยประมวลผลจำนวน 9 หน่วย



รูปที่ 3.4 ตัวอย่างการกำหนดค่าเริ่มต้นให้กับตัวแปร n, c_index และอะเรย์ Temp, S

3.2.2 ส่วนการสร้างรหัสบิตแบบฮัฟแมน

เป็นการสร้างรหัสบิตแบบฮัฟแมนที่ใช้แทนสัญลักษณ์แต่ละสัญลักษณ์ในข้อมูล ซึ่งมีขั้นตอนวิธีดังนี้

ขั้นตอนวิธีที่ 3.2 ขั้นตอนวิธีการสร้างรหัสบิต

```
While (elements of Temp > 1) do
    P1 compute t ← sum two minimums of Temp.freq
    Forall Processors Pj (j=1 to length(Temp)) pardo
        a ←Σ1| t≤Temp.freq[j] //number of elements which their value no more than t
    End Forall

    Forall Processors Pi (i=1 to ⌊a/2⌋) pardo
        IntTemp.index[i] ← c_index+i
        IntTemp.freq[i] ← Temp.freq[i*2-1] + Temp.freq[i*2]

        //Condition for left sub-tree coding
        ParHuffmanCodes(Temp.index, S.index, IntTemp.index, i*2-1, '0')

        //Condition for right sub-tree coding
        ParHuffmanCodes(Temp.index, S.index, IntTemp.index, i*2, '1')

    End Forall

    c_index ← c_index+⌊a/2⌋
    Delete all elements in Temp that melded.
    Temp ← Merge_BSR(Temp, IntTemp)
    Delete all elements in IntTemp.
End While
-----------------------------------------------------------------------------
ParHuffmanCodes(Temp.index, S.index, IntTemp.index, Pchild, B)
    If (Temp.index[Pchild] ≤ n) // leaf node
        S.index[Temp.index[Pchild]] ← IntTemp.index[i]
        S.cw[Temp.index[Pchild]] ← B
    Else
        Forall Processors Px(x=1 to length(S)) pardo // old parent node
            If (S.index[x] = Temp.index[Pchild])
                S.index[x] ← IntTemp.index[i]
                S.cw[x] ← inserts bit of B in front of its old bits
            End If
        End Forall
    End If
```

ขั้นตอนนี้จะเป็นการสร้างรหัสบิตให้กับแต่ละสัญลักษณ์เมื่อสมาชิกที่แทนโหนดที่เกี่ยวข้องกับสัญลักษณ์นั้นๆ ได้ถูกนำไปรวมกับสมาชิกที่แทนโหนดอื่นเพื่อสร้างโหนดพ่อแม่ใหม่ โดยจะเพิ่มรหัสบิตด้วยบิต '0' หากสมาชิกที่แทนโหนดนั้นๆ ที่ถูกนำไปรวมฝั่งซ้ายของโหนดพ่อแม่ใหม่และจะเพิ่มรหัสบิตด้วยบิต '1' หากสมาชิกที่แทนโหนดนั้นๆ ถูกนำไปรวมในฝั่งขวาของโหนดพ่อแม่ใหม่ โดยเก็บรหัสบิตที่ได้ในแต่ละรอบไว้ในขอบเขต cw ของอะเรย์ S ซึ่งขั้นตอนวิธีการสร้างรหัสบิตประกอบด้วยขั้นตอนวิธีย่อยๆ ตามลำดับ ดังนี้

ขั้นตอนที่ 1 หาจำนวนสมาชิกในอะเรย์ Temp ที่เหมาะสมสำหรับนำไปรวมกันเพื่อสร้างโหนดพ่อแม่ใหม่

ขั้นตอนที่ 2 จับคู่สมาชิกที่เลือกไว้ในอะเรย์ Temp เพื่อสร้างโหนดพ่อแม่ใหม่และสร้างรหัสบิตของแต่ละสมาชิกในอะเรย์ S

ขั้นตอนที่ 3 เตรียมอะเรย์และปรับค่าตัวแปรเพื่อนำไปใช้ในการประมวลผลรอบถัดไป

ขั้นตอนที่ 4 กระทำซ้ำในขั้นตอนที่ 1 ถึง 3 จนกระทั่งมีเพียง 1 สมาชิกในอะเรย์ Temp

ซึ่งแต่ละขั้นตอนมีรายละเอียดดังต่อไปนี้

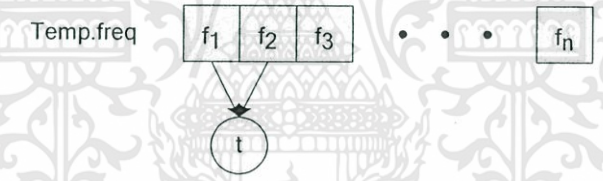
<u>ขั้นตอนที่ 1</u> การหาจำนวนสมาชิกในอะเรย์ Temp เพื่อนำมาสร้างโหนดพ่อแม่ใหม่

ประกอบด้วยขั้นตอนดังนี้

```
t ← sum of two minimum value in Temp.freq
Forall Processors Pj (j=1 to length(Temp)) pardo
    a ← Σ1 | t≤ Temp.freq[j]
End Forall
```

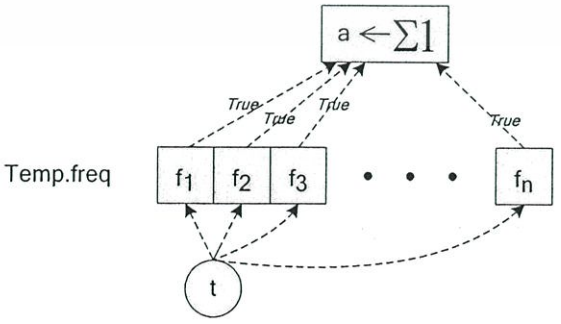
ซึ่งแบ่งออกเป็น 2 กระบวนการดังนี้

1. รวมค่าความถี่ในสมาชิกช่องที่ 1 และ 2 ของอะเรย์ Temp แล้วเก็บความถี่รวมที่ได้ไว้ในตัวแปร t ดังรูปที่ 3.5



รูปที่ 3.5 การคำนวณหาค่าสำหรับตัวแปร t

2. ทุกหน่วยประมวลผลอ่านค่า t พร้อมๆ กันเพื่อนำไปเปรียบเทียบกับค่าใน Temp.freq ที่อยู่ในแต่ละหน่วยประมวลผลพร้อมกัน ซึ่งจำนวนของสมาชิกที่มีค่าใน Temp.freq น้อยกว่าหรือเท่ากับค่าในตัวแปร t จะถูกเก็บไว้ในตัวแปร a โดยใช้ขั้นตอนวิธีการประมวลผลแบบบีเอสอาร์ ดังแสดงการทำงานในรูปที่ 3.6

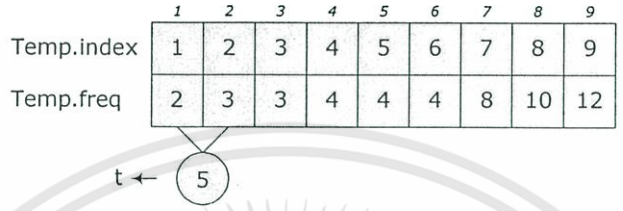


รูปที่ 3.6 การอ่านค่า t โดยทุก P_i พร้อมๆ กันเพื่อหาค่าสำหรับตัวแปร a

ตัวอย่างการหาจำนวนสมาชิกในอะเรย์ Temp เพื่อนำมาสร้างโหนดพ่อแม่ใหม่

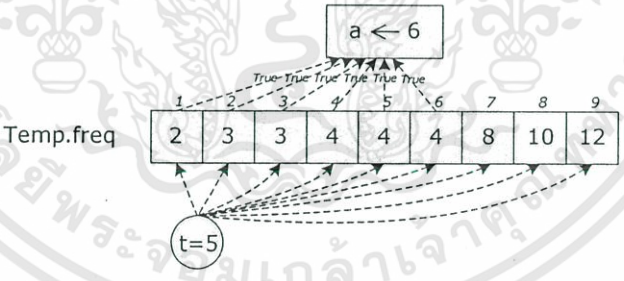
จากตัวอย่างในขั้นเตรียมความพร้อม เมื่อนำข้อมูลในอะเรย์ Temp ที่ได้ มาคำนวณหาจำนวนสมาชิกในอะเรย์ Temp ที่เหมาะสมสำหรับนำไปจับคู่เพื่อสร้างโหนดพ่อแม่ใหม่ซึ่งมีขั้นตอนดังนี้

1. รวมความถี่ของสมาชิกช่องที่ 1 และ 2 ในอะเรย์ Temp เก็บไว้ในตัวแปร t ซึ่งในตัวอย่างนี้จะได้ ค่า t เท่ากับ 5 ดังในรูปที่ 3.7



รูปที่ 3.7 ตัวอย่างการคำนวณหาค่าสำหรับตัวแปร t

2. ทุกหน่วยประมวลผลอ่านค่าในตัวแปร t (= 5) นำไปเปรียบเทียบกับค่าใน Temp.freq ที่อยู่ในแต่ละหน่วยประมวลผลพร้อมกัน ซึ่งจำนวนของสมาชิกที่มีค่าใน Temp.freq น้อยกว่าหรือเท่ากับค่าในตัวแปร t จะถูกเก็บไว้ในตัวแปร a ซึ่งในที่นี้ t มีค่าเท่ากับ 5 ดังนั้นจำนวนสมาชิกของ Temp ที่มีค่าใน Temp.freq ที่น้อยกว่าหรือเท่ากับ 5 ซึ่งมีจำนวน 6 สมาชิก คือ สมาชิกช่องที่ 1, 2, 3, 4, 5 และ 6 ซึ่งมีความถี่ 2, 3, 3, 4, 4 และ 4 ตามลำดับ ดังแสดงการทำงานในรูปที่ 3.8



รูปที่ 3.8 ตัวอย่างการอ่านค่า t โดยทุก P_i พร้อมๆ กันเพื่อหาค่าสำหรับตัวแปร a

ขั้นตอนที่ 2 การจับคู่สมาชิกในอะเรย์ Temp เพื่อสร้างโหนดพ่อแม่ใหม่และสร้างรหัสบิตของแต่ละสมาชิกในอะเรย์ S

ขั้นตอนนี้เป็นขั้นตอนการจับคู่สมาชิกในอะเรย์ Temp จำนวน $\lfloor a/2 \rfloor$ คู่ ซึ่งโหนดพ่อแม่ที่สร้างขึ้นใหม่จะถูกกำหนดหมายเลขประจำโหนดด้วยตัวแปร c_index โดยจะเก็บหมายเลขประจำโหนดและค่าความถี่ของโหนดพ่อแม่ใหม่ไว้ในขอบเขต index และ freq ของอะเรย์ IntTemp

ตามลำดับ และสร้างรหัสบิตที่ใช้แทนสัญลักษณ์แต่ละสัญลักษณ์เก็บไว้ในขอบเขต cw ของอะเรย์ S โดยขั้นตอนนี้มีการประมวลผลแบบขนาน ดังนั้นการสร้างโหนดพ่อแม่ใหม่ การเก็บข้อมูลของโหนดพ่อแม่ใหม่และการสร้างรหัสบิตจะถูกประมวลผลพร้อมกัน ซึ่งแบ่งขั้นตอนออกเป็นขั้นตอนย่อยๆ ตามลำดับดังนี้

1. จับคู่เพื่อสร้างโหนดพ่อแม่ใหม่
2. สร้างรหัสบิตให้กับสมาชิกที่แทนโหนดลูกในฝั่งซ้าย
3. สร้างรหัสบิตให้กับสมาชิกที่แทนโหนดลูกในฝั่งขวา

ในแต่ละขั้นตอนย่อยมีรายละเอียดดังนี้

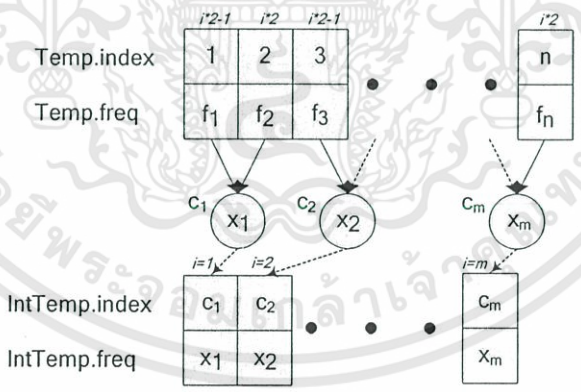
2.1) การจับคู่เพื่อสร้างโหนดพ่อแม่ใหม่

ประกอบด้วยขั้นตอนดังนี้

```
IntTemp.freq[i] ← Temp.freq[i*2-1] + Temp.freq[i*2]
IntTemp.index[i] ← c_index+i
```

1. รวมค่า Temp.freq ในช่อง i*2-1 กับ i*2 แล้วเก็บความถี่รวมซึ่งเป็นความถี่ของโหนดพ่อแม่ใหม่ไว้ใน IntTemp.freq ช่องที่ i ซึ่งตรงกับลำดับการสร้างโหนดพ่อแม่ใหม่

2. กำหนดหมายเลขประจำโหนดให้กับโหนดพ่อแม่ใหม่ โดยการนำค่า c_index บวกด้วยลำดับการสร้างโหนดพ่อแม่ใหม่ แล้วเก็บไว้ใน IntTemp.index ช่อง i ดังแสดงในรูปที่ 3.9

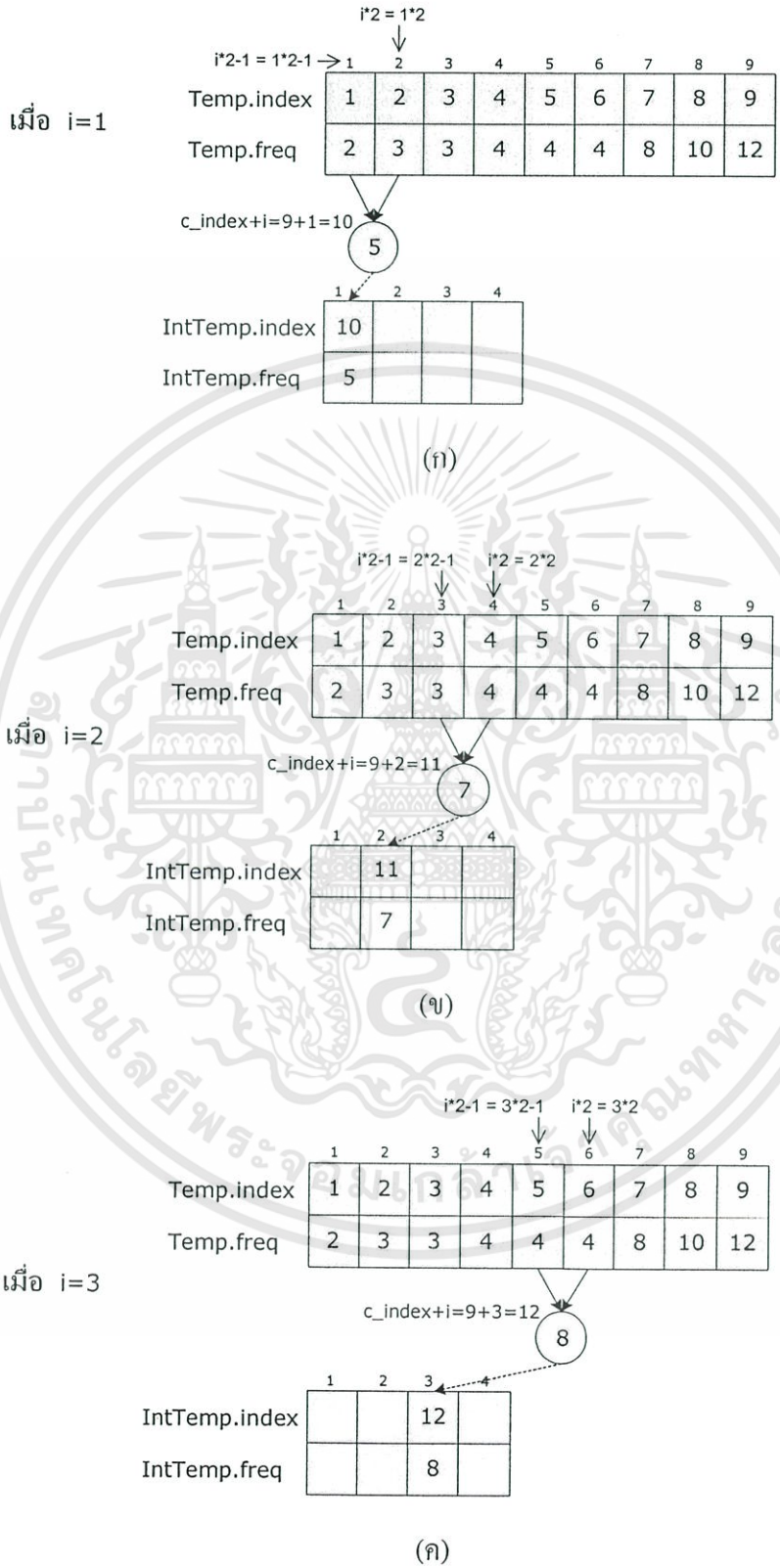


รูปที่ 3.9 การสร้างโหนดพ่อแม่ใหม่ โดยเก็บข้อมูลไว้ในอะเรย์ IntTemp

<u>ตัวอย่างการสร้างโหนดพ่อแม่ใหม่</u>

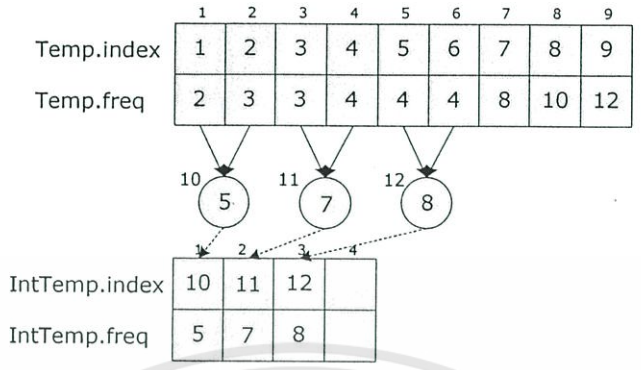
จากตัวอย่างในขั้นตอนการหาจำนวนสมาชิกเพื่อนำมาสร้างโหนดพ่อแม่ได้จำนวนสมาชิกของ Temp ที่มีค่าใน Temp.freq น้อยกว่าหรือเท่ากับค่า t (= 5) จำนวน 6 สมาชิก ทำให้สามารถจับคู่สมาชิกในอะเรย์ Temp เพื่อสร้างโหนดพ่อแม่ใหม่ได้ $\lfloor a/2 \rfloor = \lfloor 6/2 \rfloor = 3$ คู่ โดยให้ i คือ

ลำดับการจับคู่ ซึ่งจะมีการรวมค่าความถี่และกำหนดหมายเลขประจำโหนดพ่อแม่ใหม่ตามลำดับ(i) พร้อมทั้งเก็บข้อมูลดังกล่าวไว้ในอะเรย์ IntTemp ตามลำดับของการจับคู่ ดังรูปที่ 3.10

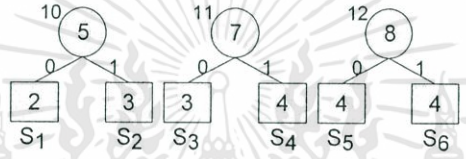


รูปที่ 3.10 การสร้างโหนดพ่อแม่ใหม่ตามลำดับที่ i (ก) เมื่อ i = 1 (ข) เมื่อ i = 2 (ค) เมื่อ i = 3

การจับคู่เพื่อสร้างโหนดพ่อแม่ใหม่ทั้ง 3 คู่ได้ประมวลผลแบบขนาน ดังนั้นจึงได้ผลดังรูปที่ 3.11 (ก) ซึ่งเมื่อเปรียบเทียบกับมุมมองต้นไม้ฮัฟแมนจะได้ดังรูปที่ 3.11 (ข)



(ก)



(ข)

รูปที่ 3.11 ตัวอย่างการจับคู่สมาชิกในอะเรย์ Temp จำนวน 3 คู่ในแบบขนาน (ก) มุมมองโครงสร้างอะเรย์ (ข) มุมมองต้นไม้ฮัฟแมน

2.2) การสร้างรหัสบิตให้กับสมาชิกที่แทนโหนดลูกในฝั่งซ้าย

ประกอบด้วยขั้นตอนดังนี้ เมื่อเรียกใช้ฟังก์ชัน ParHuffmanCodes ที่ Pchild = i*2-1 และ B = '0'

```
If (Temp.index[i*2-1] ≤ n)
    S.index[Temp.index[i*2-1]] ← IntTemp.index[i]
    S.cw[Temp.index[i*2-1]] ← '0'
Else
    Forall Processors Px(x=1 to length(S)) pardo
        If (S.index[x] = Temp.index[i*2-1])
            S.index[x] ← IntTemp.index[i]
            S.cw[x] ← inserts '0' in front of its old bits
        End If
    End Forall
End If
```

โดยในขั้นตอนนี้จะแบ่งการประมวลผลออกเป็น 2 ส่วนคือ

ส่วนที่ 1 คือ ส่วนที่กำหนดรหัสบิตเริ่มต้นให้กับสมาชิกของอะเรย์ S เมื่อสมาชิกของอะเรย์ Temp ถูกนำไปรวมในฝั่งซ้ายเป็นสมาชิกที่แทนโหนดใบหรือสัญลักษณ์

ส่วนที่ 2 คือ ส่วนที่สร้างรหัสบิตเพิ่มให้กับสมาชิกในอะเรย์ S เมื่อสมาชิกในอะเรย์ Temp ถูกนำไปรวมในฝั่งซ้ายเป็นสมาชิกที่แทนโหนดพ่อแม่เดิม

การเลือกประมวลผลในส่วนใดส่วนหนึ่งของขั้นตอนนี้จะกระทำโดยการตรวจสอบหมายเลขประจำโหนดของสมาชิกที่นำมารวมในฝั่งซ้าย (Temp.index[i*2-1]) ว่ามีค่าน้อยกว่าหรือเท่ากับ n หรือไม่ ซึ่งหากเงื่อนไขเป็นจริงแสดงว่าสมาชิกช่องนั้นเป็นสมาชิกที่แทนโหนดใบ จึงจะไปประมวลผลในส่วนที่ 1 แต่หากเงื่อนไขเป็นเท็จแสดงว่าสมาชิกที่นำไปรวมเพื่อสร้างโหนดพ่อแม่ใหม่นั้นเป็นสมาชิกที่แทนโหนดพ่อแม่ในรอบการประมวลผลก่อนหน้า จึงเป็นการประมวลผลในส่วนที่ 2

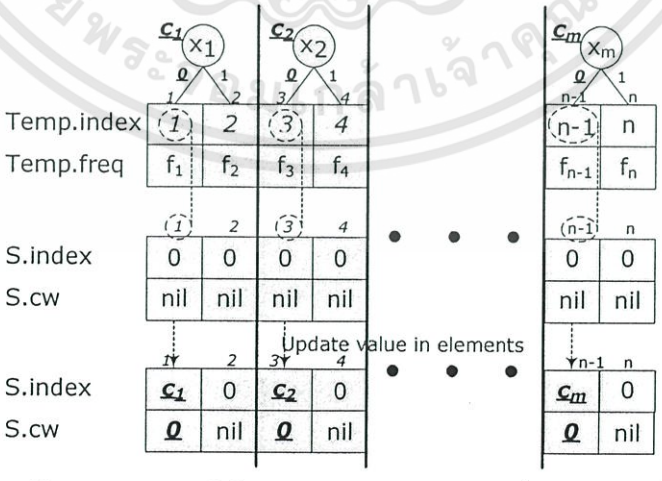
รายละเอียดการประมวลผลในแต่ละส่วนมีดังนี้

<u>ส่วนที่ 1</u> กำหนดรหัสบิตเริ่มต้นให้กับสมาชิกในอะเรย์ S เมื่อสมาชิกในอะเรย์ Temp ที่นำไปรวมในฝั่งซ้ายเป็นสมาชิกที่แทนโหนดใบหรือสัญลักษณ์

หากสมาชิกที่นำมารวมกันเพื่อสร้างโหนดพ่อแม่ใหม่เป็นสมาชิกที่แทนโหนดใบหรือสัญลักษณ์ ส่วนนี้จะทำการสร้างรหัสบิตเริ่มต้นให้กับสมาชิกในอะเรย์ S ที่แทนสัญลักษณ์ที่ถูกนำมารวมและทำการเก็บหมายเลขประจำโหนดของโหนดพ่อแม่ใหม่ที่สร้างขึ้น ตามขั้นตอนดังนี้

$$S.index[Temp.index[i*2-1]] \leftarrow IntTemp.index[i]$$
$$S.cw[Temp.index[i*2-1]] \leftarrow \text{'0'}$$

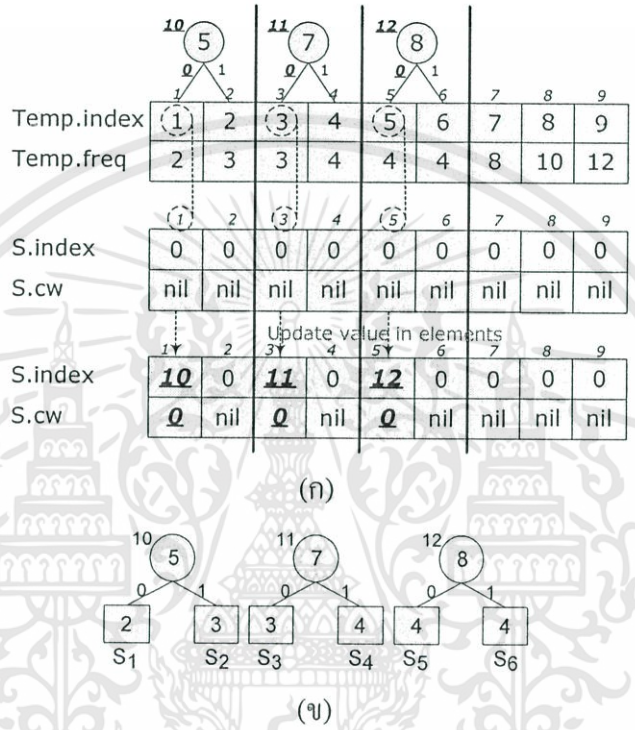
โดยจะทำการเก็บหมายเลขประจำโหนดพ่อแม่ใหม่ ซึ่งถูกเก็บไว้ใน IntTemp.index ช่องที่ i มาไว้ใน S.index ของสมาชิกที่แทนสัญลักษณ์หรือสมาชิกช่องที่ตรงกับค่าใน Temp.index ของสมาชิกที่นำมารวมในฝั่งซ้าย และเก็บรหัสบิตเริ่มต้น '0' ไว้ใน S.cw ของสมาชิกนั้นๆ



รูปที่ 3.12 การสร้างรหัสบิต '0' ให้กับแต่ละสมาชิกในอะเรย์ S ในกรณีที่สมาชิกที่แทนโหนดลูกฝั่งซ้ายเป็นโหนดใบหรือสัญลักษณ์

ดังในรูปที่ 3.12 เป็นการสร้างรหัสบิต '0' ให้กับสมาชิกใน S.cw ช่องที่ 1, 3,..., n-1 เพราะสมาชิกดังกล่าวเป็นสมาชิกที่แทนโหนดลูกฝั่งซ้ายของโหนดพ่อแม่ใหม่ และได้เก็บหมายเลขประจำโหนดพ่อแม่ใหม่ของสมาชิกนั้น ซึ่งได้ถูกเก็บใน IntTemp.index ช่องที่ 1, 2, ..., $\lfloor a/2 \rfloor$ มาไว้ใน S.index ของสมาชิกดังกล่าว

<u>ตัวอย่างการสร้างรหัสบิต '0' ให้กับโหนดลูกฝั่งซ้ายที่เป็นโหนดใบหรือสัญลักษณ์</u>

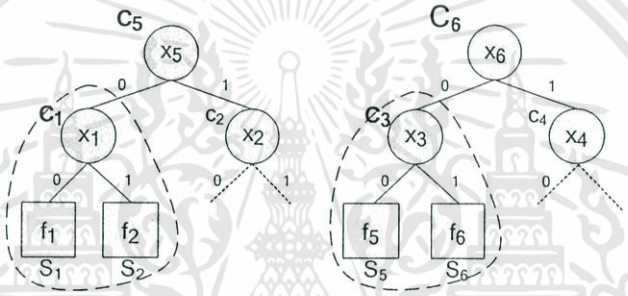


รูปที่ 3.13 ตัวอย่างการสร้างรหัสบิต '0' ให้กับแต่ละสมาชิกในอะเรย์ S กรณีที่สมาชิกที่แทนโหนดลูกฝั่งซ้ายเป็นโหนดใบหรือสัญลักษณ์ (ก) โครงสร้างอะเรย์ (ข) ต้นไม้ฮัฟแมน

จากรูปที่ 3.13 โหนดพ่อแม่หมายเลข 10, 11 และ 12 ที่ถูกสร้างขึ้น มีสมาชิกช่องที่ 1, 3, 5 ของอะเรย์ Temp เป็นโหนดลูกฝั่งซ้าย ซึ่งหมายเลขประจำโหนด (Temp.index) ของสมาชิกช่องดังกล่าวมีค่าน้อยกว่า 9 (n=9) แสดงว่าสมาชิกเหล่านั้นเป็นสมาชิกที่แทนโหนดใบ ดังนั้นจึงทำการปรับค่าใน S.index ของสมาชิกช่องที่ 1, 3 และ 5 ให้เป็น 10, 11 และ 12 ตามหมายเลขประจำโหนดพ่อแม่ตามลำดับพร้อมทั้งกำหนดรหัสบิตใน S.cw ให้กับทั้ง 3 สมาชิกเป็น '0'

<u>ส่วนที่ 2</u> สร้างรหัสบิตเพิ่มเติมให้กับสมาชิกในอะเรย์ S เมื่อสมาชิกในอะเรย์ Temp ที่นำไปรวมในฝั่งซ้ายเป็นสมาชิกที่แทนโหนดพ่อแม่เดิม

ในส่วนนี้จะทำการเพิ่มรหัสบิต '0' ให้กับโหนดใบหรือสัญลักษณ์ที่อยู่ภายใต้โหนดพ่อแม่ที่ถูกนำไปรวมไว้เป็นโหนดลูกทางฝั่งซ้ายของโหนดพ่อแม่ใหม่ ดังรูปที่ 3.14 โหนดพ่อแม่หมายเลข C_1 และ C_3 ถูกนำมารวมเป็นโหนดลูกฝั่งซ้ายของโหนดพ่อแม่ใหม่ (C_5 และ C_6) ดังนั้น สัญลักษณ์ S_1, S_2, S_5 และ S_6 จะต้องถูกเพิ่มบิต '0' ด้านหน้าบิตเดิมที่มีอยู่ นั่นคือ เดิมรหัสบิตของ S_1 เป็น '0' และรหัสบิตของสัญลักษณ์ S_2 เป็น '1' เมื่อโหนด C_1 ถูกนำไปรวมเป็นโหนดลูกฝั่งซ้ายจะทำให้รหัสบิตของสัญลักษณ์ S_1 ถูกเพิ่มเป็น "00" และรหัสบิตของสัญลักษณ์ S_2 ถูกเพิ่มเป็น "01" เช่นเดียวกัน รหัสบิตของสัญลักษณ์ S_5 และ S_6 จะถูกเพิ่มเป็น "00" และ "01" ตามลำดับ

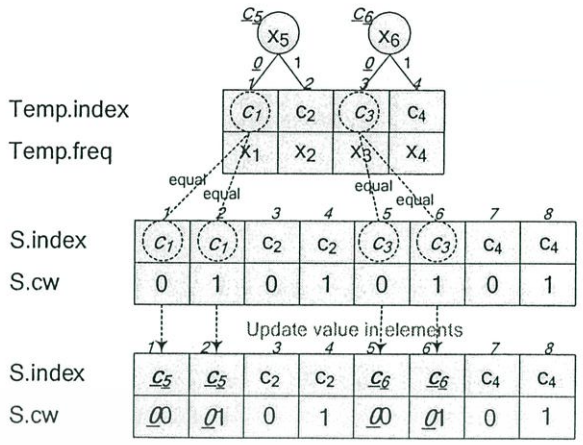


รูปที่ 3.14 แสดงโครงสร้างต้นไม้เมื่อนำโหนดพ่อแม่เดิมมาเป็นโหนดลูกฝั่งซ้ายของโหนดพ่อแม่ใหม่

จากรูปที่ 3.14 เมื่อนำมาประมวลผลด้วยขั้นตอนวิธีที่ได้นำเสนอจะมีขั้นตอนดังนี้

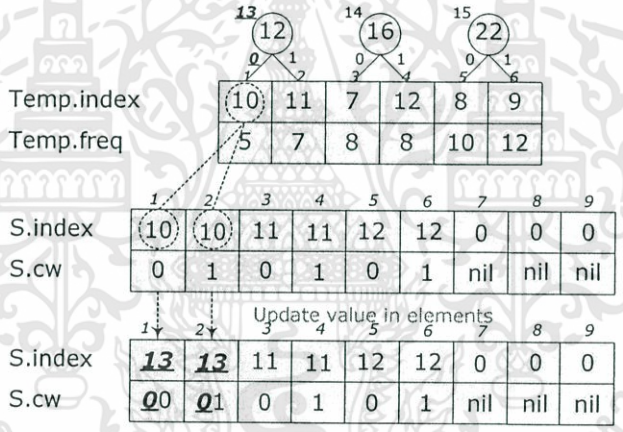
```
Forall Processors Px(x=1 to length(S)) pardo
    If (S.index[x] = Temp.index[i*2-1])
        S.index[x] ← IntTemp.index[i]
        S.cw[x] ← inserts '0' in front of its old bits
    End If
End Forall
```

โดยในขั้นตอนนี้จะทำการเปรียบเทียบหมายเลขประจำโหนด (Temp.index) ของสมาชิกที่แทนโหนดลูกในฝั่งซ้าย ([i*2-1]) กับ หมายเลขประจำโหนด (S.index) ของทุกสมาชิกในอะเรย์ S เพื่อหาว่า สมาชิกในอะเรย์ S ช่องใดเป็นสมาชิกที่แทนโหนดที่อยู่ภายใต้โหนดพ่อแม่ที่เดิมถูกนำไปรวมเพื่อสร้างโหนดพ่อแม่ใหม่ ซึ่งหากหมายเลขตรงกันแสดงว่าสมาชิกของอะเรย์ S ช่องเหล่านั้นเป็นสมาชิกที่แทนโหนดใบที่อยู่ภายใต้โหนดพ่อแม่ที่ถูกนำไปรวมในฝั่งซ้ายของโหนดพ่อแม่ใหม่ปัจจุบัน จากนั้นจะทำการเก็บหมายเลขประจำโหนดของโหนดพ่อแม่ใหม่แทนที่หมายเลขประจำโหนดพ่อแม่เดิมใน S.index ของสมาชิกช่องเหล่านั้น และเพิ่มบิต '0' ไว้ด้านหน้าบิตเดิมที่มีอยู่ใน S.cw ของสมาชิกเหล่านั้นด้วย ดังในรูปที่ 3.15

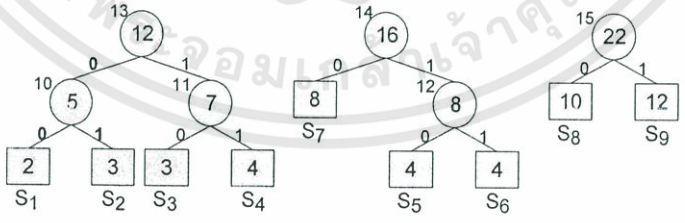


รูปที่ 3.15 การสร้างรหัสบิต '0' ให้กับแต่ละสมาชิกในอะเรย์ S กรณีที่สมาชิกที่แทนโหนดลูกฝั่งซ้ายเป็นโหนดพ่อแม่เดิม

<u>ตัวอย่างการสร้างรหัสบิตในกรณีที่โหนดลูกฝั่งซ้ายเป็นโหนดพ่อแม่เดิม</u>



(ก)



(ข)

รูปที่ 3.16 ตัวอย่างการสร้างรหัสบิต '0' ให้กับแต่ละสมาชิกในอะเรย์ S กรณีที่สมาชิกที่แทนโหนดลูกฝั่งซ้ายเป็นโหนดพ่อแม่เดิม (ก) โครงสร้างอะเรย์ (ข) ต้นไม้ฮัฟแมน

จากตัวอย่างในรูปที่ 3.16 มีโหนดพ่อแม่ที่ถูกสร้างขึ้นใหม่คือโหนดหมายเลข 13, 14 และ 15 ซึ่งโหนดพ่อแม่หมายเลข 13 มีโหนดลูกฝั่งซ้ายเป็นโหนดหมายเลข 10 ซึ่งมากกว่า 9 (n=9) แสดงว่าโหนดลูกนี้เป็นโหนดพ่อแม่ในรอบประมวลผลก่อนหน้า ซึ่งจะต้องมีโหนดลูกที่เป็นโหนดใบอยู่ ดังนั้นเมื่อจะเพิ่มบิตจะต้องทำการเพิ่มบิตให้กับโหนดใบทั้งหมดที่อยู่ภายใต้โหนดพ่อแม่หมายเลข 10 นี้ โดยการเปรียบเทียบหมายเลขประจำโหนด 10 (Temp.index) กับ S.index ทุกสมาชิกเพื่อหาว่ามีสมาชิกใดในอะเรย์ S ที่มีโหนดพ่อแม่เป็นโหนดหมายเลข 10 ซึ่งในที่นี้ได้สมาชิกช่องที่ 1 และ 2 ซึ่งแสดงว่าสองสมาชิกนี้เป็นโหนดใบที่อยู่ภายใต้โหนดพ่อแม่หมายเลข 10 จึงทำการเปลี่ยนหมายเลขประจำโหนดพ่อแม่เป็น 13 และเพิ่มบิต '0' ให้กับทั้ง 2 สมาชิกนี้พร้อมกัน

2.3) การสร้างรหัสบิตให้กับสมาชิกที่แทนโหนดลูกในฝั่งขวา

ประกอบด้วยขั้นตอนดังนี้ เมื่อเรียกใช้ฟังก์ชัน ParHuffmanCodes ที่ Pchild=i*2 และ B = '1'

```
If (Temp.index[i*2] ≤ n)
  S.index[Temp.index[i*2]] ← IntTemp.index[i]
  S.cw[Temp.index[i*2]] ← '1'
Else
  Forall Processors Px(x=1 to length(S)) pardo
  If (S.index[x] = Temp.index[i*2])
        S.index[x] ← IntTemp.index[i]
        S.cw[x] ← inserts '1' in front of its old bits
    End If
  End Forall
End If
```

โดยในขั้นตอนนี้จะแบ่งการประมวลผลออกเป็น 2 ส่วนคือ

ส่วนที่ 1 คือ ส่วนที่กำหนดรหัสบิตเริ่มต้นให้กับสมาชิกในอะเรย์ S เมื่อสมาชิกในอะเรย์ Temp ที่นำไปรวมในฝั่งขวาเป็นสมาชิกที่แทนโหนดใบหรือสัญลักษณ์

ส่วนที่ 2 คือ ส่วนที่สร้างรหัสบิตเพิ่มเติมให้กับสมาชิกในอะเรย์ S เมื่อสมาชิกในอะเรย์ Temp ที่นำไปรวมในฝั่งขวาเป็นสมาชิกที่แทนโหนดพ่อแม่เดิม

การเลือกประมวลผลในส่วนใดส่วนหนึ่งของขั้นตอนนี้จะกระทำโดยการตรวจสอบหมายเลขประจำโหนดของสมาชิกที่นำมารวมในฝั่งขวา (Temp.index[i*2]) ว่ามีค่าน้อยกว่าหรือเท่ากับ n หรือไม่ ซึ่งหากเงื่อนไขเป็นจริงแสดงว่าสมาชิกช่องนั้นเป็นสมาชิกที่แทนโหนดใบ ซึ่งจะประมวลผลด้วยส่วนที่ 1 แต่หากเงื่อนไขเป็นเท็จแสดงว่าสมาชิกที่นำไปรวมเพื่อสร้างโหนดพ่อแม่ใหม่นั้นเป็นสมาชิกที่แทนโหนดพ่อแม่ในรอบการประมวลผลก่อนหน้า ซึ่งจะเป็นการประมวลผลด้วยส่วนที่ 2

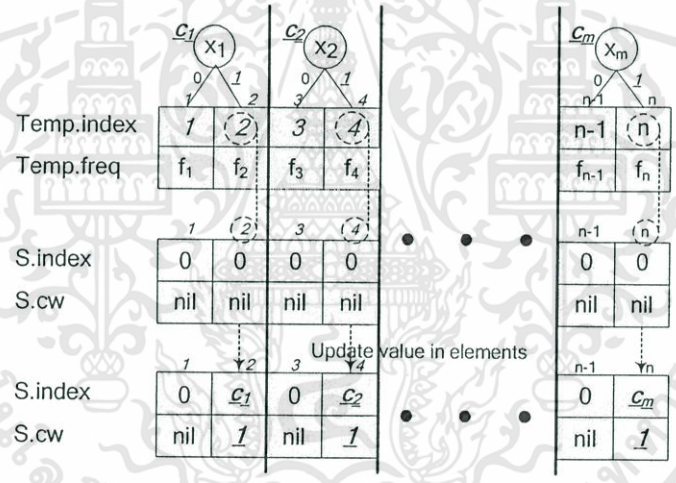
ซึ่งรายละเอียดการประมวลผลในแต่ละส่วนมีดังนี้

<u>ส่วนที่ 1</u> กำหนดรหัสบิตเริ่มต้นให้กับสมาชิกในอะเรย์ S เมื่อสมาชิกในอะเรย์ Temp ที่ได้นำไปรวมในฝั่งขวาเป็นสมาชิกที่แทนโหนดใบหรือสัญลักษณ์

หากสมาชิกที่นำมารวมกันเพื่อสร้างโหนดพ่อแม่ใหม่เป็นสมาชิกที่แทนโหนดใบ ขั้นตอนนี้จะทำการสร้างรหัสบิตเริ่มต้นให้กับสมาชิกในอะเรย์ S ที่แทนสัญลักษณ์ที่ถูกนำมารวมและทำการเก็บหมายเลขประจำโหนดของโหนดพ่อแม่ใหม่ของโหนดนั้นๆ ตามขั้นตอนดังนี้

$$S.index[Temp.index[i*2]] \leftarrow IntTemp.index[i]$$
$$S.cw[Temp.index[i*2]] \leftarrow \text{'1'}$$

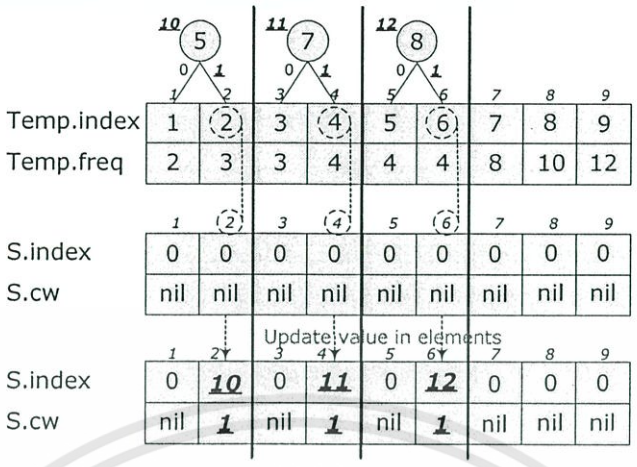
โดยจะทำการเก็บหมายเลขประจำโหนดพ่อแม่ใหม่ ซึ่งได้ถูกเก็บไว้ใน IntTemp.index ช่องที่ i มาไว้ใน S.index ของสมาชิกที่แทนสัญลักษณ์หรือสมาชิกช่องที่ตรงกับค่าใน Temp.index ของสมาชิกที่นำมารวม และแทนที่ค่าใน S.cw ของสมาชิกนั้นด้วยบิต '1'



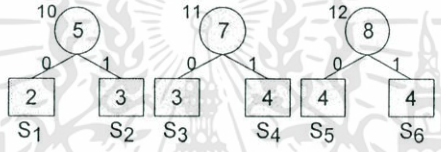
รูปที่ 3.17 การสร้างรหัสบิต '1' ให้กับแต่ละสมาชิกในอะเรย์ S กรณีที่สมาชิกที่แทนโหนดลูกฝั่งขวาเป็นโหนดใบหรือสัญลักษณ์

จากรูปที่ 3.17 เป็นการสร้างรหัสบิต '1' ให้กับสมาชิกใน S.cw ช่องที่ 2, 4,..., n เพราะสมาชิกดังกล่าวเป็นสมาชิกที่แทนโหนดลูกฝั่งขวาของโหนดพ่อแม่ใหม่ และเก็บหมายเลขประจำโหนดพ่อแม่ของสมาชิกนั้น ซึ่งถูกเก็บไว้ใน IntTemp.index ช่องที่ 1, 2, 3, ..., $\lfloor a/2 \rfloor$ มาเก็บไว้ใน S.index ของสมาชิกดังกล่าว

ตัวอย่างการสร้างรหัสบิต '1' ให้กับโหนดลูกฝั่งขวาที่เป็นโหนดใบหรือสัญลักษณ์



(ก)



(ข)

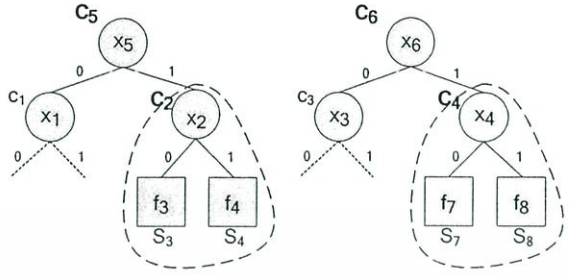
รูปที่ 3.18 ตัวอย่างการสร้างรหัสบิต '1' ให้กับแต่ละสมาชิกในอะเรย์ S กรณีที่สมาชิกที่แทนโหนดลูกฝั่งขวาเป็นโหนดใบหรือสัญลักษณ์ (ก) โครงสร้างอะเรย์ (ข) ต้นไม้ฮัฟแมน

จากรูปที่ 3.18 โหนดพ่อแม่หมายเลข 10, 11 และ 12 ที่ถูกสร้างขึ้น มีสมาชิกช่องที่ 2, 4, 6 ของอะเรย์ Temp เป็นโหนดลูกฝั่งขวา ซึ่งหมายเลขประจำโหนด (Temp.index) ของสมาชิกช่องดังกล่าวมีค่าน้อยกว่า 9 (n=9) แสดงว่าสมาชิกเหล่านั้นเป็นสมาชิกที่แทนโหนดใบ ดังนั้นจึงทำการปรับค่าใน S.index ของสมาชิกช่องที่ 2, 4 และ 6 ให้เท่ากับ 10, 11 และ 12 ตามหมายเลขประจำของโหนดพ่อแม่ตามลำดับพร้อมทั้งกำหนดรหัสบิตใน S.cw ให้กับทั้ง 3 ช่องเป็น '1'

ส่วนที่ 2 สร้างรหัสบิตเพิ่มเติมให้กับสมาชิกในอะเรย์ S เมื่อสมาชิกในอะเรย์ Temp ที่นำไปรวมในฝั่งขวาเป็นสมาชิกที่แทนโหนดพ่อแม่เดิม

ในส่วนนี้จะทำการเพิ่มรหัสบิต '1' ให้กับโหนดใบหรือสัญลักษณ์ทุกโหนดที่อยู่ภายใต้โหนดพ่อแม่ที่ถูกนำไปรวมไว้เป็นโหนดลูกทางฝั่งขวาของโหนดพ่อแม่ใหม่ ดังรูปที่ 3.19 โหนดพ่อแม่หมายเลข C_2 และ C_4 ถูกนำมารวมเป็นโหนดลูกฝั่งขวาของโหนดพ่อแม่ใหม่ (C_5 และ C_6) ดังนั้นสัญลักษณ์ S_3, S_4, S_7 และ S_8 จะต้องถูกเพิ่มบิต '1' ด้านหน้าบิตเดิมที่มีอยู่ นั่นคือจากเดิมรหัสบิตที่แทนสัญลักษณ์ S_3 คือ '0' และรหัสบิตที่แทนสัญลักษณ์ S_4 คือ '1' เมื่อโหนด C_2 ซึ่งเป็น

โหนดพ่อแม่ของทั้ง 2 สัญลักษณ์นี้ถูกนำไปรวมเป็นโหนดลูกฝั่งขวาของโหนดพ่อแม่ใหม่ (C5) จะทำให้รหัสบิตของ S_3 ถูกเพิ่มเป็น “10” และรหัสบิตของ S_4 ถูกเพิ่มเป็น “11” เช่นเดียวกัน รหัสบิตของ S_7 ก็จะถูกเพิ่มเป็น “10” และ รหัสบิตที่ของ S_8 ก็จะถูกเพิ่มเป็น “11” เช่นกัน

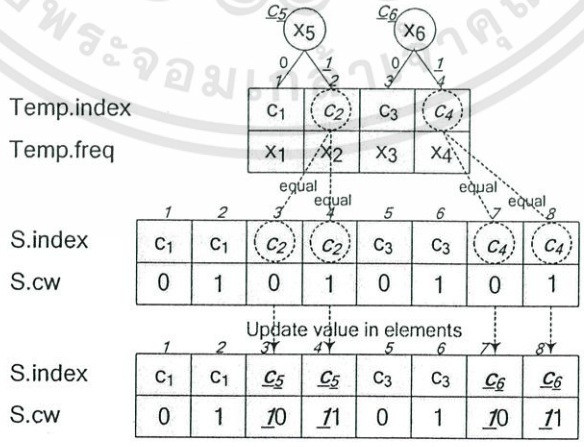


รูปที่ 3.19 แสดงโครงสร้างต้นไม้เมื่อนำโหนดพ่อแม่เดิมมาเป็นโหนดลูกฝั่งขวาของโหนดพ่อแม่ใหม่

จากรูปที่ 3.19 เมื่อนำมาประมวลผลด้วยขั้นตอนวิธีที่ได้นำเสนอจะมีขั้นตอนดังนี้

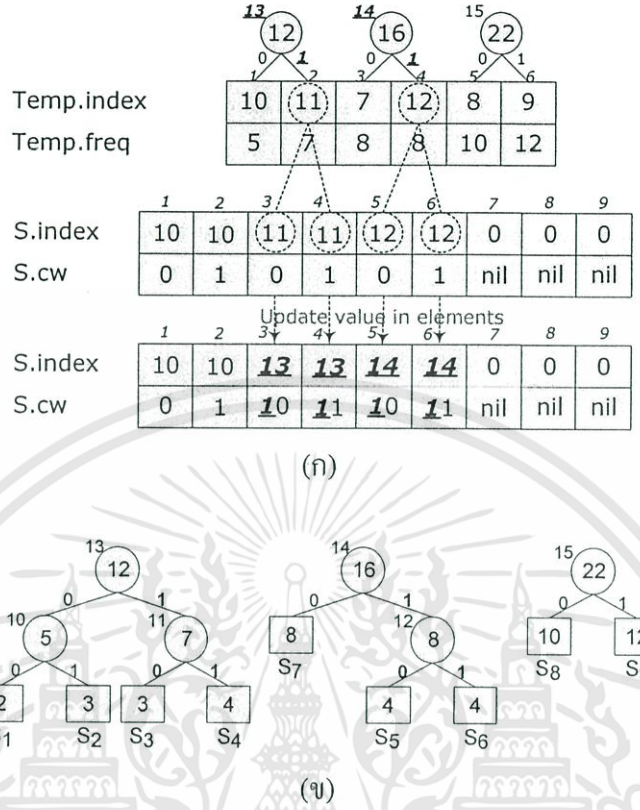
```
Forall Processors Px(x=1 to length(S)) pardo
    If (S.index[x] = Temp.index[i*2])
        S.index[x] ← IntTemp.index[i]
        S.cw[x] ← insert '1' in front of its old bits
    End If
End Forall
```

โดยในขั้นตอนนี้จะทำการเปรียบเทียบหมายเลขประจำโหนด (Temp.index) ของสมาชิกที่แทนโหนดลูกในฝั่งขวา ([i*2]) กับ หมายเลขประจำโหนด (S.index) ของทุกสมาชิกในอะเรย์ S ซึ่งหากหมายเลขตรงกันแสดงว่าสมาชิกของอะเรย์ S เหล่านั้นเป็นสมาชิกที่แทนโหนดใบภายใต้โหนดพ่อแม่เดิมที่ถูกนำไปรวมในฝั่งขวาของโหนดพ่อแม่ใหม่ปัจจุบัน จึงเก็บหมายเลขประจำโหนดพ่อแม่ใหม่แทนที่หมายเลขประจำโหนดพ่อแม่เดิมใน S.index ของสมาชิกเหล่านั้น และเพิ่มบิต ‘1’ ไว้ด้านหน้าบิตเดิมที่มีอยู่ใน S.cw ของสมาชิกเหล่านั้น ดังรูปที่ 3.20



รูปที่ 3.20 การสร้างรหัสบิต ‘1’ ให้กับแต่ละสมาชิกในอะเรย์ S กรณีที่สมาชิกที่แทนโหนดลูกฝั่งขวาเป็นโหนดพ่อแม่เดิม

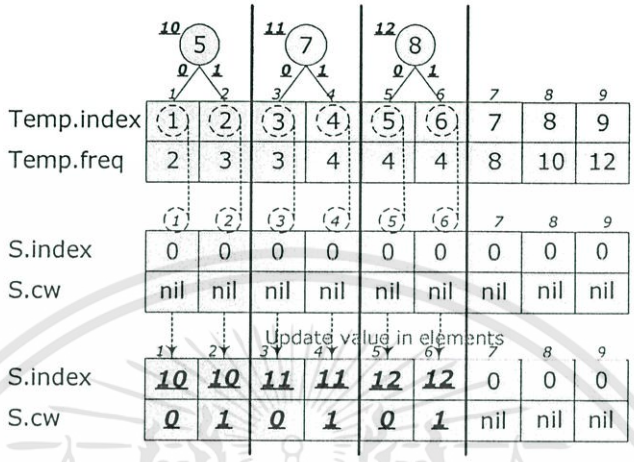
ตัวอย่างการสร้างรหัสบิตในกรณีที่โหนดลูกฝั่งขวาเป็นโหนดพ่อแม่เดิม



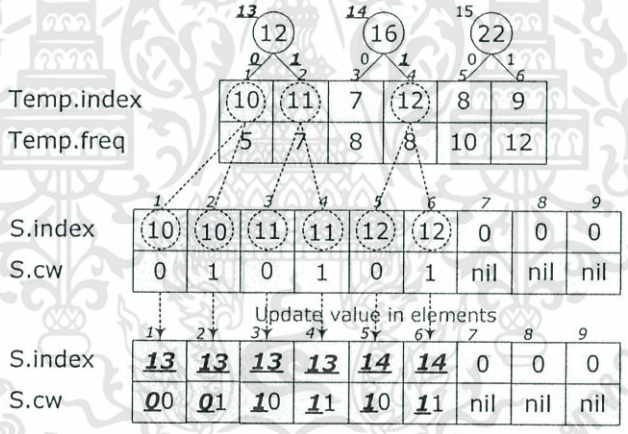
รูปที่ 3.21 ตัวอย่างการสร้างรหัสบิต '1' ให้กับแต่ละสมาชิกในอะเรย์ S กรณีที่สมาชิกที่แทนโหนดลูกฝั่งขวาเป็นโหนดพ่อแม่เดิม (ก) โครงสร้างอะเรย์ (ข) ต้นไม้ฮัฟแมน

ตัวอย่างในรูปที่ 3.21 มีโหนดพ่อแม่ที่ถูกสร้างขึ้นใหม่คือโหนดหมายเลข 13, 14 และ 15 ซึ่งโหนดพ่อแม่หมายเลข 13 และ 14 มีโหนดลูกฝั่งขวาเป็นโหนดหมายเลข 11 และ 12 ตามลำดับ ซึ่งมากกว่า 9 (n=9) แสดงว่าโหนดลูก 2 โหนดนี้เป็นโหนดพ่อแม่ในรอบการประมวลผลก่อนหน้า ซึ่งต้องมีโหนดใบที่อยู่ภายในกิ่ง ดังนั้นเมื่อจะเพิ่มบิตจะต้องทำการเพิ่มบิตให้กับโหนดใบทั้งหมดที่อยู่ภายใต้โหนดพ่อแม่หมายเลข 11 และ 12 นี้ โดยเปรียบเทียบหมายเลขประจำโหนด 11 และ 12 (Temp.index) กับค่าใน S.index ทุกสมาชิกเพื่อหาว่ามีสมาชิกใดที่มีหมายเลขประจำโหนดพ่อแม่เดิมเป็น 11 หรือ 12 ในที่นี้สมาชิกช่องที่ 3 และ 4 มีหมายเลขโหนดพ่อแม่เดิม(S.index) เป็น 11 แสดงว่า 2 สมาชิกนี้เป็นโหนดใบที่อยู่ภายใต้โหนดพ่อแม่หมายเลข 11 และสมาชิกช่องที่ 5 และ 6 ของอะเรย์ S มีหมายเลขพ่อแม่เดิม (S.index) เป็น 12 แสดงว่า 2 สมาชิกนี้เป็นโหนดใบภายใต้โหนดพ่อแม่หมายเลข 12 จึงทำการเปลี่ยนหมายเลขประจำโหนดพ่อแม่ใหม่ให้กับสมาชิกช่องที่ 3 และ 4 เป็น 13 และสมาชิกช่องที่ 5 และ 6 เป็น 14 ตามลำดับจากนั้นเพิ่มบิต '1' ไว้ด้านหน้าบิตเดิมให้กับทั้ง 4 สมาชิกนี้พร้อมกัน

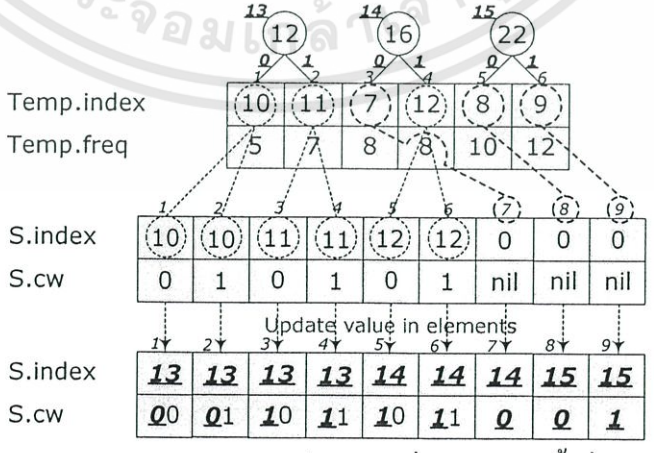
การสร้างรหัสบิตสำหรับโหนดลูกฝั่งซ้ายและฝั่งขวาจะกระทำหลังจากสร้างโหนดพ่อแม่ใหม่ทันที ดังนั้นจะได้ผลการปรับปรุงค่าในอะเรย์ S ในกรณีที่โหนดลูกเป็นโหนดใบในรูปที่ 3.22 และในกรณีที่โหนดลูกเป็นโหนดพ่อแม่เดิมในรูปที่ 3.23 สำหรับในรูปที่ 3.24 เป็นผลการปรับปรุงค่าในอะเรย์ S เมื่อโหนดลูกมีทั้งที่เป็นโหนดใบและโหนดพ่อแม่เดิม



รูปที่ 3.22 ตัวอย่างผลการปรับปรุงค่าในอะเรย์ S กรณีที่โหนดลูกเป็นโหนดใบหรือสัญลักษณ์



รูปที่ 3.23 ตัวอย่างผลการปรับปรุงค่าในอะเรย์ S กรณีที่โหนดลูกเป็นโหนดพ่อแม่เดิม



รูปที่ 3.24 ตัวอย่างผลการปรับปรุงค่าในอะเรย์ S กรณีที่โหนดลูกมีทั้งที่เป็นโหนดใบและโหนดพ่อแม่เดิม

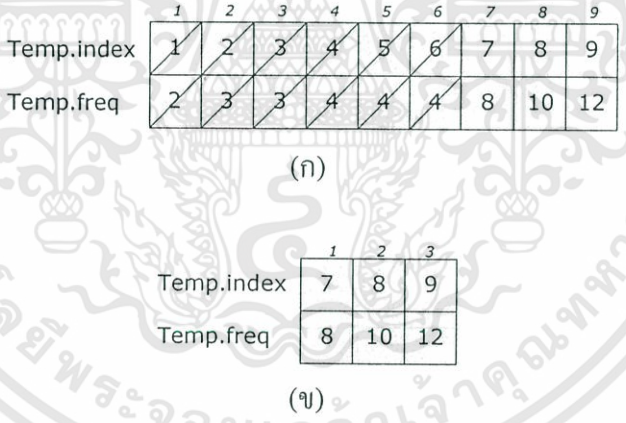
ขั้นตอนที่ 3 การเตรียมอะเรย์และปรับค่าตัวแปร

ขั้นตอนการเตรียมอะเรย์และปรับค่าตัวแปรประกอบด้วยขั้นตอนวิธีดังนี้

```
c_index ← c_index+⌊a / 2⌋
Delete all elements in Temp that melded.
Temp ← Merge_BSR(Temp, IntTemp)
Delete all elements in IntTemp.
```

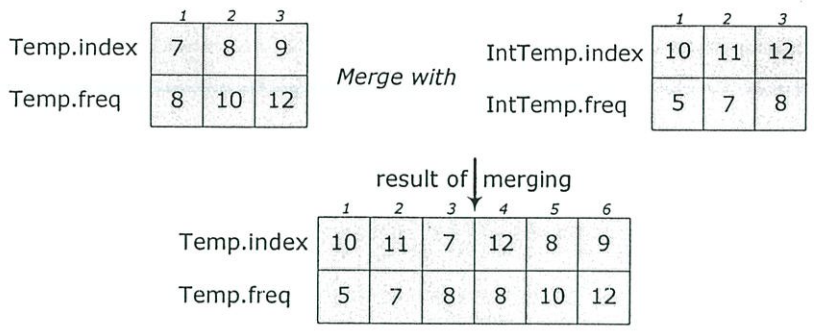
1. ปรับค่าในตัวแปร c_index ซึ่งเป็นตัวแปรสำหรับกำหนดหมายเลขประจำโหนดให้มีค่าเท่ากับหมายเลขประจำโหนดพ่อแม่ล่าสุดที่ถูกสร้างขึ้น ด้วยการนำจำนวนการจับคู่เพื่อสร้างโหนดพ่อแม่ใหม่ในรอบปัจจุบันบวกกับค่า c_index เดิม เช่น ก่อนการประมวลผลมีสัญลักษณ์นำเข้าทั้งหมด 10 สัญลักษณ์ ดังนั้นค่าเริ่มต้นของตัวแปร c_index จะเท่ากับ 10 และเมื่อได้สร้างโหนดพ่อแม่ใหม่จำนวน 3 คู่แล้ว ค่า c_index จะถูกปรับให้เพิ่มค่าเป็น 10+3 = 13

2. ลบสมาชิกในอะเรย์ Temp ที่ได้นำไปรวมเพื่อสร้างโหนดพ่อแม่ใหม่ในรอบปัจจุบันออกจากอะเรย์ ดังในรูปที่ 3.22 สมาชิกในอะเรย์ Temp ช่องที่ 1 ถึง 6 ได้ถูกนำมาจับคู่เพื่อสร้างโหนดพ่อแม่ใหม่ ดังนั้นสมาชิกในช่องที่ 1 ถึง 6 ดังกล่าวจะถูกลบออกจากอะเรย์ Temp ดังแสดงในรูปที่ 3.25



รูปที่ 3.25 (ก) การลบสมาชิกในอะเรย์ Temp (ข) สมาชิกที่เหลือในอะเรย์ Temp

3. รวมสมาชิกที่เหลือในอะเรย์ Temp กับสมาชิกในอะเรย์ IntTemp ด้วยขั้นตอนวิธีการรวมข้อมูลบนการประมวลผลแบบบีเอสอาร์ (Merging on BSR) [24] ดังแสดงการอธิบายไว้ในหัวข้อ 2.6.1 และ 2.6.2 ในบทที่ 2 เพื่อสร้างกลุ่มสมาชิกใหม่โดยผลการรวมสมาชิกจะถูกเก็บไว้ในอะเรย์ Temp ซึ่งจะนำไปประมวลผลในรอบถัดไป โดยกลุ่มสมาชิกใหม่ที่ได้จะมีการจัดเรียงลำดับค่าความถี่จากน้อยไปมาก ดังแสดงในรูปที่ 3.26



รูปที่ 3.26 การรวมสมาชิกที่เหลือของอะเรย์ Temp กับสมาชิกในอะเรย์ IntTemp

4. ลบสมาชิกทั้งหมดในอะเรย์ IntTemp สำหรับรองรับการเก็บข้อมูลของโหนดพ่อแม่ใหม่ที่จะถูกสร้างขึ้นในรอบการประมวลผลถัดไป

บทที่ 4
วิเคราะห์ความซับซ้อนด้านเวลาของขั้นตอนวิธีการที่นำเสนอ

ในบทที่ 4 นี้จะกล่าวถึงการวิเคราะห์ความซับซ้อนด้านเวลาของขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่ได้นำเสนอไปแล้วในบทที่ 3

การวิเคราะห์ความซับซ้อนด้านเวลาของขั้นตอนวิธีการที่นำเสนอนี้ ได้วิเคราะห์จากขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอที่ได้ใช้กับโครงสร้างอะเรย์ ซึ่งเวลาที่ใช้ในการวิเคราะห์จะประกอบด้วย เวลาที่ใช้ในการติดต่อสื่อสาร α (Communication Time) และเวลาที่ใช้การคำนวณ β (Computation Time) โดยกำหนดให้เวลาในการติดต่อสื่อสาร 1 ขั้นเท่ากับเวลาการคำนวณ 1 ขั้น และการส่งข้อมูลในแบบขนานจะได้ความซับซ้อนด้านเวลาเท่ากับ 1 ขั้นในการติดต่อสื่อสาร

4.1 ความซับซ้อนด้านเวลาในแต่ละรอบการประมวลผล

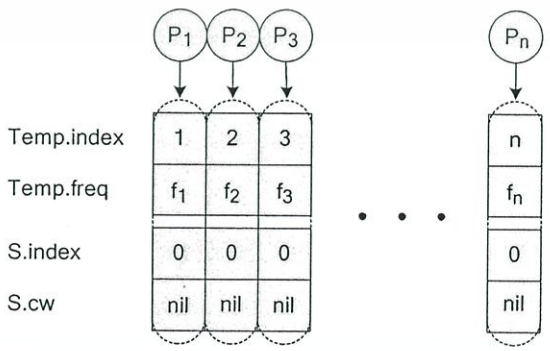
การวิเคราะห์ความซับซ้อนด้านเวลาในแต่ละรอบของการประมวลผล จะทำการวิเคราะห์จากความซับซ้อนด้านเวลาในแต่ละขั้นตอนย่อยของการประมวลผล ซึ่งมีรายละเอียดดังนี้

<u>ขั้นตอนที่ 1</u> การรับข้อมูลนำเข้าจำนวน n ค่าเข้าสู่อะเรย์

(การติดต่อสื่อสาร) การเก็บข้อมูลนำเข้าจำนวน n ค่าเข้าสู่อะเรย์เป็นการส่งค่าแบบขนานดังมีขั้นตอนดังนี้

```
Forall Processors Pi(1 ≤ i ≤ n) pardo
    Temp.index[i] ← i
    Temp.freq[i] ← frequency of symbol[i]
    S.index[i] ← 0
    S.cw[i] ← nil
    Create IntTemp array which its length equal ⌊n/2⌋
End Forall
```

ซึ่งมีการส่งข้อมูลนำเข้าไปยังทุกหน่วยประมวลผลจำนวน n หน่วยพร้อมกัน ดังรูปที่ 4.1



รูปที่ 4.1 การส่งข้อมูลนำเข้าไปยังทุกหน่วยประมวลผลจำนวน n หน่วย

ซึ่งจากรูปที่ 4.1 เป็นการส่งข้อมูลในแบบขนานจึงเป็นการติดต่อสื่อสาร 1 ครั้ง ดังนั้นเวลาที่ใช้ในขั้นตอนที่ 1 จึงได้ดังสมการที่ 4.1

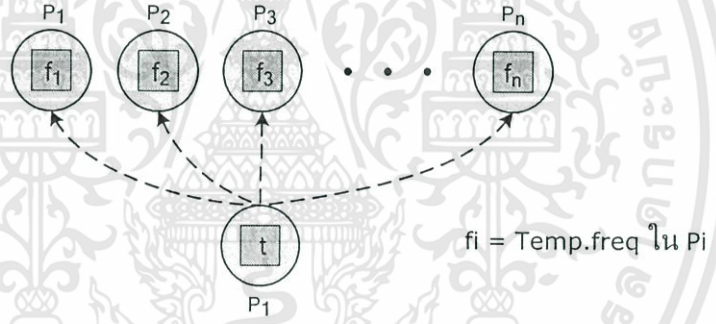
$$\alpha = 1 \tag{4.1}$$

จากสมการที่ 4.1 สามารถสรุปได้ว่าขั้นตอนที่ 1 มีความซับซ้อนด้านเวลาเป็น O(1)

ขั้นตอนที่ 2 การหาจำนวนสมาชิกที่เหมาะสมสำหรับการสร้างโหนดพ่อแม่

ขั้นตอนที่ 2.1 (การติดต่อสื่อสาร) การหาจำนวนสมาชิกในอะเรย์ Temp ที่เหมาะสมสำหรับนำมาจับคู่เพื่อสร้างโหนดพ่อแม่ใหม่โดยหน่วยประมวลผล P_1 ทำการรวมความถี่ที่น้อยที่สุด 2 ค่าเก็บไว้ในตัวแปร t จากนั้นทุกหน่วยประมวลผล (P_1, P_2, ..., P_n) อ่านค่า t พร้อมๆ กัน แล้วทำการเปรียบเทียบค่า t กับค่า Temp.freq ซึ่งอยู่ภายในหน่วยประมวลผลของตนเอง กับค่า t

หมายเหตุ การเข้าถึงค่า t ได้พร้อมกันเนื่องจากใช้รูปแบบการเข้าถึงข้อมูลแบบซีอาร์อีดับเบิลยูพีแรมโมเดล (CREW PRAM model) ดังแสดงในรูปที่ 4.2



รูปที่ 4.2 การเปรียบเทียบค่า t กับค่าความถี่ใน Temp.freq ทุกค่า

ดังนั้นจึงมีการติดต่อสื่อสารทั้งหมด 1 ครั้ง ดังแสดงในสมการที่ 4.2

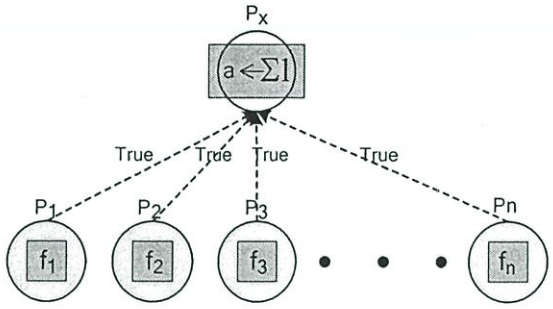
$$\alpha = 1 \tag{4.2}$$

จากสมการที่ 4.2 จึงได้ค่าความซับซ้อนด้านเวลาของขั้นตอนที่ 2.1 เป็น O(1)

ขั้นตอนที่ 2.2 (การคำนวณ) นำผลการเปรียบเทียบจากขั้นตอนที่ 2.1 มาเก็บไว้ในตัวแปร a โดยใช้วิธีการบีเอสอาร์ (BSR) ซึ่งได้กล่าวถึงในหัวข้อที่ 2.6 ในบทที่ 2 โดยเป็นการอาศัยหลักการกระจายข้อมูลเพื่อให้สามารถเก็บผลลัพธ์ไว้ในตัวแปรเดียวได้ ดังแสดงในขั้นตอนวิธีที่นำเสนอดังนี้

```
Forall Processors Pj (j=1 to length(Temp)) pardo
    a ← Σ1 | t≤ Temp.freq[j]
End Forall
```

ซึ่งสามารถแสดงการกระจายและเก็บผลลัพธ์ลงตัวแปร a ได้ดังรูปที่ 4.3



รูปที่ 4.3 การนำผลการเปรียบเทียบไปเก็บไว้ในตัวแปร a

โดยขั้นตอนนี้มีความซับซ้อนด้านเวลาเป็น O(1) [1][16][22][24][26][27]

จากค่าความซับซ้อนด้านเวลาของขั้นตอนที่ 2.1 และ 2.2 จึงได้ค่าความซับซ้อนด้านเวลาดังแสดงในสมการที่ 4.3

$$O(1) + O(1) = O(1) \tag{4.3}$$

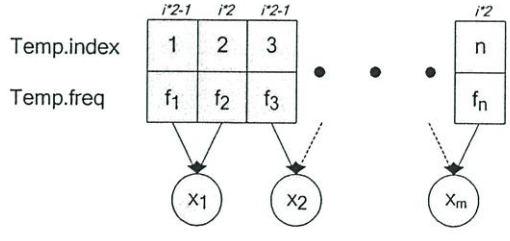
ดังนั้นจากสมการที่ 4.3 จึงสามารถสรุปได้ว่าขั้นตอนที่ 2 มีความซับซ้อนด้านเวลาเป็น O(1)

ขั้นตอนที่ 3 การจับคู่เพื่อสร้างโหนดพ่อแม่ใหม่จำนวน $\lfloor a/2 \rfloor$ คู่และการสร้างรหัสบิต

ขั้นตอนที่ 3.1 (การคำนวณ) การรวมความถี่ของโหนดลูกเพื่อสร้างโหนดพ่อแม่ใหม่ โดยในการรวมค่าจะทำการรวมทั้งหมด $\lfloor a/2 \rfloor$ คู่ในแบบขนาน ซึ่งมีขั้นตอนดังนี้

```
Forall Processors Pi (i=1 to ⌊a/2⌋) pardo
    IntTemp.index[i] ← c_index+i
    IntTemp.freq[i] ← Temp.freq[i*2-1] + Temp.freq[i*2]
    //Condition for left sub-tree coding
    ParHuffmanCodes(Temp.index, S.index, IntTemp.index, i*2-1, '0')
    //Condition for right sub-tree coding
    ParHuffmanCodes(Temp.index, S.index, IntTemp.index, i*2, '1')
End Forall
```

โดยสามารถแสดงการรวมความถี่เพื่อสร้างโหนดพ่อแม่ใหม่ในแบบขนานได้ดังรูปที่ 4.4

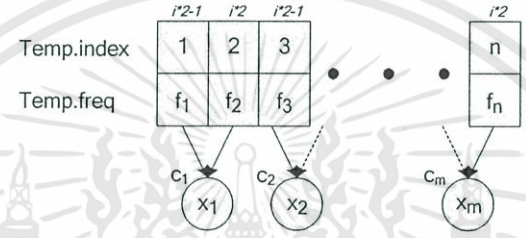


รูปที่ 4.4 รวมความถี่เพื่อสร้างโหนดพ่อแม่ใหม่แบบขนาน

ดังนั้นเวลาที่ใช้ในการรวมค่าของโหนดพ่อแม่ใหม่ จึงแสดงได้ดังสมการที่ 4.4

$$\beta = 1 \tag{4.4}$$

ขั้นตอนที่ 3.2 (การคำนวณ) การกำหนดหมายเลขประจำโหนดให้แก่โหนดพ่อแม่ใหม่จำนวน $\lfloor a/2 \rfloor$ โหนด จะกระทำในแบบขนาน ซึ่งหมายเลขประจำโหนดพ่อแม่ใหม่ปัจจุบัน (Cj = c_index + i ซึ่ง j = 1, 2, …, m) จะ เท่ากับหมายเลขประจำโหนดสุดท้ายของรอบประมวลผลก่อนหน้า (c_index) บวกลำดับการสร้างโหนดพ่อแม่ใหม่ (i) ซึ่งค่า i จะมีลำดับตั้งแต่ 1 ถึง $\lfloor a/2 \rfloor$ ดังแสดงในรูปที่ 4.5 ต่อไปนี้



รูปที่ 4.5 การกำหนดหมายเลขประจำโหนดให้แก่โหนดพ่อแม่ใหม่ในแบบขนาน

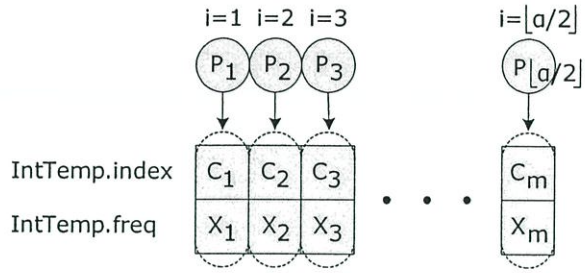
ขั้นตอนนี้เป็นการประมวลผลแบบขนานจึงได้เวลาในการประมวลผล ดังในสมการที่ 4.5

$$\beta = 1 \tag{4.5}$$

ขั้นตอนที่ 3.3 (การติดต่อสื่อสาร) การเก็บหมายเลขประจำโหนดและความถี่ของโหนดพ่อแม่ใหม่ในอะเรย์ IntTemp โดยการส่งข้อมูลในแบบขนานด้วยหน่วยประมวลผลจำนวน $\lfloor a/2 \rfloor$ หน่วย ซึ่งมีขั้นตอนการประมวลผลในแบบขนานดังนี้

```
Forall Processors Pi (i=1 to ⌊a/2⌋) pardo
    IntTemp.index[i] ← c_index+i
    IntTemp.freq[i] ← Temp.freq[i*2-1] + Temp.freq[i*2]
    //Condition for left sub-tree coding
    ParHuffmanCodes(Temp.index, S.index, IntTemp.index, i*2-1, '0')
    //Condition for right sub-tree coding
    ParHuffmanCodes(Temp.index, S.index, IntTemp.index, i*2, '1')
End For
```

จากรูปที่ 4.5 ได้โหนดพ่อแม่ใหม่จำนวน $\lfloor a/2 \rfloor$ โหนดตามลำดับ i ซึ่งจะต้องเก็บข้อมูลของโหนดพ่อแม่ใหม่เหล่านั้นไว้ในอะเรย์ IntTemp โดยกระทำในแบบขนานดังแสดงในรูปที่ 4.6



รูปที่ 4.6 การเก็บข้อมูลโหนดพ่อแม่ใหม่ไว้ในอะเรย์ IntTemp ในแบบขนาน

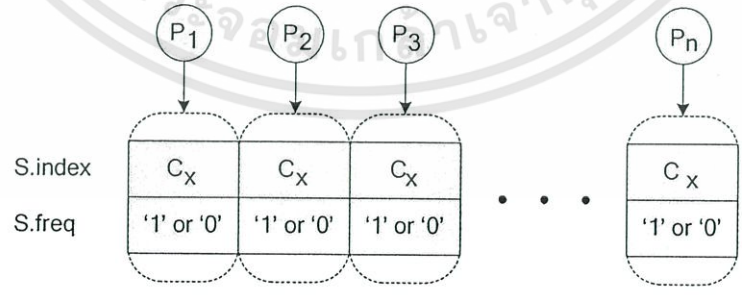
ในขั้นตอนนี้มีการประมวลผลเป็นแบบขนานดังนั้นจึงได้เวลาในการติดต่อสื่อสารดังสมการที่ 4.6

$$\alpha = 1 \tag{4.6}$$

ขั้นตอนที่ 3.4 (การติดต่อสื่อสาร) การเพิ่มบิตและเก็บหมายเลขประจำโหนดพ่อแม่ใหม่ให้กับแต่ละสมาชิกในอะเรย์ S มีขั้นตอนดังนี้

```
If (Temp.index[Pchild] ≤ n)
  S.index[Temp.index[Pchild]] ← IntTemp.index[i]
  S.cw[Temp.index[Pchild]] ← B
Else
  Forall Processors Px(x=1 to length(S)) pardo
    If (S.index[x] = Temp.index[Pchild])
      S.index[x] ← IntTemp.index[i]
      S.cw[x] ← inserts bit of B in front of its old bits
    End If
  End Forall
End If
```

ซึ่งประมวลผลโดยใช้การปรับข้อมูลในแบบขนานด้วยหน่วยประมวลผลจำนวน n หน่วย ดังแสดงในรูปที่ 4.7



รูปที่ 4.7 การเก็บหมายเลขประจำโหนดพ่อแม่ใหม่และการปรับรหัสบิตในแบบขนาน

ดังนั้นเวลาที่ใช้ในการปรับค่าในอะเรย์ S จึงได้ดังสมการที่ 4.7

$$\alpha = 1 \tag{4.7}$$

ขั้นตอนที่ 3.1 ถึง 3.4 ได้ความซับซ้อนด้านเวลาทั้งหมดเป็น τ ดังแสดงในสมการที่ 4.8

$$\begin{aligned} \tau &= \beta + \beta + \alpha + \alpha \\ &= 1 + 1 + 1 + 1 \end{aligned} \tag{4.8}$$

จากสมการที่ 4.8 สามารถสรุปได้ว่าขั้นตอนที่ 3 มีความซับซ้อนด้านเวลาเป็น O(1)

ขั้นตอนที่ 4 การปรับค่าให้กับตัวแปร c_index และการเตรียมอะเรย์

ขั้นตอนที่ 4.1 (การคำนวณ) การปรับค่าในตัวแปร c_index จะกระทำโดยการนำจำนวนโหนดพ่อแม่ใหม่ที่ได้ในรอบปัจจุบัน ($\lfloor a/2 \rfloor$) รวมกับค่า c_index เริ่มต้นของรอบนั้นๆ ดังแสดงในขั้นตอนวิธีดังนี้

c_index ← ***c_index***+$\lfloor a/2 \rfloor$
Delete all elements in *Temp* that melded.
Temp ← *Merge_BSR*(*Temp*, *IntTemp*)
Delete all elements in *IntTemp*.

ซึ่งได้เวลาในการประมวลผลดังสมการที่ 4.9

$$\beta = 1 \tag{4.9}$$

ขั้นตอนที่ 4.2 (การคำนวณ) การลบสมาชิกที่ได้นำไปใช้ในการสร้างโหนดพ่อแม่ใหม่จะกระทำโดยใช้เวลาในการประมวลผลเท่ากับ 1 ครั้งดังสมการที่ 4.10

$$\beta = 1 \tag{4.10}$$

ขั้นตอนที่ 4.3 (การคำนวณ) การรวมสมาชิกที่เหลือในอะเรย์ Temp กับสมาชิกทั้งหมดในอะเรย์ IntTemp ได้กระทำโดยใช้ขั้นตอนวิธีการรวมข้อมูลแบบบีเอสอาร์ (Merging on BSR) ซึ่งได้กล่าวไว้ในหัวข้อที่ 2.6.1 และ 2.6.2 ในบทที่ 2 ซึ่งมีความซับซ้อนด้านเวลาเป็น O(1) [27]

c_index ← *c_index*+$\lfloor a/2 \rfloor$
Delete all elements in *Temp* that melded.
***Temp* ← *Merge_BSR*(*Temp*, *IntTemp*)**
Delete all elements in *IntTemp*.

ขั้นตอนที่4.4 (การคำนวณ) การลบสมาชิกทั้งหมดในอะเรย์ IntTemp ได้ใช้เวลาในการประมวลผลเท่ากับ 1 ครั้งดังสมการที่ 4.11

$$\beta = 1 \tag{4.11}$$

จากสมการที่ 4.9 4.10 และ 4.11 ได้เวลารวมในการประมวลผลของขั้นตอนที่ 4.1 4.2 และ 4.4 เป็น τ ดังแสดงในสมการที่ 4.12

$$\begin{aligned} \tau &= \beta + \beta + \beta \\ &= 1 + 1 + 1 \end{aligned} \tag{4.12}$$

จากสมการที่ 4.12 และค่าความซับซ้อนด้านเวลาของขั้นตอนที่ 4.3 สามารถสรุปได้ว่าขั้นตอนที่ 4 มีความซับซ้อนด้านเวลาเป็น O(1)

จากผลการวิเคราะห์ความซับซ้อนด้านเวลาในขั้นตอนที่ 1 ถึง 4 จะได้ความซับซ้อนด้านเวลาที่ใช้ในแต่ละรอบการประมวลผลดังแสดงในสมการที่ 4.13

$$O(1) + O(1) + O(1) + O(1) = O(1) \tag{4.13}$$

จากสมการที่ 4.13 จึงสามารถสรุปได้ว่าการประมวลผลในแต่ละรอบจะมีความซับซ้อนด้านเวลาเป็น O(1)

4.2 ความซับซ้อนด้านเวลาโดยรวมของขั้นตอนวิธีที่นำเสนอ

การประมวลผลของขั้นตอนการเข้ารหัสของฮัฟแมนสำหรับชุดข้อมูลต้องใช้จำนวนรอบเท่ากับจำนวนลำดับชั้นของต้นไม้ฮัฟแมน ดังนั้นความซับซ้อนด้านเวลาของขั้นตอนวิธีนี้จึงขึ้นอยู่กับจำนวนรอบที่ใช้ในการประมวลผลหรือลำดับชั้นของต้นไม้ฮัฟแมน

โดยสมมุติจำนวนรอบที่ใช้ในการประมวลผลเท่ากับ L รอบ ซึ่งเท่ากับจำนวนลำดับชั้นของต้นไม้ฮัฟแมน ดังนั้นความซับซ้อนด้านเวลาสำหรับการประมวลผลทั้งหมดจึงเท่ากับ O(L) โดยคำนวณด้วยสมการที่ 4.14 ซึ่งในแต่ละรอบการประมวลผลมีความซับซ้อนด้านเวลาเท่ากับ O(1)

$$O(\sum_{i=1}^{L}(1)) \tag{4.14}$$

ในการวิเคราะห์ความซับซ้อนด้านเวลาสำหรับขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่ได้นำเสนอนี้ สามารถแบ่งขอบเขตผลการวิเคราะห์ความซับซ้อนได้ 3 กรณีคือ

1. กรณีที่โครงสร้างต้นไม้มีลักษณะการกระจายกิ่งแบบสมดุลหรือมีความซับซ้อนด้านเวลาดีที่สุด (Best Case)
2. กรณีที่โครงสร้างต้นไม้มีลักษณะกระจายกิ่งไปในทิศทางเดียวหรือมีความซับซ้อนด้านเวลาแย่ที่สุด (Worst Case)
3. กรณีที่โครงสร้างต้นไม้มีการกระจายกิ่งแบบทั่วไปหรือมีความซับซ้อนด้านเวลาโดยเฉลี่ย (Average Case)

<u>กรณีที่ 1</u> ความซับซ้อนด้านเวลาในกรณีโครงสร้างต้นไม้มีลักษณะสมดุลหรือกรณีดีที่สุด (Best Case)

กรณีที่สัญลักษณ์ในชุดข้อมูลมีความถี่ใกล้เคียงกัน ซึ่งเมื่อนำมาสร้างต้นไม้ฮัฟแมนจะได้ต้นไม้ที่มีการกระจายกิ่งอย่างสมดุลทั้งฝั่งซ้ายและขวาหรือหากเปรียบเทียบในขั้นตอนวิธีที่ได้นำโครงสร้างอะเรย์มาใช้ในการประมวลผลจะเป็นการนำสมาชิกทั้งหมดในอะเรย์ Temp มาใช้ในการสร้างโหนดพ่อแม่ใหม่ ซึ่งจะทำให้ได้สมาชิกในอะเรย์ Temp หลังจากการประมวลผลลดลงในแต่ละรอบเป็นจำนวน n/2, n/4, n/8,..., 4, 2, 1 ซึ่งแสดงว่าจะต้องทำการประมวลผลทั้งหมด log n รอบ ดังนั้นจึงได้ค่าความซับซ้อนด้านเวลาสำหรับกรณีที่ดีที่สุด ดังแสดงในสมการที่ 4.15

$$O(\sum_{i=1}^{\log n}(1) = 1_1 + 1_2 + 1_3 + \ldots + 1_{\log n})$$

$$O(\sum_{i=1}^{\log n}(1) = \log n) = O(\log n) \qquad (4.15)$$

ดังนั้นสามารถสรุปได้ว่าวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอมีความซับซ้อนด้านเวลาในกรณีที่ดีที่สุดเป็น O(log n)

<u>กรณีที่ 2</u> ความซับซ้อนด้านเวลาในกรณีโครงสร้างต้นไม้มีการกระจายไปในทิศทางเดียวหรือกรณีแย่ที่สุด (Worst Case)

กรณีที่สัญลักษณ์ในชุดข้อมูลมีความถี่ของการปรากฏต่างกันมากทำให้เมื่อนำมาสร้างเป็นต้นไม้ฮัฟแมนจะได้โครงสร้างต้นไม้ที่มีการกระจายกิ่งไปในทิศทางเดียว ซึ่งเมื่อเปรียบเทียบกับการทำงานในขั้นตอนวิธีที่ได้นำโครงสร้างอะเรย์มาใช้ในการประมวลผลจะเป็นการจับคู่สมาชิกในอะเรย์ Temp ได้ไม่เกิน 1 คู่ต่อ 1 รอบของการประมวลผลเช่นเดียวกับการเข้ารหัสของฮัฟแมนแบบดั้งเดิม ซึ่งทำให้ได้สมาชิกในอะเรย์ Temp หลังจากการประมวลผลในแต่ละรอบลดลงทีละ 1

สมาชิก (n-1, n-2, n-3,..., 3, 2, 1) ซึ่งแสดงว่าจะต้องทำการประมวลผลทั้งหมด n-1 รอบ ดังนั้นจึงได้ค่าความซับซ้อนด้านเวลาสำหรับกรณีที่แย่ที่สุด ดังแสดงในสมการที่ 4.16

$$O(\sum_{i=1}^{n-1}(1) = 1_1 + 1_2 + 1_3 + \ldots + 1_{n-1})$$

$$O(\sum_{i=1}^{n-1}(1) = n\text{-}1) = O(n) \qquad (4.16)$$

ดังนั้นสามารถสรุปได้ว่าวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอมีความซับซ้อนด้านเวลาในกรณีที่แย่ที่สุดเป็น O(n)

<u>กรณีที่ 3</u> ความซับซ้อนด้านเวลาในกรณีโครงสร้างต้นไม้มีลักษณะทั่วไปหรือมีความซับซ้อนด้านเวลาโดยเฉลี่ย (Average Case)

กรณีที่สัญลักษณ์ของข้อมูลมีความถี่ที่ทำให้การกระจายกิ่งในต้นไม้ไม่สมดุลและไม่ได้เป็นการกระจายกิ่งไปในทิศทางเดียว ซึ่งเป็นกรณีที่สามารถเกิดได้ทั่วไป ค่าความซับซ้อนด้านเวลาที่ได้จะเป็นค่าความซับซ้อนด้านเวลาโดยเฉลี่ยของขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอ ดังแสดงในสมการที่ 4.17

$$O\left(\frac{\sum_{i=1}^{\log n}(1) + \sum_{i=1}^{\log n+1}(1) + \ldots + \sum_{i=1}^{n-1}(1)}{(n-1) - (\log n - 1)}\right) = O\left(\frac{\log n + ((\log n)+1) + \ldots + (n-1)}{n - \log n}\right)$$

$$= O\left(\frac{\sum_{i=\log n}^{n-1}(i)}{n - \log n}\right)$$

$$= O\left(\frac{\sum_{i=1}^{n-1}(i) - \sum_{i=1}^{(\log n)-1}(i)}{n - \log n}\right) \qquad (4.17)$$

เนื่องจาก $\sum_{i=1}^{n}(i) = \frac{n(n+1)}{2}$

ดังนั้น $\sum_{i=1}^{n-1}(i) = \frac{(n-1)((n-1)+1)}{2}$

$$= \frac{n(n-1)}{2} \quad (4.18)$$

และ $\sum_{i=1}^{(\log n)-1}(i) = \frac{((\log n)-1))(((\log n)-1)+1)}{2}$

$$= \frac{\log n((\log n)-1)}{2} \quad (4.19)$$

เมื่อนำสมการที่ 4.18 และ 4.19 แทนค่าในสมการที่ 4.17 จะได้ผลดังสมการที่ 4.20

$$O\left(\frac{\sum_{i=1}^{n-1}(1) - \sum_{i=1}^{(\log n)-1}(1)}{n - \log n}\right) = O\left(\frac{\frac{n(n-1)}{2} - \frac{(\log n)((\log n)-1)}{2}}{n - \log n}\right)$$

$$< O\left(\frac{n^2 - (\log n)^2}{2} \times \frac{1}{n - \log n}\right)$$

$$= O\left(\frac{(n + \log n)\cancel{(n - \log n)}}{2\cancel{(n - \log n)}}\right)$$

$$= O\left(\frac{n + \log n}{2}\right)$$

$$= O(n) \quad (4.20)$$

ดังนั้นจากสมการที่ 4.20 จึงสรุปได้ว่าวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอในงานวิจัยนี้มีความซับซ้อนด้านเวลาเป็น O(n)

บทที่ 5
การเปรียบเทียบการทำงานของขั้นตอนวิธีและบทสรุป

ในบทนี้จะเปรียบเทียบขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอกับขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานแบบเดิม บทสรุปงานวิจัย และแนวทางการพัฒนางานวิจัย

5.1 การเปรียบเทียบขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอกับขั้นตอนการเข้ารหัสของฮัฟแมนแบบขนานแบบเดิม

จากขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอโดย Milidiu และคณะ [18] ในปี ค.ศ.1999 ได้มีการพัฒนาการสร้างต้นไม้ฮัฟแมนในแบบขนาน โดยไม่ได้พัฒนาส่วนการแสดงรหัสจากโหนดราก (Root Node) ไปยังโหนดใบ (Leaf Node) แต่ละโหนด ทำให้ได้ค่าความซับซ้อนด้านเวลาในกรณีที่ดีที่สุดเป็น O(log(1/p1) log(log n)) ซึ่ง p1 คือผลหารระหว่างความถี่ของสัญลักษณ์ที่น้อยที่สุดกับความถี่ของสัญลักษณ์ทั้งหมด และในปี ค.ศ.2006 Ostadzadeh และคณะ [19] ได้นำหลักการสร้างต้นไม้ฮัฟแมนในแบบขนานของ Milidiu และคณะมาพัฒนาต่อเพื่อให้สามารถสร้างรหัสบิตของฮัฟแมนโดยใช้โครงสร้างอะเรย์ในการเก็บข้อมูลของโหนดในต้นไม้ฮัฟแมนแทนการสร้างต้นไม้ฮัฟแมน ซึ่งได้ความซับซ้อนด้านเวลาในกรณีที่ดีที่สุดเป็น O(log((log(n-1))!)) แต่ขั้นตอนวิธีนี้ใช้โครงสร้างอะเรย์จำนวนมากในการประมวลผล งานวิจัยนี้จึงได้พัฒนาขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่ลดความซับซ้อนของขั้นตอนการประมวลผลและมีค่าความซับซ้อนด้านเวลาในกรณีที่ดีที่สุดเป็น O(log n) ในขณะที่ความซับซ้อนด้านเวลาในกรณีที่แย่ที่สุดเป็น O(n) ซึ่งเท่ากับความซับซ้อนด้านเวลาในกรณีที่แย่ที่สุดของวิธีการเข้ารหัสของฮัฟแมนแบบขนานแบบเดิม [18][19] นอกจากนี้เมื่อนำโครงสร้างอะเรย์มาประยุกต์ใช้กับวิธีที่นำเสนอในวิจัยนี้พบว่าใช้จำนวนอะเรย์และขนาดของอะเรย์ที่น้อยกว่าวิธีการของ Ostadzadeh และคณะ [19] ดังแสดงการเปรียบเทียบความแตกต่างของขั้นตอนวิธีที่นำเสนอกับขั้นตอนวิธีการเดิม [18][19] ในตารางที่ 5.1 และการเปรียบเทียบค่าความซับซ้อนด้านเวลาและหน่วยความจำที่ใช้ในการประมวลผลระหว่างวิธีที่นำเสนอกับขั้นตอนวิธีการเดิมในตารางที่ 5.2

ตารางที่ 5.1 แสดงการเปรียบเทียบขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานระหว่างขั้นตอนวิธีที่นำเสนอกับขั้นตอนวิธีการเดิม

ขั้นตอนวิธี	ส่วนการสร้างต้นไม้ฮัฟแมน	ส่วนการเข้ารหัสฮัฟแมน
ES-ParHuff [18]	พัฒนาบนรูปแบบการประมวลผลแบบขนาน	ไม่ได้พัฒนา
Two-phase Practical Parallel [19]	นำขั้นตอนจากวิธี ES-PatHuff มาพัฒนาด้วยโครงสร้างอะเรย์	พัฒนาด้วยโครงสร้างอะเรย์
ขั้นตอนที่นำเสนอในงานวิจัยนี้	สร้างรหัสบิตของแต่ละสัญลักษณ์ในขั้นตอนเดียวกับการสร้างโครงสร้างต้นไม้ฮัฟแมน	

ผลการเปรียบเทียบค่าความซับซ้อนด้านเวลาและขนาดหน่วยความจำที่ใช้ในการประมวลผลของวิธีการที่นำเสนอกับขั้นตอนวิธีการเดิม [18][19] แสดงในตารางที่ 5.2

ตารางที่ 5.2 แสดงการเปรียบเทียบค่าความซับซ้อนด้านเวลาและหน่วยความจำที่ใช้ในระหว่างประมวลผลระหว่างขั้นตอนวิธีที่นำเสนอกับขั้นตอนวิธีการเดิม

ขั้นตอนวิธี	ความซับซ้อนด้านเวลากรณีดีที่สุด	ความซับซ้อนด้านเวลากรณีแย่ที่สุด	จำนวนความจำที่ใช้ในการประมวลผล (อะเรย์)
ES-ParHuff [18]	O(log(1/p1) log(log n))	O(n)	-
Two-phase Practical Parallel [19]	O(log((log(n-1))!))	O(n)	7
ขั้นตอนที่นำเสนอในงานวิจัยนี้	O(log n)	O(n)	3

5.2 บทสรุปงานวิจัย

งานวิจัยนี้ได้นำเสนอขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่ลดความซับซ้อนของขั้นตอนการประมวลผลและมีความซับซ้อนด้านเวลาในกรณีที่โครงสร้างต้นไม้มีการกระจายกิ่งแบบสมดุลหรือกรณีที่ดีที่สุดเป็น O(log n) ด้วยหน่วยประมวลผล n หน่วย ซึ่งมีค่าความซับซ้อนด้านเวลาน้อยกว่าวิธีการเดิมเมื่อสัญลักษณ์ในข้อมูลมีจำนวนเพิ่มมากขึ้น สำหรับในกรณีที่โครงสร้างต้นไม้มีการกระจายกิ่งไปในทิศทางเดียวหรือกรณีที่แย่ที่สุดจะมีความซับซ้อนด้านเวลาเป็น O(n) และได้ความซับซ้อนด้านเวลาโดยเฉลี่ยเป็น O(n)

ตารางที่ 5.3 แสดงผลสรุปความซับซ้อนด้านเวลาของขั้นตอนวิธีการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอในงานวิจัย

กรณีที่ใช้ในการวิเคราะห์	ค่าความซับซ้อนด้านเวลา
กรณีต้นไม้ฮัฟแมนมีการกระจายกิ่งแบบสมดุล (Best Case)	O(log n)
กรณีต้นไม้ฮัฟแมนมีการกระจายกิ่งแบบทั่วไป (Average Case)	O(n)
กรณีต้นไม้ฮัฟแมนมีการกระจายกิ่งในทิศทางเดียว (Worst Case)	O(n)

การเข้ารหัสของฮัฟแมนด้วยวิธีการที่นำเสนอนี้ใช้หลักการสร้างเช่นเดียวกับการเข้ารหัสของฮัฟแมนแบบดั้งเดิม ดังนั้นผลลัพธ์ซึ่งเป็นรหัสบิตที่ประมวลผลจากขั้นตอนวิธีการที่นำเสนอนี้จะมีลักษณะเหมือนกับรหัสบิตที่ประมวลผลจากวิธีการเข้ารหัสของฮัฟแมนแบบดั้งเดิม ด้วยเหตุนี้ชุดรหัสบิตที่ได้จึงสามารถนำไปถอดรหัสได้ด้วยขั้นตอนวิธีการถอดรหัสที่มีประสิทธิภาพที่รองรับชุดรหัสบิตของฮัฟแมน อีกทั้งขั้นตอนวิธีการที่นำเสนอนี้มีขั้นตอนที่ไม่ซับซ้อน เข้าใจง่าย และสามารถนำไปประยุกต์ใช้กับโครงสร้างข้อมูลที่มีประสิทธิภาพเพื่อสะดวกในการใช้งานโดยใช้หน่วยความจำในการประมวลผลน้อยกว่าวิธีการเดิม

5.3 แนวทางการพัฒนางานวิจัย

จากการขั้นตอนวิธีเข้ารหัสของฮัฟแมนทุกวิธีที่ผ่านมาจะทำให้ได้ผลลัพธ์ที่เป็นรหัสบิตที่สอดคล้องกับสัญลักษณ์ในชุดข้อมูลนำเข้าซึ่งเมื่อส่งชุดรหัสบิตเหล่านั้นไปยังปลายทางโดยผ่านช่องทางการสื่อสารต่างๆ จะทำให้ข้อมูลไม่มีความปลอดภัย เนื่องจากอาจจะมีบุคคลที่สามคัดลอกรหัสบิตเหล่านั้นไปถอดรหัสเพื่อหาความหมายของข้อมูล หรือ นำข้อมูลเหล่านั้นไปแก้ไขให้เกิดความผิดเพี้ยนได้ ดังนั้นแนวทางการพัฒนางานวิจัยในอนาคตคือการออกแบบขั้นตอนวิธีการเข้ารหัสของฮัฟแมนโดยเพิ่มขั้นตอนวิธีการเข้ารหัสลับ (Encryption) เข้าไปในกระบวนการเข้ารหัสของฮัฟแมนเพื่อเพิ่มความปลอดภัยให้กับข้อมูล

เอกสารอ้างอิง

[1] Albert Y. Zomaya. **Parallel & Distributed Computing Handbook**. McGraw-Hill. 1996.

[2] Atallah M. J., Kosaraju S. R., Larmore L. L., Miller G. L. and Teng S-H. "Constructing Trees in Parallel." **ACM SIGACT, Proc. 1st Annual ACM Symposium on Parallel Algorithm and Architectures.,** Jun. 1989. pp. 421-431.

[3] Buayen P. and Werapun J. "A New Efficient Parallel Huffman Coding Algorithm." **The 5th International Joint Conference on Computer Science and Software Engineering (JSCCE2008).**, vol. 1, May. 2008. pp. 149-154.

[4] Chuang Y. J. and Wu J.-L. "An SGII-Tree Based Memory Efficient and Constant Decoding Algorithm for Huffman Codes." **IEEE Proc. International on Intelligent Multiple, Video and Speech Processing.,** Oct. 2004. pp. 374-377.

[5] David Salomon. **Data Compression: The complete reference Fourth edition.** Springer-Verlag, London Limited. 2007.

[6] Gheorghe S. Almasi and Allan G. **Highly Parallel Computing**. The Benjamin/Cummings Publishing Company Inc. 1994.

[7] Huffman D. A. "A method for the construction of minimum redundancy codes." **Proceedings of the Institute of Radio Engineers (IRE).,** vol. 40, Sep. 1952. pp. 1098-1101.

[8] Harry F. Jordan and Gita Alaghband. **Fundamentals of Parallel Processing**. Pearson Education Inc. 2003.

[9] Hashemian R. "Memory Efficient and High-Speed Search Huffman Coding." IEEE Trans. Comm. **Transactions and Communications.,** vol. 43, no. 10, Oct. 1995. pp. 2576-2581.

[10] Hashemian R. "Condensed Table of Huffman Coding, a New Approach to Efficient Decoding." **IEEE Transactions and Communications.,** vol. 52, No.1, Jan. 2004. pp. 6-8.

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

[12] Kirkpatrick D. G. and Przytycka, T. "Parallel construction of near optimal binary trees." **ACM SIGACT, Proc. 2nd Annual ACM Symposium on Parallel Algorithm and Architectures.,** Jul. 1990. pp. 234-242.

[13] Kruskal C. P. "Searching, Merging and Sorting in Parallel Computation." **IEEE Trans.,** vol. c-32, no. 10, October 1983. pp. 942-946.

[14] Lin Y.-K. and Chung K.-L. "A space-efficient Huffman decoding algorithm and its parallelism." **ELSEVIER on Theoretical Computer Science.,** 2000. pp. 227-238.

[15] Lin J.-Y., Liu Y. and Yi K.-C. "Balance of 0, 1 Bits for Huffman and Reversible Variable-Length Coding." **IEEE Transactions and Communications.,** vol. 52, no. 3, Mar. 2004. pp. 359-361.

[16] Lindon L. F. and Akl S. G. "An Optimal Implementation of Broadcasting with Selective Reduction." **IEEE Trans. on Parallel and Distributed systems.,** vol. 4, no. 3, Mar. 1993.

[17] Mark Nelson and Jean-Loup Gailly. **The Data Compression Book.** M&T Publishing, Inc. 1991.

[18] Milidiú R. L., Laber E. S. and Pessoa A. A. "A work efficient parallel algorithm for constructing Huffman codes." **Proceedings of Data Compression Conference (DCC '99).** Mar. 1999.

[19] Ostadzadeh S. A., Elahi M. B., Tabrizi Z. Z., Moulavi A. M. and Bertels K. "A Two-phase Practical Parallel Algorithm for Construction of Huffman Codes." **3rd International Conference on Neural, Parallel and Scientific Computations.** Morehouse College, IFNA, Atlanta, GA, USA, Aug. 2006.

[20] Peter Wayner. **Compression Algorithms for Real Programmers**. Academic Press. 2000.

[21] Richard E. Neapolitan and Kumarss Naimipor. **Foundations of Algorithms Using Java Pseudocode**. Jones and Bartlett Publishers Inc. 2004.

[22] Semé D. and Myoupo J.-F. "Efficient BSR-Based Parallel Algorithms for Geometrical Problems." **IEEE Proceeding of Parallel and Distribted.,** Feb. 2001. pp. 265-272.

[23] Wang S.-W., Chuang S.-C., Hsiao C.-C., Tung Y.-S and Wu J.-L. "An Efficient Memory Construction Scheme for an Arbitrary Side Growing Huffman Table." **IEEE International Conference on Multimedia and Expo (ICME).,** Jul. 2006. pp. 141 - 144

[24] Xiang L. and Ushijima K. "Optimal Parallel Merging Algorithms on BSR." **IEEE Proceedings of Parallel Architecture, Algorithms and Networks (I-SPAN).,** Dec. 2000. pp. 12-17.

[25] Xiang L., Ushijima K. and Cheng K. "Encoding Binary Search Tree in Constant Time on BSR." **IEEE Proc. Parallel and Distributed Computing, Applications and Technologies (PDCAT'05).,** Dec. 2005. pp. 740-744.

[26] Xiang L., Ushijima K., Cheng K., Zhao J. and Lu C. "O(1) Time algorithm on BSR for Constructing a Binary Search Tree with Best Frequencies." **Springer-Verlag Berlin Heidelberg.,** 2004. pp. 218-225.

[27] Xiang L., Ushijima K., Zhao J., Zhang T. and Tang C. "O(1) Time Algorithm in BSR for Constructing a Random Binary Search Tree." **IEEE Proc. Parallel and Distributed Computing, Applications and Technologies (PDCAT'03).,** Aug. 2003. pp. 599-602.

[28] Yuh-Dauh Lyuu. **Information dispersal and parallel computation (Cambridge International Series on Parallel Computation: 3).** Cambridge University Press, 1992.

ภาคผนวก ก.

ตัวอย่างการเข้ารหัสของฮัฟแมนแบบขนานที่นำเสนอในงานวิจัย

ตัวอย่างที่ 1

การเข้ารหัสของฮัฟแมนด้วยขั้นตอนวิธีแบบขนาน สำหรับข้อมูลที่มีความถี่ใกล้เคียงกัน ซึ่งทำให้การกระจายกิ่งในต้นไม้ฮัฟแมนมีลักษณะสมดุล

สมมุติในชุดข้อมูลมีสัญลักษณ์ S_1, S_2, S_3, ... , S_{16} ตามลำดับ ซึ่งมีความถี่ของการปรากฏสัญลักษณ์แต่ละสัญลักษณ์ดังนี้ 3, 3, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6 และ 6 ตามลำดับ

ขั้นตอนเตรียมข้อมูล

ในขั้นตอนนี้จะกำหนดค่าเริ่มต้นให้กับตัวแปร n และ c_index ให้เท่ากับจำนวนสัญลักษณ์ในชุดข้อมูล คือ 16 จากนั้นนำค่าความถี่ของการปรากฏสัญลักษณ์แต่ละสัญลักษณ์เก็บไว้ใน Temp.freq พร้อมทั้งกำหนดหมายเลขประจำโหนดตั้งแต่ 1 ถึง 16 ใน Temp.index ตามลำดับ ซึ่งหมายเลขนี้จะตรงกับลำดับของสมาชิกในอะเรย์ S และกำหนดค่าเริ่มต้น 0 และ nil ให้กับ S.index และ S.cw ตามลำดับดังรูป

n = 16 c_index = 16

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Temp.index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Temp.freq	3	3	3	3	3	3	4	4	4	5	5	5	6	6	6	6

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S.index	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S.cw	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil

ขั้นตอนการสร้างโหนดพ่อแม่ใหม่และการเพิ่มบิต

รอบที่ 1

1. นำค่าความถี่ของอะเรย์ Temp ช่องที่ 1 และ 2 ซึ่งเป็นสมาชิกที่มีความถี่น้อยสุด มารวมกันแล้วเก็บไว้ในตัวแปร t ซึ่งในที่นี้จะได้ค่าในตัวแปร t เท่ากับ 6 (3+3) ดังรูป

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Temp.index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Temp.freq	3	3	3	3	3	3	4	4	4	5	5	5	6	6	6	6

t ← (6)

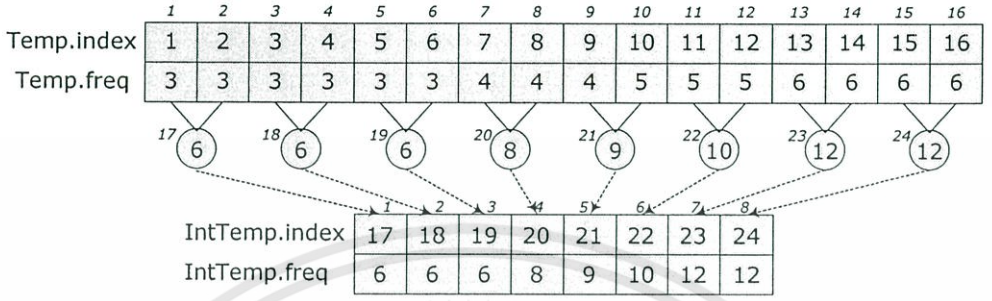
แล้วให้ทุกหน่วยประมวลผลมาอ่านค่า t (= 6) นำไปเปรียบเทียบกับทุกค่าใน Temp.freq ซึ่งค่าใน Temp.freq ที่น้อยกว่าหรือเท่ากับ 6 มี 16 ตัว ดังนั้นค่าในตัวแปร a จึงเท่ากับ 16 ดังรูป

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Temp.index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Temp.freq	3	3	3	3	3	3	4	4	4	5	5	5	6	6	6	6

t = (6)

a = 16

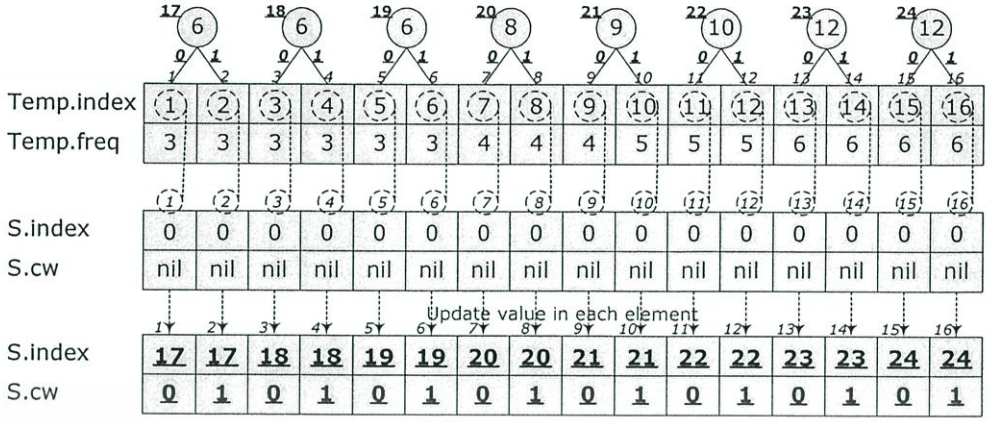
2. จับคู่สมาชิกในอะเรย์ Temp จำนวน 8 ($\lfloor a/2 \rfloor = \lfloor 16/2 \rfloor$) คู่ ตามลำดับของตำแหน่งช่องสมาชิกเพื่อสร้างโหนดพ่อแม่ใหม่ โดยนำค่าใน Temp.freq ของแต่ละคู่มารวมกันแล้วเก็บผลรวมไว้ใน IntTemp.freq ตามลำดับการจับคู่และเก็บหมายเลขประจำโหนดพ่อแม่ใหม่ของแต่ละสมาชิกไว้ใน IntTemp.index ดังรูป



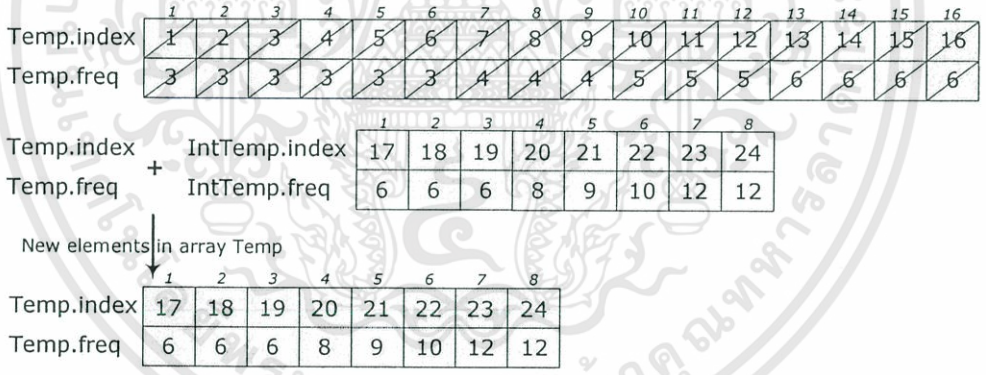
3. ปรับค่าในอะเรย์ S เฉพาะสมาชิกที่เกี่ยวข้องกับการสร้างโหนดพ่อแม่ใหม่ โดยจะเก็บหมายเลขประจำโหนดพ่อแม่ไว้ใน S.index ของสมาชิกที่เป็นลูกของโหนดพ่อแม่ใหม่นั้น และทำการเพิ่มบิตให้กับสมาชิกนั้นๆ ตามตำแหน่งของสมาชิกใน Temp ที่ถูกนำไปรวม โดยหากสมาชิกในอะเรย์ Temp ที่นำไปรวมเป็นสมาชิกฝั่งซ้ายก็จะทำการเพิ่มบิต '0' ใน S.cw ของสมาชิกที่เป็นโหนดลูกของสมาชิกอะเรย์ Temp นั้น และเพิ่มบิต '1' เมื่อสมาชิกในอะเรย์ Temp ถูกนำมารวมในฝั่งขวา

การปรับค่าในสมาชิกของอะเรย์ S เริ่มจากการตรวจสอบเงื่อนไข โดยเปรียบเทียบค่าใน Temp.index ว่าน้อยกว่าหรือเท่ากับค่า n หรือไม่ ซึ่งในรอบนี้ค่าใน Temp.index ของสมาชิกทั้งหมดในอะเรย์ Temp ตรงตามเงื่อนไข เพราะทุกสมาชิกในอะเรย์ Temp ยังคงเป็นสมาชิกที่แทนโหนดใบหรือสัญลักษณ์ ดังนั้นจะทำการกำหนดบิตเริ่มต้นให้กับทุกสมาชิกโดยจะเปลี่ยนค่าใน S.index ให้เท่ากับหมายเลขประจำโหนดของโหนดพ่อแม่ใหม่ของสมาชิกนั้นและทำการกำหนดบิตเริ่มต้นใน S.cw ของกับสมาชิกช่องนั้นๆ

ในที่นี้ โหนดพ่อแม่ใหม่หมายเลข 17 เกิดจากการรวมค่าของสมาชิกช่องที่ 1 และ 2 ใน Temp.freq ซึ่งค่าใน Temp.index ของสมาชิกทั้งสองช่องมีค่าน้อยกว่า 16 (n=16) ดังนั้นการปรับค่าในอะเรย์ S จะกระทำโดยการเปรียบเทียบค่าใน Temp.index กับหมายเลขตำแหน่งสมาชิกในอะเรย์ S ซึ่งในที่นี้สมาชิกทั้งหมดในอะเรย์ S มีหมายเลขช่องสมาชิกตรงกับค่าใน Temp.index ดังนั้นจึงเก็บหมายเลขประจำโหนดของโหนดพ่อแม่หมายเลข 17 ไว้ใน S.index ของสมาชิกช่องที่ 1 และ 2 หมายเลข 18 สำหรับสมาชิกช่องที่ 3 และ 4 จนกระทั่งถึงหมายเลข 24 สำหรับสมาชิกช่องที่ 15 และ 16 ตามลำดับ และกำหนดบิตเริ่มต้น '0' ให้กับสมาชิกช่องที่ 1, 3, 5, 7, 9, 11, 13 และ 15 เนื่องจากเป็นสมาชิกที่ตรงกับโหนดลูกฝั่งซ้าย และกำหนดบิต '1' ให้กับสมาชิกช่องที่ 2, 4, 6, 8, 10, 12, 14 และ 16 เนื่องจากเป็นสมาชิกที่ตรงกับโหนดลูกฝั่งขวา

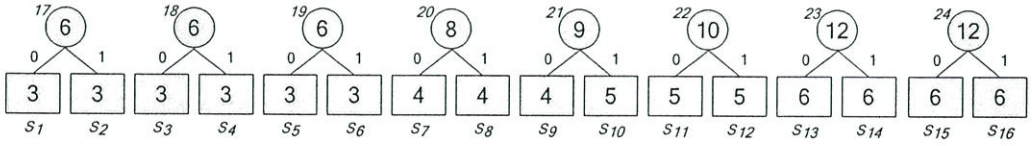


4. ลบสมาชิกในอะเรย์ Temp ที่ถูกนำมารวมเพื่อสร้างโหนดพ่อแม่ใหม่ และนำสมาชิกที่เหลือในอะเรย์ Temp มารวมกับสมาชิกในอะเรย์ IntTemp ด้วยวิธีเมิร์จ-บีเอสอาร์ (Merge-BSR) ที่กล่าวถึงในหัวข้อที่ 2.6.1 และ 2.6.2 ในบทที่ 2 เพื่อให้ได้กลุ่มสมาชิกใหม่ของอะเรย์ Temp ในที่นี้สมาชิกในอะเรย์ Temp ได้ถูกนำไปสร้างโหนดพ่อแม่ใหม่ทั้งหมด ดังนั้นสมาชิกในอะเรย์ Temp จึงโดนลบทั้งหมด และข้อมูลของโหนดพ่อแม่ใหม่ที่สร้างขึ้นจะเก็บไว้ในอะเรย์ IntTemp ซึ่งมีจำนวน 8 สมาชิก ดังนั้นเมื่อนำสมาชิกใน 2 อะเรย์นี้รวมกันจะได้สมาชิกชุดใหม่ในอะเรย์ Temp จำนวน 8 สมาชิกตามสมาชิกในอะเรย์ IntTemp ดังรูป



และปรับค่า c_index ซึ่งจะเก็บหมายเลขประจำโหนดล่าสุด เป็น c_index + $\lfloor a/2 \rfloor$ = 16+8 = 24

ซึ่งในรอบนี้จะได้ผลการสร้างโหนดพ่อแม่ใหม่ในมุมมองต้นไม้ฮัฟแมนดังนี้



<u>รอบที่ 2</u> ในรอบนี้มีอะเรย์ Temp เริ่มต้นคือ

	1	2	3	4	5	6	7	8
Temp.index	17	18	19	20	21	22	23	24
Temp.freq	6	6	6	8	9	10	12	12

1. รวมค่า Temp.freq ของสมาชิกช่องที่ 1 และ 2 นั่นคือ 6 + 6 เก็บไว้ในตัวแปร t ดังรูป

	1	2	3	4	5	6	7	8
Temp.index	17	18	19	20	21	22	23	24
Temp.freq	6	6	6	8	9	10	12	12

t ← (12)

ทุกหน่วยประมวลผลอ่านค่าในตัวแปร t (= 12) เพื่อไปเปรียบเทียบกับค่าใน Temp.freq ที่อยู่ในแต่ละหน่วยประมวลผล ซึ่งในที่นี้มีจำนวนของสมาชิกในอะเรย์ Temp ที่มีค่าในขอบเขต freq น้อยกว่าหรือเท่ากับ 12 อยู่ 8 สมาชิก ดังนั้นค่าในตัวแปร a จึงเท่ากับ 8 ดังรูป

	1	2	3	4	5	6	7	8
Temp.index	17	18	19	20	21	22	23	24
Temp.freq	6	6	6	8	9	10	12	12

t = (12) a = 8

2. จับคู่ของสมาชิกในอะเรย์ Temp จำนวน 4 ($\lfloor a/2 \rfloor = \lfloor 8/2 \rfloor$) คู่ ตามลำดับของตำแหน่งช่องสมาชิกเพื่อสร้างโหนดพ่อแม่ใหม่ แล้วเก็บผลของโหนดพ่อแม่ที่สร้างขึ้นใหม่ไว้ในอะเรย์ IntTemp ดังรูป

	1	2	3	4	5	6	7	8
Temp.index	17	18	19	20	21	22	23	24
Temp.freq	6	6	6	8	9	10	12	12

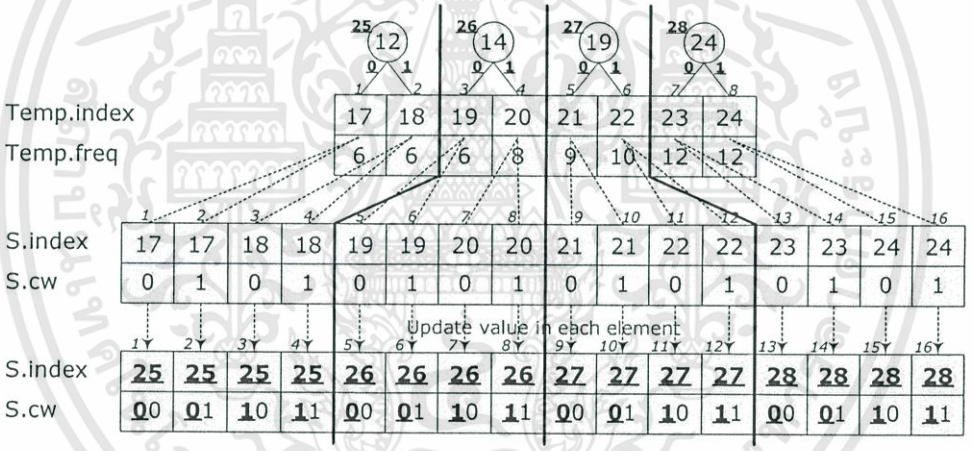
25 (12) 26 (14) 27 (19) 28 (24)

	1	2	3	4
IntTemp.index	25	26	27	28
IntTemp.freq	12	14	19	24

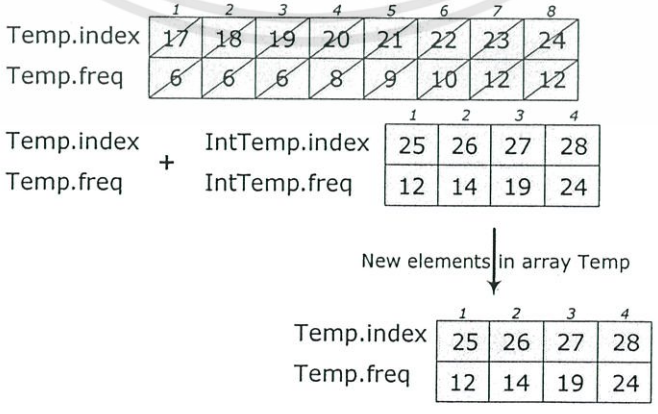
3. ปรับปรุงข้อมูลของสมาชิกในอะเรย์ S ที่มีหมายเลขโหนดใน S.index ตรงกับหมายเลขโหนดลูกของโหนดพ่อแม่ใหม่

ในที่นี้สมาชิกทุกสมาชิกในอะเรย์ Temp ที่ถูกนำมารวมเพื่อสร้างโหนดพ่อแม่ใหม่มีค่า Temp.index มากกว่า n (=16) ทั้งหมด แสดงว่าเป็นโหนดพ่อแม่เดิมจึงทำการเปรียบเทียบหมายเลขประจำโหนด (Temp.index) ของสมาชิกในอะเรย์ Temp กับค่าใน S.index ในทุกสมาชิกของอะเรย์ S ซึ่งหากสมาชิกใดในอะเรย์ S มีค่าใน S.index ตรงกับ Temp.index ใดแสดงว่าสมาชิกในอะเรย์ S เหล่านั้นเป็นโหนดใบภายใต้โหนดที่แทนด้วยสมาชิกในอะเรย์ Temp นั้นๆ ดังนั้นจะทำการปรับค่าในขอบเขต index และ cw ของสมาชิกเหล่านั้นตามโหนดพ่อแม่ใหม่

รอบนี้สมาชิกในอะเรย์ Temp ช่องที่ 1 และ 2 รวมกันได้โหนดพ่อแม่ใหม่ที่มีหมายเลขประจำโหนดคือ 25 ซึ่งค่าใน Temp.index ของสมาชิกช่องที่ 1 และ 2 มีค่ามากกว่า 16 ดังนั้นแสดงว่าสมาชิกทั้ง 2 ช่องจะต้องมีสมาชิกที่เป็นโหนดลูกอยู่ ก็จะทำการเปรียบเทียบค่า Temp.index กับ S.index ทุกสมาชิกว่ามีสมาชิกใดในอะเรย์ S เป็นโหนดใบภายใต้โหนดที่แทนด้วยสมาชิกช่องที่ 1 และ 2 ซึ่งในที่นี้ สมาชิกช่องที่ 1 และ 2 ของอะเรย์ S เป็นสมาชิกที่แทนโหนดใบภายใต้โหนดที่แทนด้วยสมาชิกช่องที่ 1 ของอะเรย์ Temp (หมายเลขประจำโหนดคือ 25) และสมาชิกช่องที่ 3 และ 4 เป็นสมาชิกที่แทนโหนดใบภายใต้โหนดที่แทนด้วยสมาชิกช่องที่ 2 ของอะเรย์ Temp (หมายเลขประจำโหนดคือ 26) ดังนั้นสมาชิกช่องที่ 1, 2, 3 และ 4 ในอะเรย์ S จะถูกเปลี่ยนหมายเลขประจำโหนดพ่อแม่ (S.index) เป็น 25 และเพิ่มบิต '0' ไว้ด้านหน้าบิตเดิมในสมาชิกช่องที่ 1 และ 2 ของ S.cw เพราะโหนดหมายเลข 25 ได้ถูกนำไปเป็นโหนดลูกฝั่งซ้ายของโหนดพ่อแม่ใหม่ สำหรับสมาชิกในช่องที่ 3 และ 4 ของ S.cw ก็จะได้ '1' เพิ่มด้านหน้าบิตเดิมเพราะโหนดหมายเลข 26 ซึ่งแทนด้วยสมาชิกช่องที่ 2 ของอะเรย์ Temp ได้ถูกนำไปเป็นโหนดลูกฝั่งขวาของโหนดพ่อแม่ใหม่ สำหรับสมาชิกช่องที่ 5 ถึง 16 ก็จะถูกปรับปรุงค่าด้วยวิธีการเดียวกันดังรูป

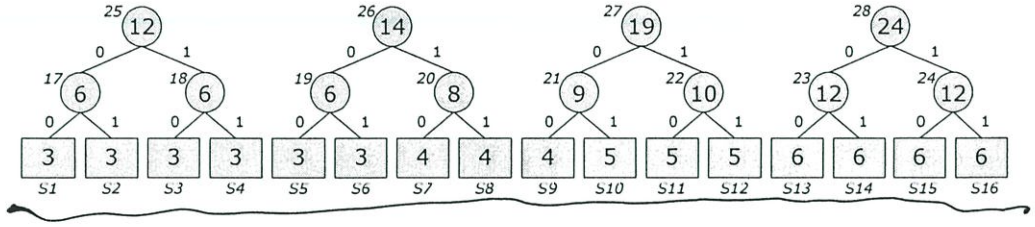


4. ลบสมาชิกในอะเรย์ Temp ที่ได้ถูกนำไปรวมเพื่อสร้างโหนดพ่อแม่ใหม่ แล้วรวมสมาชิกที่เหลือกับอะเรย์ IntTemp ซึ่งในที่นี้ได้สมาชิกใหม่ในอะเรย์ Temp จำนวน 4 สมาชิกดังรูป



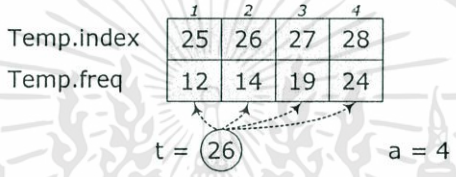
และได้ค่า c_index ซึ่งจะเก็บหมายเลขประจำโหนดล่าสุด เป็น c_index + $\lfloor a/2 \rfloor$ = 24 + 4 = 28

ผลการสร้างโหนดพ่อแม่ใหม่ในรอบที่ 2 ในมุมมองต้นไม้ฮัฟแมน

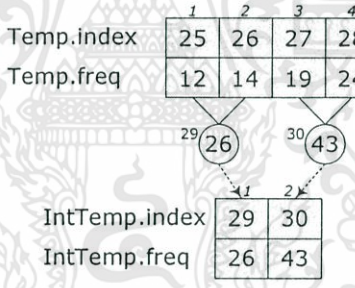


<u>รอบที่ 3</u> ประมวลผลเช่นเดียวกับรอบที่ 2 ดังนี้

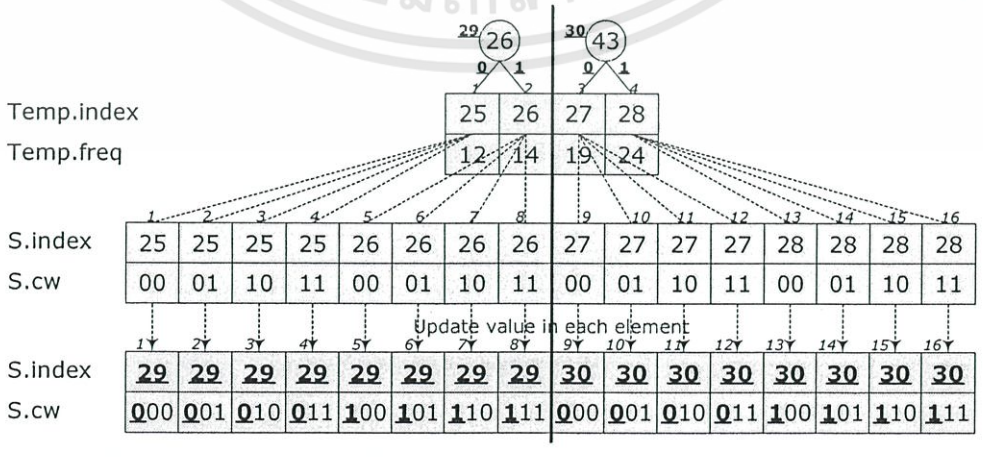
1. รวมค่า Temp.freq ที่มีความถี่ที่น้อยที่สุด 2 ค่า ซึ่งในที่นี้ได้เท่ากับ 26 เก็บไว้ในตัว t แล้วเปรียบเทียบค่า t (= 26) กับทุกค่าใน Temp.freq พร้อมกัน เพื่อหาจำนวนสมาชิกที่มีความถี่น้อยกว่าหรือเท่ากับ 26 เก็บไว้ในตัวแปร a ในที่นี้ได้ 4 สมาชิกดังรูป



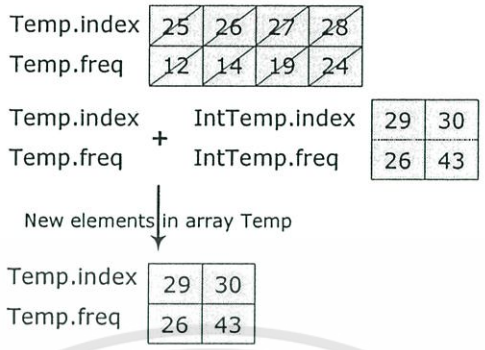
2. จับคู่สมาชิกจำนวน 2 คู่ แล้วเก็บหมายเลขประจำโหนดและความถี่ของโหนดพ่อแม่ใหม่ไว้ในอะเรย์ IntTemp ดังรูป



3. ปรับค่าในอะเรย์ S โดยเปลี่ยนค่าใน S.index ให้ตรงกับหมายเลขประจำโหนดพ่อแม่ใหม่ และ เพิ่มบิตให้กับแต่ละสมาชิกใน S.cw ตามตำแหน่งการรวมของโหนดพ่อแม่เดิมดังรูป

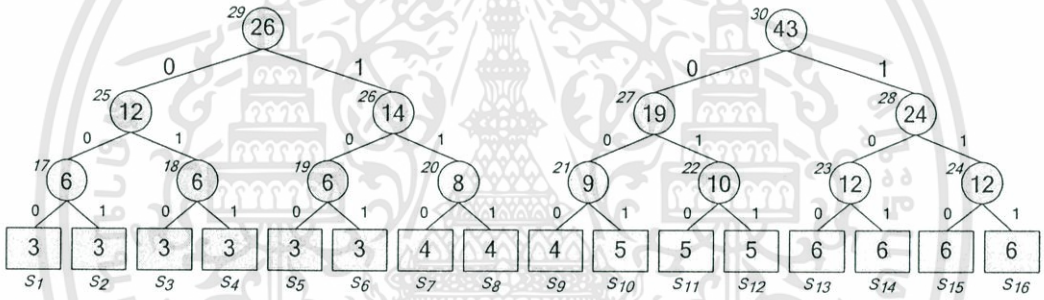


4. ลบสมาชิกในอะเรย์ Temp แล้วรวมสมาชิกที่เหลือกับอะเรย์ IntTemp เพื่อสร้างกลุ่มสมาชิกใหม่ของอะเรย์ Temp ซึ่งจะนำไปสร้างโหนดพ่อแม่ใหม่ในรอบถัดไป



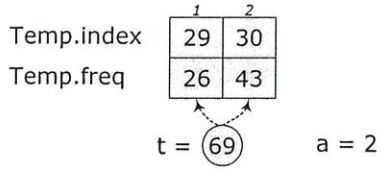
ในรอบนี้ค่า c_index จะถูกปรับเป็น c_index + $\lfloor a/2 \rfloor$ = 28+2 = 30

ผลการสร้างโหนดพ่อแม่ใหม่ในรอบที่ 3 ในมุมมองต้นไม้ฮัฟแมนจะมีโครงสร้างดังนี้

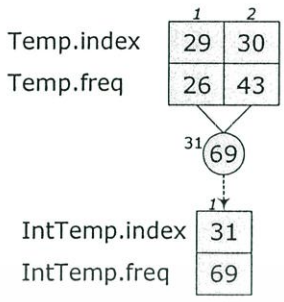


<u>รอบที่ 4</u> ประมวลผลเช่นเดียวกับรอบที่ 2 และ 3 เนื่องจากสมาชิกในอะเรย์ Temp ประกอบด้วยสมาชิกที่แทนโหนดพ่อแม่เดิม ดังนั้นการปรับค่าในอะเรย์ S จะทำการเปรียบเทียบหมายเลขประจำโหนดใน Temp.index กับ S.index ทุกสมาชิก ซึ่งหากสมาชิกใดในอะเรย์ S มีค่าใน S.index เท่ากับ Temp.index ของสมาชิกที่นำไปรวม ก็ทำการเก็บหมายเลขประจำโหนดพ่อแม่ใหม่ไว้ใน S.index ของสมาชิกนั้นๆ และเพิ่มบิตให้กับแต่ละสมาชิกใน S.cw ซึ่งแสดงได้ตามลำดับดังนี้

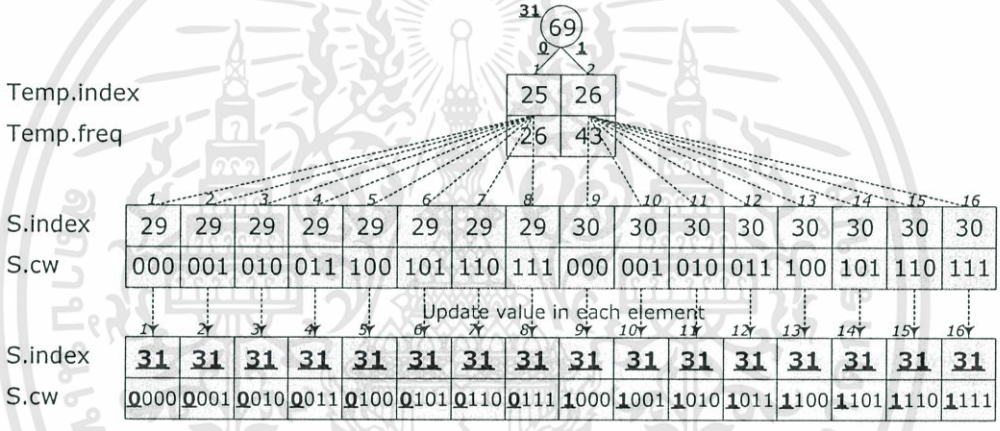
1. รวมค่า Temp.freq ของสมาชิกช่องที่ 1 และ 2 ซึ่งได้เท่ากับ 69 แล้วเปรียบเทียบค่า t (= 69) กับทุกค่าใน Temp.freq พร้อมกัน เพื่อหาจำนวนสมาชิกที่มีความถี่น้อยกว่าหรือเท่ากับ 69 เก็บไว้ในตัวแปร a ในที่นี้ได้ 2 สมาชิกดังรูป



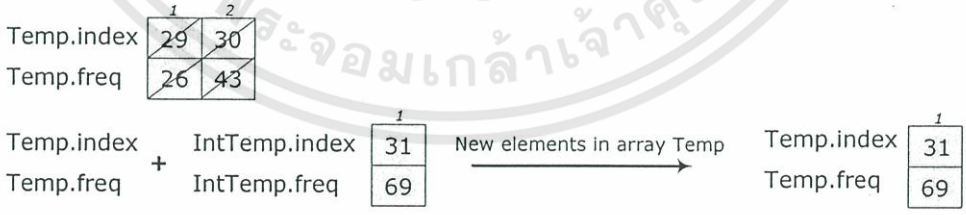
2. จับคู่สมาชิกจำนวน 1 คู่ เก็บหมายเลขประจำโหนดพ่อแม่ใหม่และความถี่ของโหนดพ่อแม่ใหม่ไว้ในอะเรย์ IntTemp



3. ปรับเปลี่ยนหมายเลขประจำโหนดพ่อแม่ของแต่ละสมาชิกในอะเรย์ S ซึ่งเก็บใน S.index ให้ตรงกับหมายเลขประจำโหนดพ่อแม่ใหม่ และ เพิ่มบิตให้กับแต่ละสมาชิกใน S.cw ตามตำแหน่งการรวมของโหนดพ่อแม่เดิม

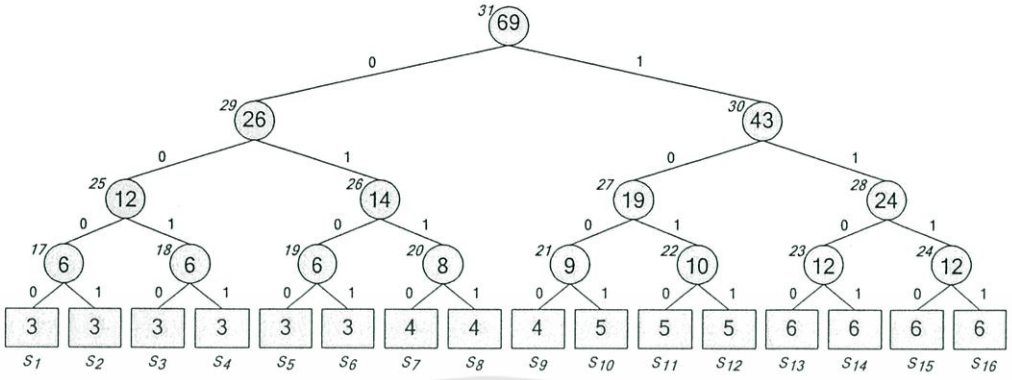


4. ลบสมาชิกในอะเรย์ Temp แล้วรวมสมาชิกกับอะเรย์ IntTemp เพื่อสร้างกลุ่มสมาชิกใหม่ของอะเรย์ Temp ซึ่งในที่นี้มี 1 สมาชิก



และปรับค่าในตัวแปร c_index ซึ่งในรอบนี้จะได้ c_index + $\lfloor a/2 \rfloor$ = 30+1 = 31

ผลการสร้างโหนดพ่อแม่ใหม่ในรอบที่ 4 ในมุมมองต้นไม้ฮัฟแมนจะมีโครงสร้างดังนี้



ในขั้นตอนสุดท้ายของรอบนี้จะได้จำนวนสมาชิกในอะเรย์ Temp เพียง 1 สมาชิก ซึ่งสมาชิกนี้จะเปรียบได้กับโหนดรากเมื่อเทียบในมุมมองต้นไม้ฮัฟแมน ดังนั้นการประมวลผลรอบนี้จึงเป็นการประมวลผลรอบสุดท้าย ซึ่งผลของรหัสที่ใช้แทนสัญลักษณ์แต่ละสัญลักษณ์มีดังนี้

S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

ตัวอย่างที่ 2

การเข้ารหัสของฮัฟแมนด้วยขั้นตอนวิธีแบบขนาน สำหรับข้อมูลที่มีความถี่ไม่ใกล้เคียงกันทำให้มีการกระจายกิ่งในมุมมองต้นไม้ฮัฟแมนแบบไม่สมดุล

สมมุติในชุดข้อมูลมีสัญลักษณ์ S_1, S_2, S_3, ... , S_{11} ตามลำดับ ซึ่งมีความถี่ของการปรากฏสัญลักษณ์นั้นๆ ในชุดข้อมูลดังนี้ 6, 7, 7, 10, 12, 14, 17, 29, 37, 42 และ 99 ตามลำดับ

ขั้นตอนเตรียมข้อมูล

กำหนดค่าให้กับตัวแปร n = 11 และ c_index = 11 และกำหนดค่าเริ่มต้นให้แต่ละสมาชิกในอะเรย์ Temp และอะเรย์ S ดังนี้

n = 11 c_index = 11

	1	2	3	4	5	6	7	8	9	10	11
Temp.index	1	2	3	4	5	6	7	8	9	10	11
Temp.freq	6	7	7	10	12	14	17	29	37	42	99

	1	2	3	4	5	6	7	8	9	10	11
S.index	0	0	0	0	0	0	0	0	0	0	0
S.cw	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil

ขั้นตอนการรวมสมาชิกเพื่อสร้างโหนดใหม่และการปรับเพิ่มรหัสบิต

รอบที่ 1 ประมวลผลตามลำดับดังนี้

1. รวมค่า Temp.freq ที่มีความถี่ที่น้อยที่สุด 2 ค่า เก็บไว้ในตัว t ในที่นี้คือ 13 และเปรียบเทียบค่า t (=13) กับทุกค่าใน Temp.freq พร้อมกัน เพื่อหาจำนวนสมาชิกที่มีความถี่น้อยกว่าหรือเท่ากับ 11 เก็บไว้ในตัวแปร a ในที่นี้ได้ 5 สมาชิกดังนี้

	1	2	3	4	5	6	7	8	9	10	11
Temp.index	1	2	3	4	5	6	7	8	9	10	11
Temp.freq	6	7	7	10	12	14	17	29	37	42	99

t = 13 a = 5

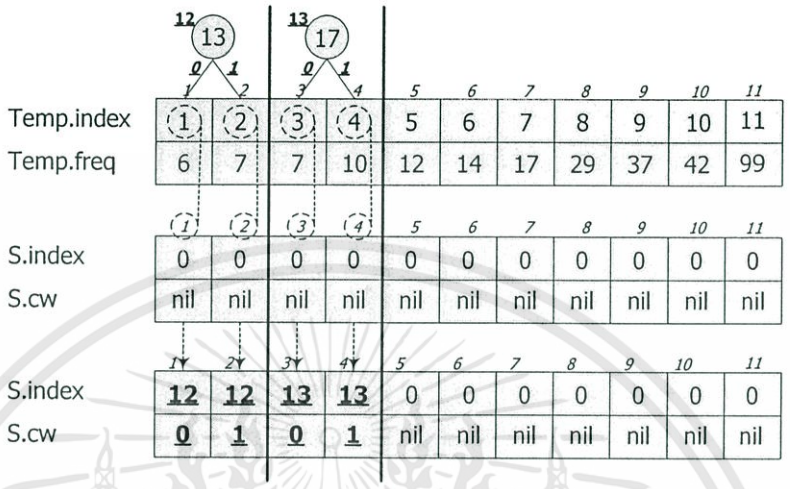
2. จับคู่สมาชิกจำนวน $\lfloor a/2 \rfloor = \lfloor 5/2 \rfloor = 2$ คู่ โดยเก็บหมายเลขประจำโหนดพ่อแม่ใหม่และความถี่ของโหนดพ่อแม่ใหม่ไว้ในอะเรย์ IntTemp

	1	2	3	4	5	6	7	8	9	10	11
Temp.index	1	2	3	4	5	6	7	8	9	10	11
Temp.freq	6	7	7	10	12	14	17	29	37	42	99

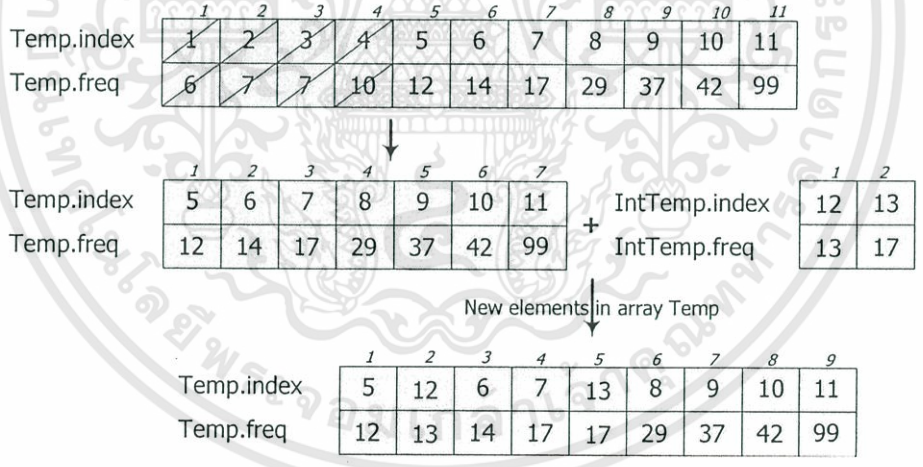
c_index+i =11+1=12 → 13 ← 17 ← c_index+i =11+2=13

	1	2
IntTemp.index	12	13
IntTemp.freq	13	17

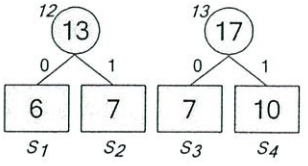
3. รอบนี้ โหนดพ่อแม่ใหม่ใหม่ถูกสร้างมาจากโหนดลูกที่เป็นโหนดใบ เพราะ ค่าใน Temp.index ของสมาชิกที่เป็นโหนดลูกน้อยกว่า n จึงปรับหมายเลขประจำโหนดพ่อแม่ของแต่ละสมาชิกในอะเรย์ S ช่องที่ 1, 2, 3 และ 4 ซึ่งมีตำแหน่งสมาชิกตรงกับค่าใน Temp.index และกำหนดรหัสบิตเริ่มต้นให้ทั้ง 6 สมาชิกด้วย '0', '1', '0' และ '1' ตามลำดับ ดังรูป



4. ลบสมาชิกในอะเรย์ Temp ที่ถูกนำไปรวมแล้ว และนำสมาชิกที่เหลือมารวมกับสมาชิกของอะเรย์ IntTemp เพื่อสร้างกลุ่มสมาชิกใหม่ของอะเรย์ Temp



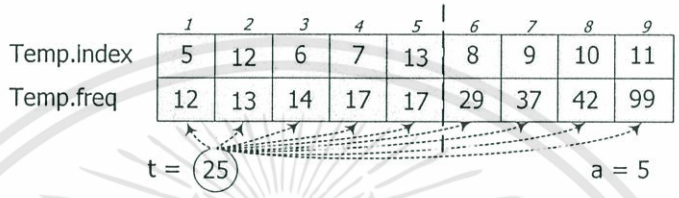
ซึ่งจะได้ผลการสร้างโหนดพ่อแม่ใหม่ในมุมมองต้นไม้ฮัฟแมนดังนี้



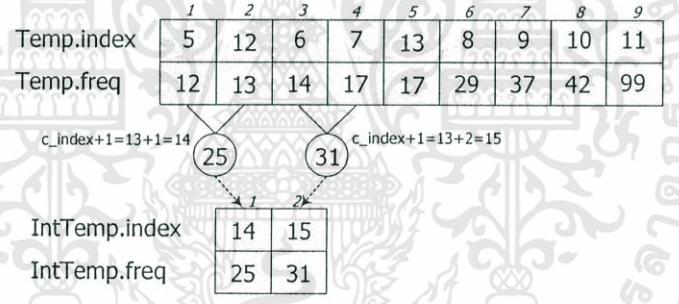
หลังจากสร้างโหนดพ่อแม่ใหม่และปรับค่าในอะเรย์ S แล้ว ค่าในตัวแปร c_index ซึ่งเป็นหมายเลขประจำโหนดพ่อแม่ใหม่จะถูกปรับค่าเพื่อนำไปกำหนดหมายเลขประจำโหนดพ่อแม่ใหม่ในรอบถัดไป โดยคำนวณจาก c_index+$\lfloor a/2 \rfloor$ = 11+$\lfloor 5/2 \rfloor$ = 11+2 = 13 ดังนั้น c_index=13

<u>รอบที่ 2</u> ประมวลผลตามลำดับดังนี้

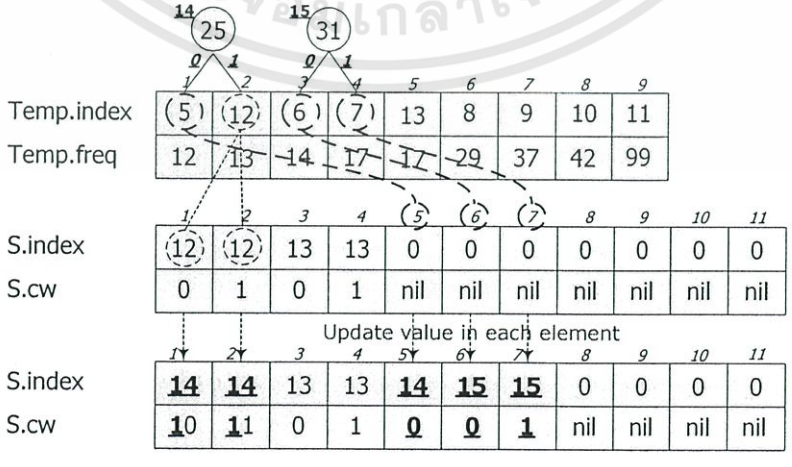
1. รวมค่า Temp.freq ที่มีความถี่ที่น้อยที่สุด 2 ค่า เก็บไว้ในตัวแปร t ในที่นี้คือ 25 และเปรียบเทียบค่า t (= 25) กับทุกค่าใน Temp.freq พร้อมกัน เพื่อหาจำนวนสมาชิกที่มีความถี่น้อยกว่าหรือเท่ากับ 11 เก็บไว้ในตัวแปร a ในที่นี้ได้ 5 สมาชิก



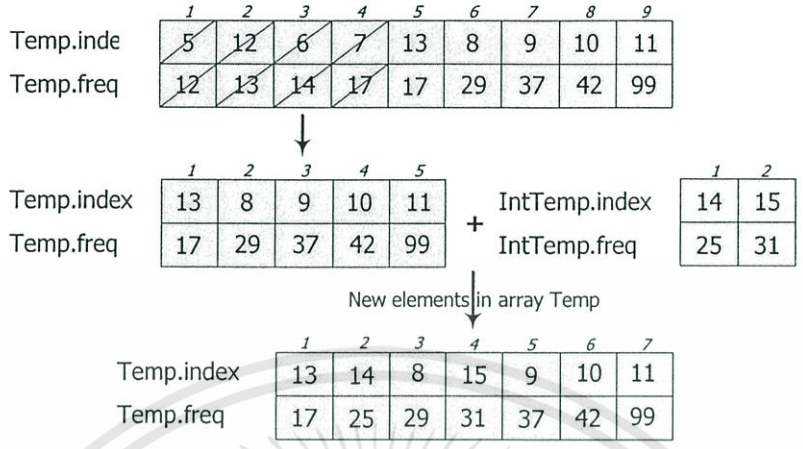
2. จับคู่สมาชิกจำนวน $\lfloor a/2 \rfloor$ คู่ นั่นคือ $\lfloor 5/2 \rfloor$=2 คู่ โดยเก็บหมายเลขประจำโหนดพ่อแม่ใหม่และความถี่ของโหนดพ่อแม่ใหม่ไว้ในอะเรย์ IntTemp



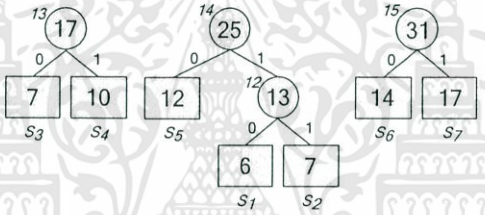
3. ปรับหมายเลขประจำโหนดพ่อแม่ (S.index) และปรับรหัสบิต (S.cw) ให้กับทุกสมาชิกในอะเรย์ S ที่แทนโหนดใบภายใต้โหนดพ่อแม่ใหม่ ดังรูป



4. ลบสมาชิกในอะเรย์ Temp ที่ถูกนำไปรวมแล้วออกจากอะเรย์ Temp แล้วนำสมาชิกที่เหลือมารวมกับสมาชิกของอะเรย์ IntTemp เพื่อสร้างกลุ่มสมาชิกใหม่ของอะเรย์ Temp



ซึ่งจะได้ผลการสร้างโหนดพ่อแม่ใหม่ในมุมมองต้นไม้ฮัฟแมนดังนี้

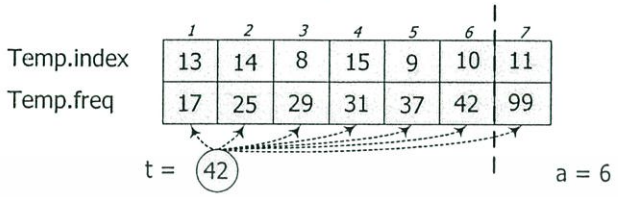


สำหรับการประมวลผลในรอบที่ 2 นี้ มีโหนดพ่อแม่ใหม่ที่ถูกสร้างจากโหนดลูกที่เป็นโหนดพ่อแม่ในรอบที่ 1 อาทิ โหนดพ่อแม่หมายเลข 14 ที่มีความถี่เท่ากับ 25 ถูกสร้างมาจากโหนดลูกหมายเลข 5 กับโหนดลูกหมายเลข 12 ซึ่งโหนดหมายเลข 5 เป็นโหนดใบดังนั้นสมาชิกช่องที่ 5 ของอะเรย์ S จะถูกกำหนดค่าใน S.index ให้เป็น 14 ตามหมายเลขประจำโหนดพ่อแม่ใหม่ และกำหนดบิตใน S.cw ให้เป็น '0' เพราะโหนดลูกหมายเลข 5 ถูกนำมารวมในฝั่งซ้าย สำหรับโหนดหมายเลข 12 เป็นโหนดลูกที่มีหมายเลขประจำโหนดมากกว่า 11 (n = 11) แสดงว่าเป็นโหนดพ่อแม่ในรอบก่อนหน้า ดังนั้นโหนดใบที่อยู่ภายใต้โหนด 12 จะถูกปรับค่าให้ตรงกับโหนดพ่อแม่ใหม่ ซึ่งในที่นี้สมาชิกช่องที่ 1 และ 2 ของอะเรย์ S เป็นสมาชิกที่แทนโหนดลูกของโหนดหมายเลข 12 ดังนั้นค่าใน S.index ของสมาชิก 2 ช่องนี้จะถูกปรับให้เป็น 14 และเพิ่มบิต '1' ไว้ด้านหน้าบิตเดิม เพราะโหนดหมายเลข 12 ถูกนำไปรวมเป็นโหนดลูกฝั่งขวา

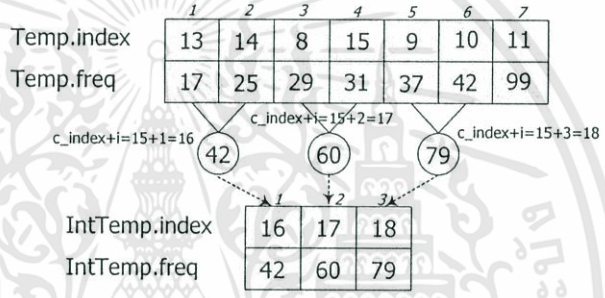
หลังจากปรับรหัสบิตให้กับสมาชิกในอะเรย์ S ที่เกี่ยวข้องกับโหนดพ่อแม่ใหม่แล้ว ค่าในตัวแปร c_index ซึ่งเป็นหมายเลขประจำโหนดพ่อแม่ใหม่จะถูกปรับค่าเพื่อนำไปกำหนดหมายเลขประจำโหนดพ่อแม่ใหม่ในรอบถัดไป โดยคำนวณจาก c_index+$\lfloor a/2 \rfloor$ = 13+$\lfloor 5/2 \rfloor$ = 13+2 = 15 ดังนั้นค่าในตัวแปร c_index =15

<u>รอบที่ 3</u> ประมวลผลตามลำดับดังนี้

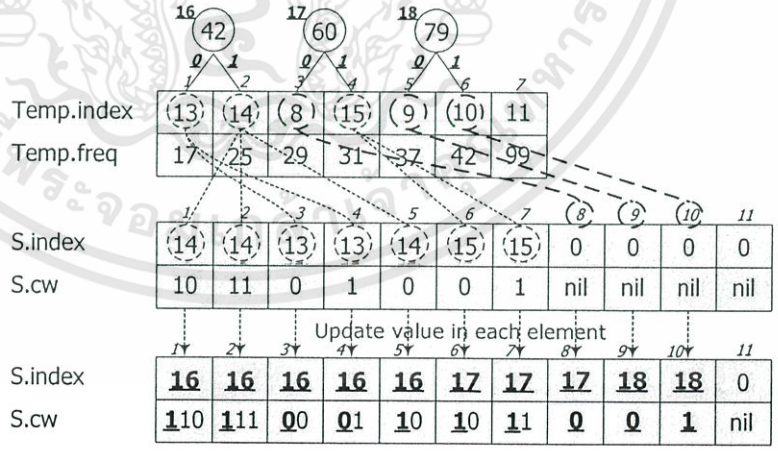
1. รวมค่า Temp.freq ที่มีความถี่ที่น้อยที่สุด 2 ค่า เก็บไว้ในตัวแปร t ซึ่งในที่นี้คือ 42 และเปรียบเทียบค่า t (= 42) กับทุกค่าใน Temp.freq พร้อมกัน เพื่อหาจำนวนสมาชิกที่มีความถี่น้อยกว่าหรือเท่ากับ 42 เก็บไว้ในตัวแปร a ในที่นี้ได้ 6 สมาชิกดังรูป



2. จับคู่สมาชิกจำนวน $\lfloor a/2 \rfloor = \lfloor 6/2 \rfloor = 3$ คู่ โดยเก็บหมายเลขประจำโหนดพ่อแม่ใหม่และความถี่ของโหนดพ่อแม่ใหม่ไว้ในอะเรย์ IntTemp



3. เก็บหมายเลขประจำโหนดพ่อแม่ใหม่ไว้ใน S.index และเพิ่มบิต '0' หรือบิต '1' ใน S.cw ของทุกสมาชิกที่แทนโหนดใบภายใต้โหนดลูกฝั่งซ้ายและฝั่งขวาตามลำดับ ดังรูป

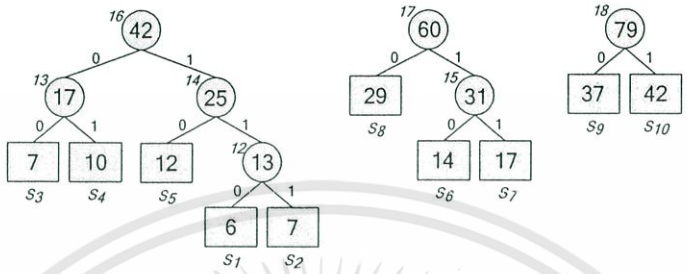


4. ลบสมาชิกในอะเรย์ Temp ที่ถูกนำไปรวมแล้ว ซึ่งในที่นี้เหลือเพียง 1 สมาชิก แล้วนำสมาชิกนั้นมารวมกับสมาชิกของอะเรย์ IntTemp เพื่อสร้างกลุ่มสมาชิกใหม่ของอะเรย์ Temp

ซึ่งได้สมาชิกชุดใหม่ในอะเรย์ Temp ดังนี้

	1	2	3	4
Temp.index	16	17	18	11
Temp.freq	42	60	79	99

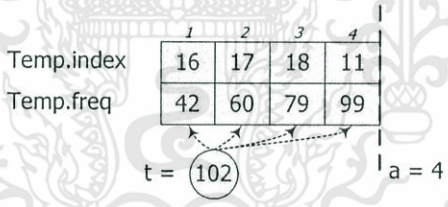
ในรอบนี้จะได้ผลการสร้างโหนดพ่อแม่ใหม่ในมุมมองต้นไม้ฮัฟแมนดังนี้



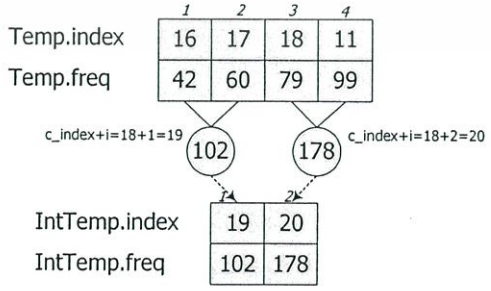
จากนั้นปรับค่าในตัวแปร c_index เป็น c_index + $\lfloor a/2 \rfloor$ = 15+$\lfloor 6/2 \rfloor$ = 15+3 = 18

<u>รอบที่ 4</u> ประมวลผลตามลำดับดังนี้

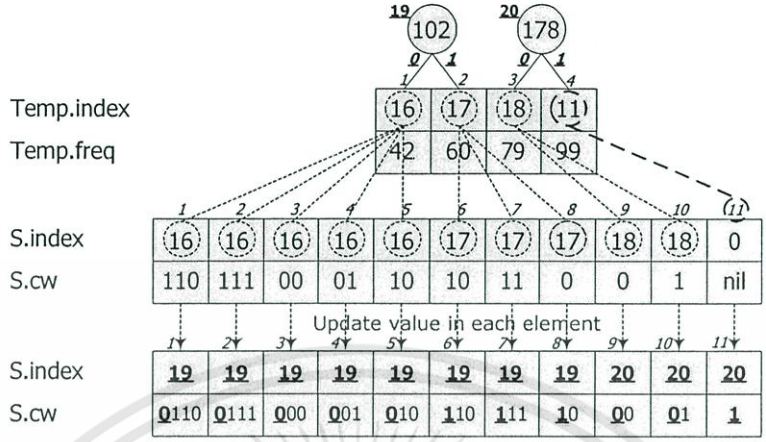
1. รวมค่า Temp.freq ที่มีความถี่ที่น้อยที่สุด 2 ค่าเก็บไว้ในตัว t ในที่นี้คือ 102 แล้วเปรียบเทียบค่า t (=102) กับทุกค่าใน Temp.freq พร้อมกัน เพื่อหาจำนวนสมาชิกที่มีความถี่น้อยกว่าหรือเท่ากับ 102 เก็บไว้ในตัวแปร a ซึ่งในที่นี้ได้ 4 สมาชิก



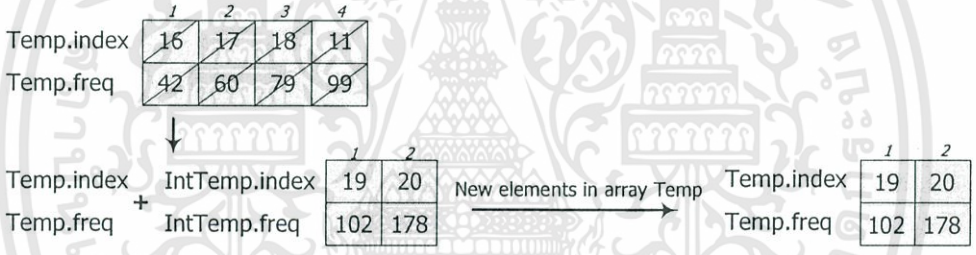
2. จับคู่สมาชิกจำนวน $\lfloor a/2 \rfloor$=$\lfloor 4/2 \rfloor$=2 คู่ โดยเก็บหมายเลขประจำโหนดพ่อแม่ใหม่และความถี่ของโหนดพ่อแม่ใหม่ไว้ในอะเรย์ IntTemp



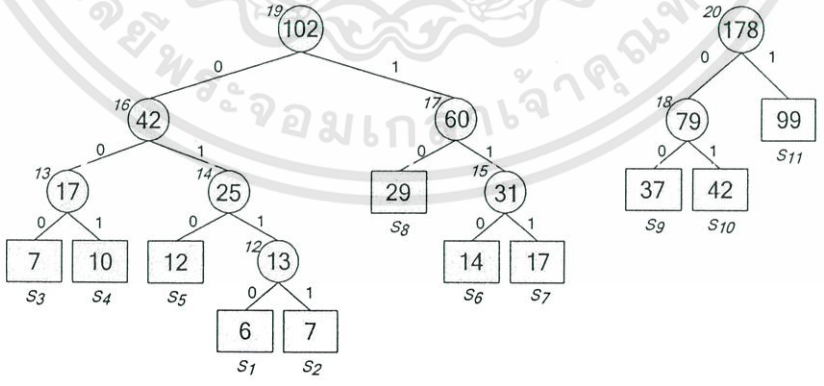
3. เก็บหมายเลขประจำโหนดพ่อแม่ใหม่ไว้ใน S.index และเพิ่มบิต '0' หรือ บิต '1' ใน S.cw ทุกสมาชิกที่แทนโหนดใบภายใต้โหนดลูกฝั่งซ้ายและฝั่งขวาตามลำดับ



4. ลบสมาชิกในอะเรย์ Temp ที่ถูกนำไปรวมแล้ว และนำสมาชิกที่เหลือมารวมกับสมาชิกของอะเรย์ IntTemp เพื่อสร้างกลุ่มสมาชิกใหม่ของอะเรย์ Temp



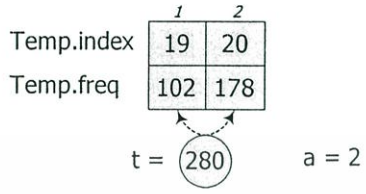
ซึ่งจะได้ผลการสร้างโหนดพ่อแม่ใหม่ในมุมมองต้นไม้ฮัฟแมนดังนี้



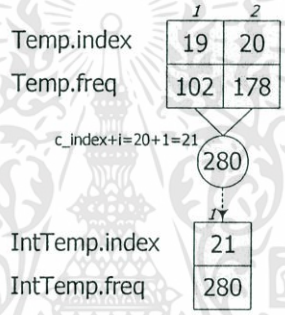
จากนั้นปรับค่าในตัวแปร c_index ซึ่งได้ $c\_index + \lfloor a/2 \rfloor = 18 + \lfloor 4/2 \rfloor = 18 + 2 = 20$

<u>รอบที่ 5</u> ประมวลผลตามลำดับดังนี้

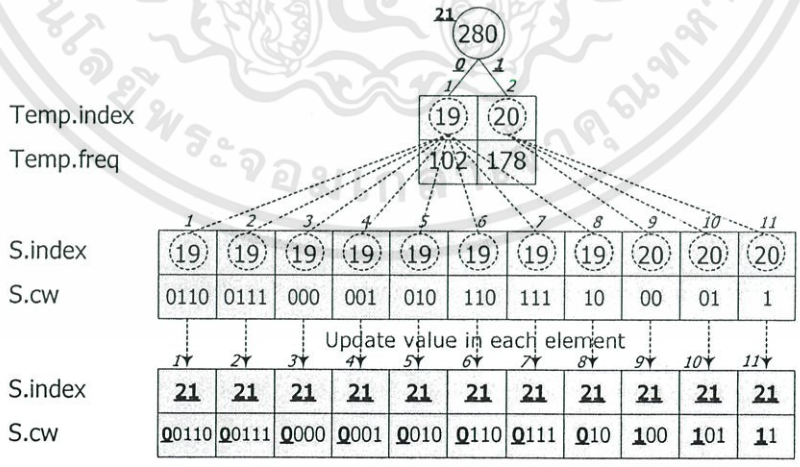
1. รวมค่า Temp.freq ที่มีความถี่ที่น้อยที่สุด 2 ค่า เก็บไว้ในตัว t ซึ่งในที่นี้คือ 280 แล้วเปรียบเทียบค่า t (= 280) กับทุกค่าใน Temp.freq พร้อมกัน เพื่อหาจำนวนสมาชิกที่มีความถี่น้อยกว่าหรือเท่ากับ 280 เก็บไว้ในตัวแปร a ซึ่งในที่นี้ได้ 2 สมาชิก



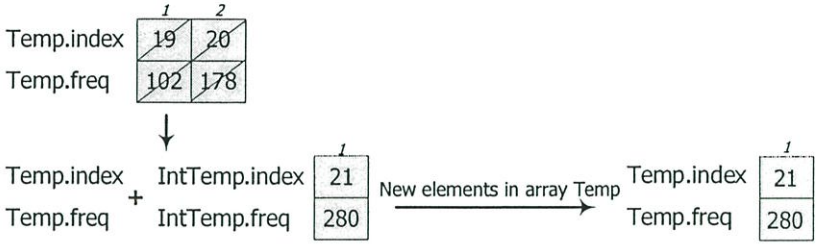
2. จับคู่สมาชิกจำนวน $\lfloor a/2 \rfloor = \lfloor 2/2 \rfloor = 1$ คู่ พร้อมทั้งเก็บหมายเลขประจำโหนดพ่อแม่ใหม่และความถี่ของโหนดพ่อแม่ใหม่ไว้ในอะเรย์ IntTemp



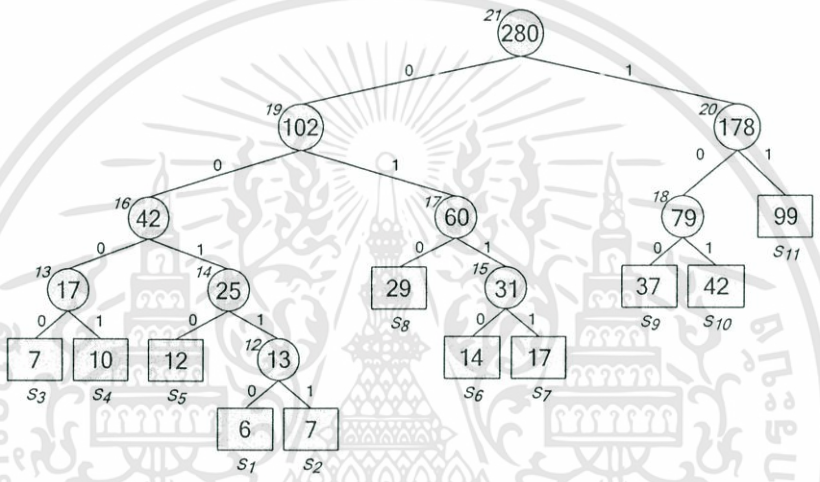
3. เก็บหมายเลขประจำโหนดพ่อแม่ใหม่ไว้ใน S.index และเพิ่มบิต '0' หรือ บิต '1' ใน S.cw ของทุกสมาชิกในอะเรย์ S ที่แทนโหนดใบภายใต้โหนดลูกฝั่งซ้ายและฝั่งขวาตามลำดับ



4. ลบสมาชิกในอะเรย์ Temp ที่ถูกนำไปรวมแล้ว และนำสมาชิกที่เหลือมารวมกับสมาชิกของอะเรย์ IntTemp เพื่อสร้างกลุ่มสมาชิกใหม่ของอะเรย์ Temp



ซึ่งจะได้ผลการสร้างโหนดพ่อแม่ใหม่ในมุมมองต้นไม้ฮัฟแมนดังนี้



ในรอบนี้เป็นการประมวลผลรอบสุดท้ายสำหรับการสร้างรหัสให้กับสัญลักษณ์ในชุดข้อมูลตัวอย่างที่ 2 เพราะจำนวนสมาชิกในอะเรย์ Temp หลังจากการรวมสมาชิกกับอะเรย์ IntTemp มีเพียง 1 สมาชิกซึ่งไม่สามารถจะนำไปรวมกับสมาชิกใดเพื่อสร้างโหนดพ่อแม่ใหม่ได้ และสมาชิกที่มีเพียง 1 สมาชิกในอะเรย์ Temp นั้นเปรียบได้กับโหนดรากในมุมมองต้นไม้ฮัฟแมน นั่นแสดงว่าการสร้างต้นไม้เป็นไปอย่างสมบูรณ์และจะได้รหัสบิตที่ใช้แทนสัญลักษณ์แต่ละสัญลักษณ์ในข้อมูลดังนี้

S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}
00110	00111	0000	0001	0010	0110	0111	010	100	101	11

ภาคผนวก ข.

ผลงานวิจัยที่ได้รับการตีพิมพ์

JCSSE

2008

International Joint Conference on
Computer Science and Software Engineering

The 5th International Joint Conference on Computer Science and Software Engineering (JCSSE2008)

May 7th - 9th, 2008

Department of Computing, Faculty of Science,
Silpakorn University, Thailand

Vol 1 - International Sessions

65th Silpakorn University's Anniversary

NECTEC SIPA CAT TELECOM ธนาคารธนชาต Thanachart Bank IEEE THAILAND SECTION

A New Efficient Parallel Huffman Coding Algorithm

Patcharin Buayen and Jeeraporn Werapun
Department of Mathematics and Computer Science, Faculty of Science,
King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand 10520
E-mail: s9067501@kmitl.ac.th and ksjeerap@kmitl.ac.th

Abstract

In this paper, we propose a new efficient parallel algorithm for constructing Huffman codes on CREW-PRAM model using BSR (Broadcasting with Selective Reduction). Our approach presents the more efficient memory space as well as simplifies the coding process. Time complexity of our efficient proposed parallel algorithm is O(log n) time in the best case and O(n) time in the worst case, which is improved over the existing parallel algorithm (O(log(log(n-1))!)) in best case.

Keywords: Parallel Huffman coding algorithm, CREW-PRAM model, BSR-PRAM.

1. Introduction

D. A. Huffman [1] presented the Huffman algorithm for constructing the minimum redundancy (binary) codes, corresponding to the occurrence probability of each symbol in the (input) data file. Moreover, Huffman coding has been widely using in compression data, images, and video [2]. With dominant feature of the Huffman coding, there are many developed algorithms in sequential models and parallel models.

In sequential approach, R. Hashemian [3] proposed an algorithm to speedup the searching process in the Huffman tree by clustering and combining code-bits for searching in a smaller look-up table. In [4], the authors presented an algorithm for balancing of '0' and '1' bits in encoded bit-streams for discussing the problem of Huffman codes and reversible variable-length codes (RVLCs). In addition, there are many interesting related work about Huffman coding. A technique was presented in [5] which define the condensed Huffman table (CHT) that was smaller than the ordinary Huffman table, and hence performed the faster decoding because of the shorter search in the CHT. The study in [6] and [7] introduced the development for efficient memory in the single side Huffman table (SGHT), corresponding to the single side Huffman tree (SGH-tree).

However, time complexity of the best sequential algorithm for the Huffman codes construction was *O(nlogn)* [8], [9]. Therefore, the parallel approach of Huffman coding was introduced to improve such time complexity. The ES-ParHuff algorithm which shown in [8] constructed the Huffman tree in parallel and required *O(n)* time in the worst case and *O(log(1/p1)loglogn)*, where $p1=w1/\sum_{i=1}^{n} w_i$ time in the best case when the Huffman tree was nearly balanced. Y.-K. Lin and K.-L. Chung [10] proposed a data structure to design a parallel Huffman decoding algorithm in only *O(1)* time. In [9], the authors proposed an efficient parallel algorithm for generating Huffman codes in practical. That algorithm consisted of two phases. First is to compute codeword lengths for all symbols by using an innovation direct parallelism. In the second phase, codewords for all symbols, corresponding to codeword lengths in previous phase were generated in parallel. That parallel algorithm required *O(log(log(n-1))!)* time in the best case and *O(n)* time in the worst case, which was smaller than that of the existing study in [1], [2], and [8]. Both Huffman coding algorithms presented in [8] and [9] applied the process of searching and merging algorithms introduced in [11].

In this paper, we propose a new efficient parallel algorithm for constructing Huffman codes on CREW-PRAM model. We introduce the more efficient memory space and simplify the coding process. In particular, the partial codewords of all symbols are generated during the construction of Huffman tree without computing the codeword lengths. Moreover, we applied the efficient broadcasting, called BSR (Broadcasting with Selective Reduction) for parallel merging [12]. Therefore, time complexity of our efficient parallel Huffman coding algorithm for the construction is only *O(log n)* in the best case when the constructed Huffman tree is nearly balanced and *O(n)* in the worst case when the Huffman tree is in one-sided form.

The rest of this paper is organized as follows. In section 2, we describe the original Huffman coding algorithm and mention the related parallel algorithms. Section 3 illustrates our proposed efficient parallel Huffman coding algorithm along with a practical example. Then performance analysis of our algorithm is introduced in section 4. Finally, conclusion and future study are mentioned in section 5.

2. Related Work

2.1 Huffman Coding Algorithm

The basic idea of the minimum redundancy code algorithm, called "Huffman coding algorithm" [1], [2] is selecting the first two minimum frequencies (or probability of occurrences) of all (n) symbols in the input list ({S_1, S_2, S_3, S_4, S_5,..., and S_n}). For any iteration, a new internal node (or a parent node) is constructed by merging a couple selected (minimum-frequency) nodes from the list. The frequency of the new internal node is computed from the summation of those two selected nodes. Then, the selected nodes are removed from the list, while the new internal node is inserted into the list. The above merging process is reapplied until there is only one node remaining in the list that is the root node. Such a binary tree structure is called the "Huffman tree". In order to construct the "Huffman codes" from this tree, the coding process assigns the left node of any internal node with bit '0' and its right one with bit '1' since the left one is smaller than the corresponding right one. Finally, the (binary) Huffman code of each symbol is generated by searching into the Huffman tree starting from the root until reaching to each leaf node. The feature key of the Huffman coding is that the more frequency symbol, the shorter code assigned.

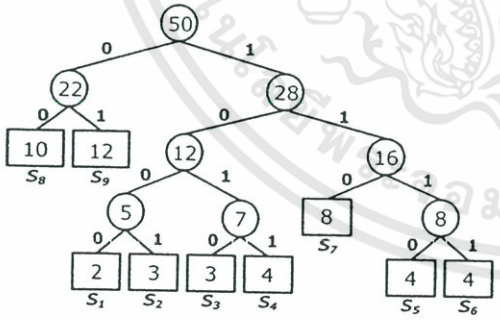


Figure1. An example of the Huffman tree.

For example, a list of all (n = 9) input symbols are {S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, and S_9} and its corresponding ordered frequencies are {2, 3, 3, 4, 4, 4, 8, 10, and 12}. After applying the above Huffman algorithm, the Huffman tree was constructed as shown in Figure 1. The corresponding Huffman codes and their lengths for {S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, and S_9} are {1000, 1001, 1010, 1011, 1110, 1111, 110, 00, and 01} and {4, 4, 4, 4, 4, 4, 3, 2, and 2}, respectively.

2.2 PRAM models and BSR model

The purpose of developing parallel algorithms using more processors is to process in higher speed and less memory space than those of sequential algorithms using only one processor. For simplify parallel algorithms, the PRAM (Parallel Random Access Machine) model is usually applied as an abstract parallel computer model. Therefore to create the generalized parallelism of Huffman coding, most of parallel algorithms were proposed based on the PRAM model [8], [9], [10], [11], [12], [13], [14], and [15]. In general, there are 4 forms of the PRAM model. The first one is EREW (Exclusive Read Exclusive Write) PRAM model, which is originally called Parallel Random Access Computer (PRAC) [13]. In this model, no two processors can access to the same memory location whether reading or writing. The second model is ERCW (Exclusive Read Concurrent Write) PRAM model. This model allows many processors can write into the same memory location at the same time but only one processor can read, and hence it is not popular for developing applications. The next model is CREW (Concurrent Read Exclusive Write) PRAM model, which is popular to develop many applications. This model allows more than one processors to read a value in the same memory location at the same time but only one processor can write. The last one is called CRCW (Concurrent Read Concurrent Write) PRAM model. In this flexible model, it is possible for several processors to access (read or write) a value into the same memory location at the same time [13]. Moreover, the study in [12] and [14] presented the optimal computing result of some parallel applications by using BSR (Broadcasting with Selective Reduction) PRAM model, namely the Broadcast Instruction (BI). The idea of the BSR-PRAM model was presented in [15]. This model showed that its time was faster than that of the CRCW-PRAM model because its relation is many-to-many correspondence between memory locations and processors. Then [12] showed optimal parallel merging algorithm in $O(1)$ time based on BSR which presented in [15].

3. Proposed Parallel Huffman Coding Algorithm

We propose a new efficient parallel Huffman algorithm on CREW-PRAM model. Our algorithm maximum requires n processors for an input file of n symbols. In our efficient approach, we can simply

generate both partial codewords of all symbols during the construction of the Huffman tree, represented by using an array. We also use the more efficient memory space and the simpler coding process since our approach does not require the computing of the codeword lengths before generating the codewords. In addition, in order to improve our time complexity we apply the efficient broadcasting, called BSR, introduced in for parallel merging [12] in our approach. Thus, this new proposed algorithm can process in *O(log n)* time in the best case when the Huffman tree is nearly balanced since in each cycle all pairs of elements can be melded in parallel and *O(n)* time in the worst case when the Huffman tree is in one-sided form since in each cycle only one pair of elements can be melded.

Our parallel algorithm consists of two main steps, which are the initial step and the construction of codeword step. In the first step "preparing input data" in parallel by using n processors (see Figure. 2), the process starts with transferring frequencies and a sequence number of all symbols into temporary memory called a *Temp* queue structure. Then, assign initial values to *S* queue structure for keeping codewords. Create another *IntTemp* queue structure, having similar structure to that of the *Temp* queue, for keeping detail of new internal nodes which created in each round. Finally, assign the value of n to *c_index*. The *c_index* represents an index (or label) of each of internal nodes, which is constructed in each round.

The next main step (see Figure. 3) is to create the Huffman tree (represented in array structure) and corresponding codewords iteratively until only one element remaining in the *Temp* queue that is the root of the complete construction of the Huffman tree. In the construction of codeword step, the parallel generation of each new level in the tree is generated and the corresponding partial codewords of all symbols are increased one bit.

In each round, the process consists of two substeps, which are creating internal (combined) nodes and assigning partial codewords. Initially in the first one (creating internal nodes), processor P_1 computes a summation of two minimum values from *Temp.freq* and stores the result in a variable t and then broadcasts t to all processors. Next, only processors that have the number of elements in *Temp.freq* no more than t are active in order to compute the partial codeword of each symbol in parallel. For simplify the algorithm, assume that a number of active processors, according to that condition are equal to a. For all i=1 to $\lfloor a/2 \rfloor$ of active processors (P_i), new internal nodes are constructed by computing the summation of *Temp.freq* at location i*2-1 and i*2 then store them into *IntTemp.freq* at location i. Corresponding indices of all new internal nodes are *c_index*+i, which is the label of each new internal nodes, and store them into *IntTemp.index*.

```
n ← number of symbols
c_index ← n
Forall Processors Pi (i=1 to n) pardo
  Temp.index[i]← i
  Temp.freq[i] ← frequency of symbol[i]
  S.index[i] ← 0; S.cw[i] ← nil
  Create IntTemp queue which its length equal to ⌊n/2⌋
End For
```

Figure 2. The initial step.

```
While (elements of Temp > 1) do
  P1 compute t ← sum two minimums of Temp
     and broadcast t to all processors
  Assume a ← number of elements which their
     values in Temp.freq are not more than t
  Forall Processors Pi (i=1 to ⌊a/2⌋) pardo
    IntTemp.index[i] ← c_index+i
    IntTemp.freq[i]←Temp.freq[i*2-1]+Temp.freq[i*2]
    // construct partial codewords
    // 1. Assign '0' to the left sub-tree
    If (Temp.index[i*2-1] ≤ n)  // leaf node
      S.index[Temp.index[i*2-1]]←IntTemp.index[i]
      S.cw[Temp.index[i*2-1]]  ← '0'
    Else    // internal node
     Forall Processors Px(x=1 to length(S)) pardo
      If (S.index[x] = Temp.index[i*2-1])
        S.index[x] ← IntTemp.index[i]
        S.cw[x] ← inserts '0' in front of old bits
      End If
     End For
    End If
    // 2. Assign '1' to the right sub-tree
    If (Temp.index[i*2] ≤ n)  // leaf node
      S.index[Temp.index[i*2]] ← IntTemp.index[i]
      S.cw[Temp.index[i*2]]  ← '1'
    Else    // internal node
     Forall Processors Px(x=1 to length(S)) pardo
      If (S.index[x] = Temp.index[i*2])
        S.index[x] ← IntTemp.index[i]
        S.cw[x]← inserts '1' in front of old bits
      End If
     End For
    End If
  End For
  c_index ← c_index+ ⌊a/2⌋
  Delete all elements in Temp that were melded
  Temp ← Merge_BSR(Temp, IntTemp)
  Delete all elements in IntTemp
End While
```

Figure 3. The construction of codeword step.

Next in the second substep (constructing partial codewords), the parallel process starts by checking the indices of concurrent elements, that are melded. If each combined node is left node and if its *Temp.index* is more than n (that node is internal node) then bit '0' is inserted in front of its (old) sequence of bits, stored in the *S.cw* queue and copy the internal node's index into its *S.index*. Otherwise (or *Temp.index* ≤ n), it is a leaf node then we replace '0' into its *S.cw* and copy the internal node's index into its *S.index*. For all concurrent elements that are

right nodes, the similar process is performed accept assigning '1' bit (for the right node) instead of '0' bit (for the left node). Finally, all melded elements in the *Temp* queue are deleted whereas all elements in the *IntTemp* queue are merged into the *Temp* queue by applying the optimal merging algorithm on BSR [12], after that all elements in *IntTemp* are deleted.

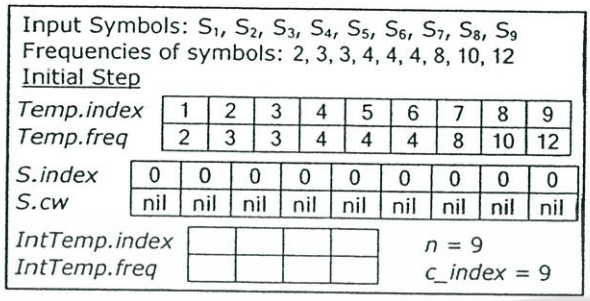


Figure 4. An example shows the initial step.

For example, the list of input symbols is the same as that of the previous example presented in section 2.1. Start at initial step (see Figure 4) by transferring all frequencies of symbols into *Temp.freq* and assigning an index for each element in *Temp.index*. Set all elements' values of *S.index* to be 0 and those of *S.cw* to be "*nil*" and construct *IntTemp* queue, which its length is equal to $\lfloor n/2 \rfloor$. Assign the initial value n=9 to *c_index*.

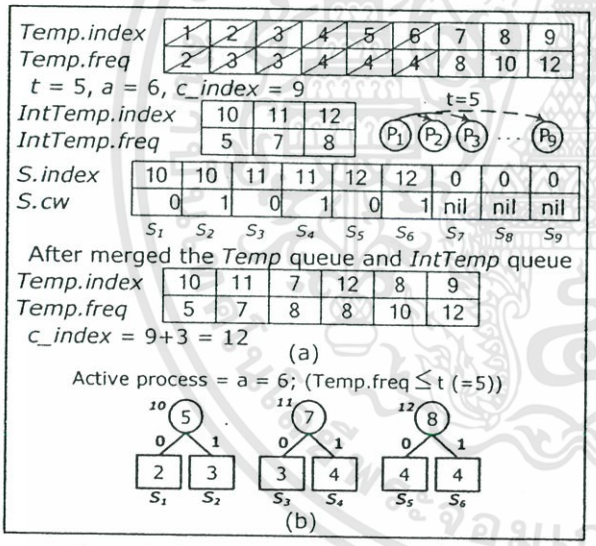


Figure 5. An example shows the 1st iteration of construction of codeword step.

In the first round of the construction of codewords (Figure 5), first processor (P_1) computed a summation of two minimum elements (2 and 3) of *Temp.freq* and the result was stored in t (=2+3=5) and t was broadcasted to all processors. Then, every processor compared the receiving t with its frequency in *Temp.freq*. Assume a number of elements, which their values were no more than t was equal to a, and hence such a value of t was 5 and a value of a was 6. Next, all $\lfloor a/2 \rfloor$ processors combined a couple elements of *Temp.freq* for constructed a new internal node and stored the combined results (2+3=5, 3+4=7, 4+4=8) into *IntTemp.freq* and assigned corresponding index (*c_index*+i), namely, 9+1=10, 9+2=11, 9+3=12 and stored them into *IntTemp.index*. These processors constructed partial codewords of corresponding symbols that were melded frequencies, according to the following conditions. If merged elements are not internal node (or its *Temp.index* $\leq n$), then replace '0' into *S.cw* for the left node and replace '1' into *S.cw* for the right node and also copy indices of these new internal nodes (holded in *IntTemp.index*[i]) into *S.index*. Thus in this example, the contents of *S.index* and *S.cw* of the first symbol (S1) were 10 (*c_index*+i = 9+1=10) and '0' because it was the left node of the new internal which its index are 10 and symbol S2 were10 and '1' because it was the right node of the same internal node. In this round, first six elements in *S.index* were 10, 10, 11, 11, 12, and 12 and its *S.cw* were '0', '1', '0', '1', '0',and '1'. Finally, elements which melded from *Temp.freq* and *Temp.index* were deleted from *Temp* and elements of *IntTemp* were inserted into *Temp* for computing in the next round. All elements in *IntTemp* queue were deleted and updated the value of *c_index* (since *c_index*= *c_index*+$\lfloor a/2 \rfloor$).

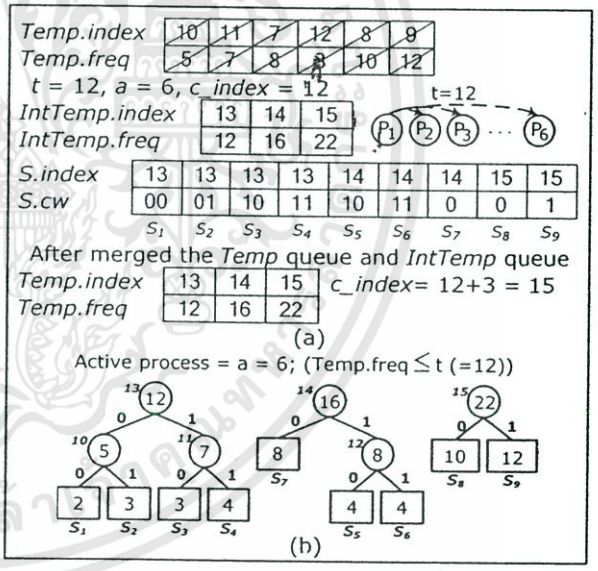


Figure 6. An example shows the 2nd iteration of construction of codeword step.

In the 2nd round, the process performed the same as that of the 1st round for leaf nodes; except some elements of *Temp.freq* melded were internal nodes since their *Temp.index* were more than n (= 9). Thus, in parallel for all processors (P_x), if each child of each new internal node is left node (since *S.index*[x]= *Temp.index*[i*2-1]), then the process will insert '0' in

front of their old bits in *S.cw* or insert '1' if they are right nodes (since *S.index*[*x*]=*Temp.index*[*i**2]) and replaced the index of their internal nodes into *S.index* (i.e., 13, 13, 13, and 13 in the 1st to 4th locations of *S.index*). Insert '0' in front of old bits of the 1st and 2nd elements in *S.cw* because their parent nodes were left nodes. Insert '1' in front of old bits of the 3rd and 4th elements in *S.cw* because their parent nodes were right nodes and hence partial codewords of the 1st to 4th symbols were "00", "01", "10", and "11". Similar process was performed for the 5th to 9th elements (see Figure 6).

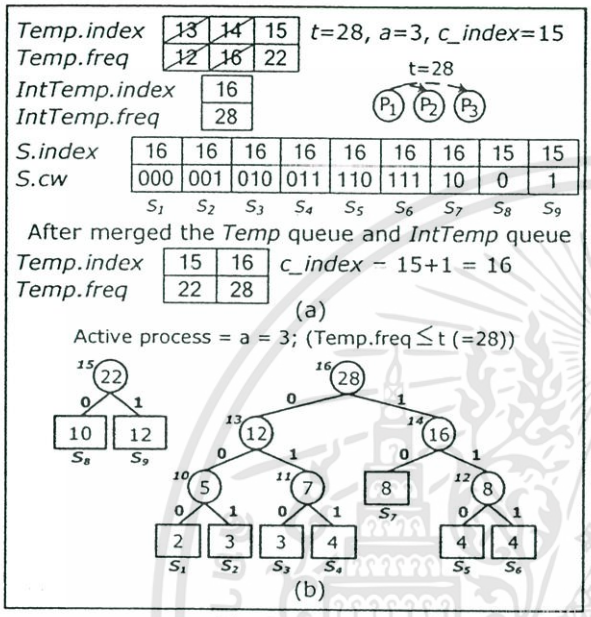


Figure 7. An example shows the 3rd iteration of construction of codeword step.

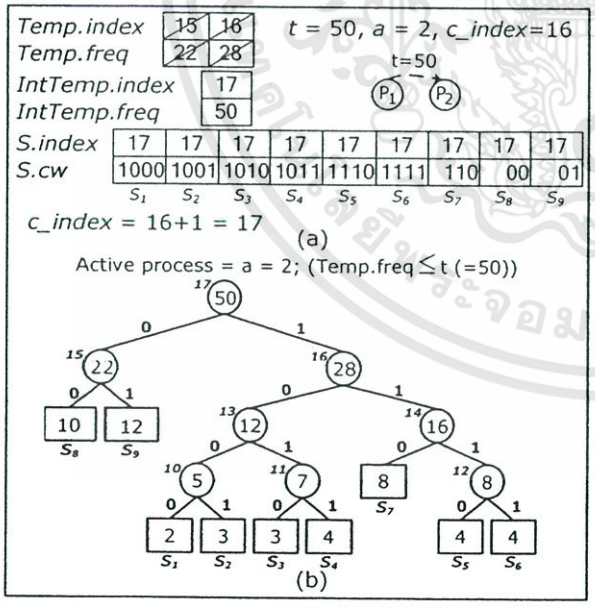


Figure 8. An example shows the 4th iteration of construction of codeword step.

Next, the same process was reapplied in the 3rd round (see Figure7) until there was only one element in *IntTemp* that was the root node in the 4th round (see Figure 8). In each iteration, the corresponding subtrees, performed in parallel with a active process, were also illustrated (in part (b)) including partial codewords that are generated.

4. Performance Analysis

This section illustrates the time complexity analysis of the proposed parallel Huffman coding algorithm, as shown in the following theorems.

Theorem 1 The initial step (see Figure 2) performs in parallel in *O(1)* time.

Proof. In the data preparation step, the process performs by n processors for n input symbols. Clearly, assign the index of each symbol to each corresponding processor in parallel is *O(1)* time. Transfer the frequency of each symbol is also *O(1)* in parallel. Finally, assign the initial value in the *S* queue (into fields *index* and *cw*) can be processed in parallel and hence they can perform in *O(1)* time.

Theorem 2 Each cycle for constructing Huffman code (see Figure 3) performs in parallel in *O(1)* time.

Proof. Each cycle of this algorithm contains many instruction sets, which run in constant time. First, processor P_1 computes the summation of two minimum frequencies of *Temp.freq* and stores in t in *O(1)* time. Then t is broadcasted in *O(1)* time. All processors compare the receiving t with all elements in *Temp.freq* also take *O(1)* time. Next, all processors P_i (i=1 to $\lfloor a/2 \rfloor$) process the following operations in *O(1)* time, where a represents a number of elements, which their values are no more than t. Such operations are constructing a new internal node, assigning 0 into *IntTemp.index*, and also making decision either left or right sub-trees, and inserting the corresponding bit ('0' or '1') in front of its old bit-strings. Also the decision of either leaf or internal node operation for computing partial codewords is also *O(1)* time. Finally, time to update the value of *c_index* and time to parallel delete all elements in *IntTemp* are *O(1)*. Time to merge *IntTemp* into *Temp* using BSR algorithm presented in [12] is *O(1)* time.

Theorem 3 The total time complexity of the performed parallel Huffman coding algorithm runs in *O(log n)* in the best case and *O(n)* in the worst case.

Proof. Assume the construction of all codewords (in Figure 3) performs in L cycles, which is equal to the number levels in the completed Huffman tree. Therefore,

total time complexity is $O(L)$, computed from $O\left(\sum_{i=1}^{L}(1)\right)$ since each level requires $O(1)$ time (proved in Theorem 2).

In the best case, in each cycle all pairs of elements in *Temp* can be melded in parallel. Therefore, the process always has a half of elements ($n/2$, $n/4$, $n/8$, ..., 2, 1) remaining in *Temp* for the next round or there are $\log n$ rounds to process until only one element remaining in *Temp*. In this case, the Huffman tree is nearly balanced. Thus time complexity of the best case is $O(\log n)$, derived as follows:

$$O\left(\sum_{i=1}^{\log n}(1) = 1_1 + 1_2 + 1_3 + \ldots + 1_{\log n}\right) = O\left(\sum_{i=1}^{\log n}(1) = \log n\right)$$

In the worst case, in each cycle there is no more than one pair elements of *Temp* can be melded. Hence, the number of elements in *Temp* is decreased by one (n-1, n-2, n-3, ..., 3, 2, 1) for the next round or there are n-1 rounds to process until only one element remaining in *Temp*. In this case, the Huffman tree is in one-sided form. Thus time complexity of the worst case is $O(n)$ time.

Finally, we showed the comparison of our algorithm with existing algorithms in Table 1.

Table 1. The feature of the comparison of 3 algorithms.

Algorithms	Space Complexity	Time Complexity
ES-ParHuff	-	O(log(1/p1) loglogn)
Two-phase practical parallel	7 arrays	O(log(log(n-1))!)
Our algorithm	3 arrays	O(log n)

5. Conclusion and future work

We propose a new parallel Huffman coding algorithm on CREW-PRAM model using BSR. Our new approach has the following advantages. First, our parallel algorithm has the more simplified coding process and the more efficient utilized memory space than those of the existing parallel Huffman coding algorithms. Second, improved time complexity is only $O(\log n)$ in the best case and $O(n)$ in the worst case for n symbols using n processors.

In the future work, we will apply our parallel Huffman coding algorithm to speed up the process in other related applications such as the encryption data (in the binary-code form) of the security study.

Acknowledgments

The authors deeply thank you to King Mongkut's Institute of Technology Ladkrabang (KMITL) Bangkok, Thailand for the academic support and the research for a financial support.

6. References

[1] D.A. Huffman, "A method for the construction of minimum redundancy codes", Proc. IRE., vol. 40, Sep. 1952, pp. 1098-1101.

[2] T.C. Bell, J.G. Cleary and I.H. Witten, Text Compression, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[3] R. Hashemian, "Memory Efficient and High-Speed Search Huffman Coding", IEEE Trans. Comm., vol. 43, no. 10, Oct. 1995, pp. 2576-2581.

[4] J.-Y. Lin, Y. Liu and K.-C. Yi, "Balance of 0, 1 Bits for Huffman and Reversible Variable-Length Coding", IEEE Trans. Comm., vol. 52, no. 3, Mar. 2004, pp. 359-361.

[5] R. Hashemian, "Condensed Table of Huffman Coding, a New Approach to Efficient Decoding", IEEE Tran. Comm.,Vol. 52, No.1, Jan. 2004, pp. 6-8.

[6] Y.-J. Chuang and J.-L. Wu, "An SGH-Tree Based Memory Efficient and Constant Decoding Algorithm for Huffman Codes", Proc. International on Intelligent Multiple, Video and Speech Processing, Oct. 2004, pp. 374-377.

[7] S.-W. Wang, S.-C. Chuang, C.-C. Hsiao, Y.-S. Tung and J.-L.Wu, "An Efficient memory construction scheme for and arbitary side growing huffman table", IEEE ICME, 2006.

[8] R.L.Milidiu, E.S.Laber and A.A.Pessa, "A work efficient parallel algorithm for constructing Huffman codes", Proc. Data Compression Conference, Mar. 1999.

[9] S.A. Ostadzadeh, B.M. Elahi, Z.Z. Tabrizi, M.A. Moulavi and K. Bertels, "A Two-phase Practical Parallel Algorithm for Construction of Huffman Codes", 3rd ICNPSC, Morehouse College, IFNA, Atlanta, GA, USA, Aug. 2006.

[10] Y.-K. Lin and K.-L. Chung, "A space-efficient Huffman decoding algorithm and its parallelism", ELSEVIER on Theoretical Computer Science, 2000, pp. 227-238.

[11] C.P. Kruskal, "Searching, Merging and Sorting in Parallel Computation", IEEE Trans., vol. c-32, no. 10, Oct. 1983, pp. 942-946.

[12] L. Xiang and K. Ushijma, "Optimal Parallel Merging Algorithms on BSR", IEEE, 2000.

[13] Y.D. Lyuu, Information dispersal and parallel computation (Cambridge International Series on Parallel Computation:3), Cambridge University Press, 1992.

[14] L. Xiang, K. Ushijiam, K. Cheng, J. Zhao, and C. Lu, "O(1) Time algorithm on BSR for Constructing a Binary Search Tree with Best Frequencies", Springer-Verlag Berlin Heidelberg, 2004, pp. 218-225.

[15] L.F. Lindon and S.G. Akl, "An Optimal Implementation of Broadcasting with Selective Reduction", IEEE Trans. on Parallel and Distributed systems, vol. 4, no. 3, Mar. 1993.

ประวัติผู้เขียน

ชื่อ – สกุล	นางสาวพัชรินทร์ บัวเย็น
วัน เดือน ปีเกิด	19 พฤศจิกายน 2520
ที่อยู่	272/103 หมู่ 2 ถนนเชียงใหม่-แม่โจ้ ตำบลหนองจ๊อม อำเภอสันทราย จังหวัดเชียงใหม่ 50210
ประวัติการศึกษา	2542 วิทยาศาสตรบัณฑิต สาขาวิทยาการคอมพิวเตอร์ สถาบันราชภัฏลำปาง
	2544 ประกาศนียบัตรวิชาชีพครู มหาวิทยาลัยสุโขทัยธรรมาธิราช

ประสบการณ์การทำงานและผลงานวิจัย

พ.ศ.2542-2543 ตำแหน่งนักวิชาการคอมพิวเตอร์ ศูนย์คอมพิวเตอร์ มหาวิทยาลัยราชภัฏเชียงใหม่

พ.ศ.2543-2544 ตำแหน่งครูประจำศูนย์คอมพิวเตอร์ท๊อต และ โรงเรียนศิริมาตย์เทวี จังหวัดเชียงราย

พ.ศ.2544-2547 ตำแหน่งอาจารย์ประจำแผนกคอมพิวเตอร์ธุรกิจ โรงเรียนพณิชยการเชียงใหม่

ผลงาน

- งานวิจัยชั้นเรียน หัวข้อ “กิจกรรมแบบศูนย์การเรียนมีผลต่อพฤติกรรมการเรียนในรายวิชาคอมพิวเตอร์เบื้องต้น”
- งานวิจัยชั้นเรียน หัวข้อ “การประเมินตามสภาพจริง (Authentic Assessment) เพื่อบูรณาการการเรียนรู้ของนักศึกษาในรายวิชา การเขียนโปรแกรมภาษาซี”
- งานวิจัยชั้นเรียน หัวข้อ “การนำการประเมินตามสภาพจริง (Authentic Assessment) มาประยุกต์เพื่อพัฒนาการเรียนรู้ของนักศึกษาในรายวิชาความรู้เกี่ยวกับระบบปฏิบัติขั้นต้น”
- ออกแบบนวัตกรรมการเรียนการสอน ชื่อ “วงจรการวิเคราะห์งาน (Analysis Cycle Model)”

รางวัลเกียรติคุณ

- รางวัลผลงานวิจัยในชั้นเรียนดีเด่น
- โล่ประกาศเกียรติคุณด้านอาจารย์ที่ปรึกษาดีเด่น

	- รางวัลชนะเลิศในโครงการเสริมสร้างระเบียบวินัย
พ.ศ.2547-2549	อาจารย์พิเศษประจำภาควิชาวิทยาการคอมพิวเตอร์และ
	ภาควิชาคอมพิวเตอร์ศึกษา มหาวิทยาลัยราชภัฏภูเก็ต
พ.ศ.2548	ตำแหน่งหัวหน้างานสหกิจศึกษา
	มหาวิทยาลัยสงขลานครินทร์ วิทยาเขตภูเก็ต