การพัฒนาควิกซอร์ตแบบขนานด้วยโอเพนเอ็มพี

PARALLEL QUICKSORT DEVELOPMENT USING OPENMP

พัสกร จุลพล

สุรพงศ์ เท่าเทียมตน

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

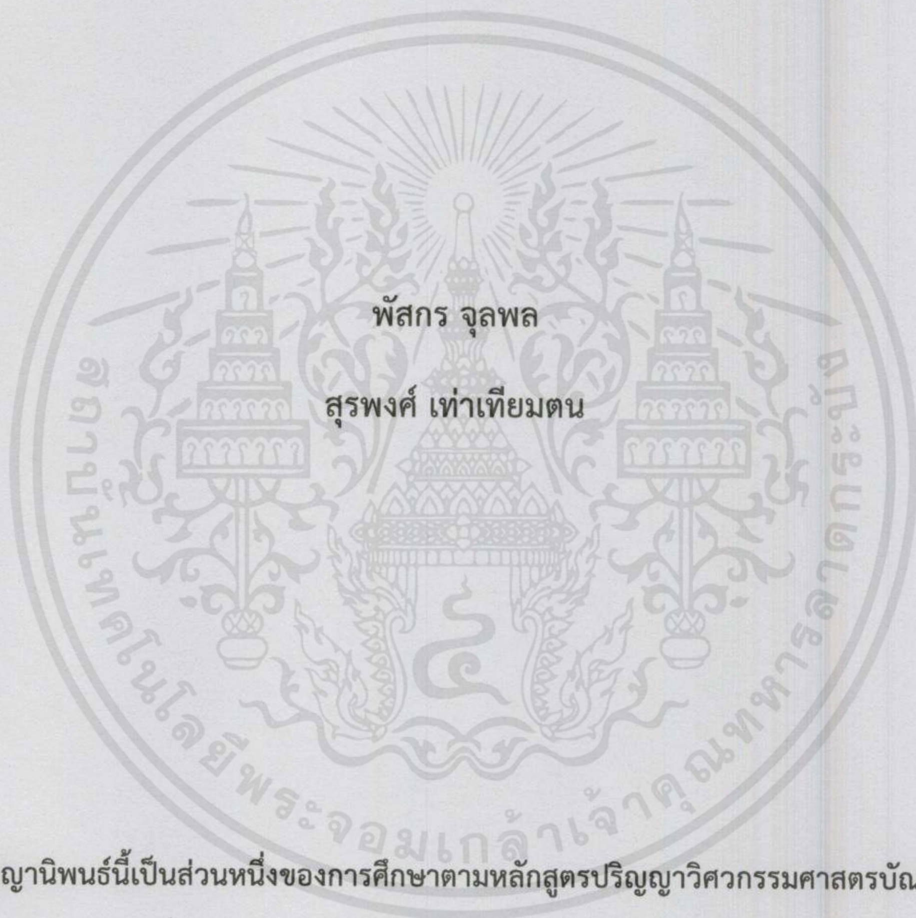
ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2557

การพัฒนาควิกซอร์ตแบบขนานด้วยโอเพนเอ็มพี

PARALLEL QUICKSORT DEVELOPMENT USING OPENMP



พัสกร จุลพล

สุรพงศ์ เท่าเทียมตน

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2557

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ปริญญานิพนธ์ปีการศึกษา 2557

ภาควิชาวิศวกรรมคอมพิวเตอร์

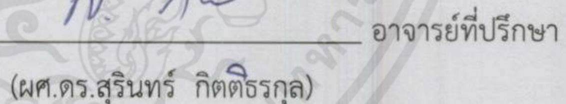
คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การพัฒนาควิกซอร์ตแบบขนานด้วยโอเพนเอ็มพี

PARALLEL QUICKSORT DEVELOPMENT USING OPENMP

ผู้จัดทำ

1. นายพัสกร จุลพล รหัสนักศึกษา 54010907
2. นายสรุพงศ์ เท่าเทียมตน รหัสนักศึกษา 54011423



อาจารย์ที่ปรึกษา

(ผศ.ดร.สุรินทร์ กิตติธรกุล)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

การพัฒนาควิกซอร์ตแบบขนานด้วยโอเพนเอ็มพี

นายพัสกร	จุลพล	54010907
นายสุรพงศ์	เท่าเทียมตน	54011423
ผศ.ดร.สุรินทร์	กิตติธรกุล	อาจารย์ที่ปรึกษา
ปีการศึกษา	2557	

บทคัดย่อ

การจัดเรียงเป็นปัญหาพื้นฐานของวิทยาการคอมพิวเตอร์ การเร่งประสิทธิภาพการจัดเรียงให้เร็วขึ้นด้วยฮาร์ดแวร์จำเป็นต้องอาศัยอัลกอริทึมที่มีประสิทธิภาพและง่ายต่อการพัฒนา เนื่องจากการวิเคราะห์ข้อมูลขนาดใหญ่มาก (Big Data) จำเป็นต้องอาศัยการจัดเรียงข้อมูลจำนวนมหาศาลบนเครื่องชนิด Multicore จำนวนมาก ดังนั้นหากเราสามารถพัฒนาอัลกอริทึมการจัดเรียงให้สามารถทำงานแบบขนานได้ก็จะสามารถเพิ่มประสิทธิภาพการทำงานของโปรแกรมคอมพิวเตอร์ให้สูงขึ้นได้ ผู้วิจัยได้เล็งเห็นความสำคัญนี้จึงคิดทำโครงงานการพัฒนาควิกซอร์ตแบบขนานด้วยโอเพนเอ็มพี เพื่อให้ได้อัลกอริทึมการจัดเรียงซึ่งสามารถทำงานแบบขนานได้และสามารถทำการจัดเรียงได้กับข้อมูลทุกชนิด

โดยโครงงานนี้ทำการพัฒนาอัลกอริทึมการจัดเรียงแบบขนานขึ้นมาใหม่โดยให้ชื่อว่า SPQsort ซึ่งผลการทดลองเป็นที่น่าพอใจเพราะสามารถทำงานเร็วขึ้น 2 เท่าเมื่อเทียบกับ Stdlib qsort() และผลการทดลองก็สอดคล้องกับค่า Complexity ของ BigO ที่ทำการวิเคราะห์

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

PARALLEL QUICKSORT DEVELOPMENT USING OPENMP

Mr. Passakorn	Jonlapon	54010907
Mr. Surapong	Toatiamton	54011423
Asst.Prof.Dr. Surin	Kittitornkun	Advisor
Academic year	2014	

ABSTRACT

Sorting is a basic problem in computer science. Increasing the efficiency of sorting by hardware require optimal algorithm and ease of algorithm development since Big data demands sorting a large amount of data on multicore computer. If we can develop parallel sorting algorithm then we can increase the efficiency of sorting. Hence we initiated this project "Parallel Quicksort Development using OpenMP" to be able to sort every data type. This project has proposed a new yet simple parallel sorting algorithm called "SPQsort". The result of project is satisfactory because SPQsort can achieve Speedup of 2 times faster than the Stdlib qsort. In addition, experiment results of experiment correspond with its BigO complexity

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

กิตติกรรมประกาศ

ปริญญานิพนธ์ฉบับนี้สำเร็จลุล่วงได้ด้วยดีด้วยความช่วยเหลือจากหลายฝ่ายทั้งในทางตรงและทางอ้อม โครงงานฉบับนี้จะสำเร็จลงไม่ได้หากปราศจากความช่วยเหลือของบุคคลเหล่านี้

อาจารย์ที่ปรึกษาคือ ผศ.ดร.สุรินทร์ กิตติธรกุล เป็นผู้ที่ให้คำแนะนำ ปรึกษา และให้ความช่วยเหลือตลอดการทำโครงงาน ซึ่งทำให้การทำงานต่าง ๆ เป็นไปได้อย่างราบรื่นและทำให้โครงงานฉบับนี้สำเร็จไปได้ด้วยดี

อาจารย์และบุคลากรต่าง ๆ ในสาขาวิชาวิศวกรรมคอมพิวเตอร์ที่ได้ให้คำแนะนำและสั่งสอนความรู้ต่าง ๆ มาตลอด รวมถึงอำนวยความสะดวกด้านสถานที่ในการทำวิจัยและพัฒนาโครงงาน

ในท้ายที่สุดนี้ขอขอบคุณบิดา มารดา และครอบครัวที่ได้เลี้ยงดูและสั่งสอน พร้อมทั้งให้โอกาสในการศึกษา และให้กำลังใจเสมอมา

นายพัสกร จุลพล 54010907

นายสุรพงศ์ เท่าเทียมตน 54011423

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

สารบัญ

หน้า

บทคัดย่อภาษาไทย I

บทคัดย่อภาษาอังกฤษ II

กิตติกรรมประกาศ III

สารบัญ IV

สารบัญตาราง VI

สารบัญรูป VII

บทที่ 1 บทนำ 1

1.1 ความสำคัญและที่มาของโครงงาน 1

1.2 วัตถุประสงค์ของโครงงาน 1

1.3 ขอบเขตของโครงงาน 1

1.4 วิธีการดำเนินการ 2

1.5 ประโยชน์ที่คาดว่าจะได้รับ 2

1.6 ส่วนประกอบของปริญญานิพนธ์ 2

บทที่ 2 เอกสารที่เกี่ยวข้อง 4

2.1 Quick sort 4

2.2 Stdlib qsort [1] 6

2.3 โอเพนเอ็มพี (OpenMP) [2] 8

2.4 งานที่เกี่ยวข้อง 10

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

สารบัญ(ต่อ)

หน้า

บทที่ 3 วิธีการดำเนินงาน..14
3.1 แนวคิดพื้นฐานของการจัดเรียงแบบ 2 เทรด.. 14
3.2 การคำนวณหาค่า BigO จากแนวคิดพื้นฐานของการจัดเรียงแบบ 2 เทรด............................ 16
3.3 การทำงานแบบขนานที่ 2^m เทรด.. 17
3.4 การคำนวณหาค่า BigO ของ 2^m เทรด 20

บทที่ 4 การทดลองและผลการทดลอง...23
4.1 การทดลอง 23
4.2 การวัดประสิทธิภาพ... 24
4.3 ผลการทดลอง... 25

บทที่ 5 บทสรุปและข้อเสนอแนะ 32
5.1 บทสรุป... 32
5.2 ปัญหาและอุปสรรค.. 32
5.3 แนวทางการแก้ไข 32
5.4 แนวทางในการพัฒนาต่อ.. 33

บรรณานุกรม 34

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

สารบัญตาราง

หน้า

ตารางที่ 2.1 การเปรียบเทียบอัลกอริทึมของการเรียงลำดับแบบต่างๆ 6

ตารางที่ 2.2 แสดงค่าของ BigO ของอัลกอริทึมของ Duhu Man และคณะ [3].................... 12

ตารางที่ 2.3 Quick sort ข้อมูลชนิด Integer จำนวน 10 ล้านตัว โดยใช้ 8 thread บน 8 เครื่อง (เครื่องละ thread).................... 13

ตารางที่ 3.1 การเปรียบเทียบค่า BigO ของแต่ละขั้นตอน ในกรณีแบ่งข้อมูลเป็น $h=2^m$.................... 20

ตารางที่ 4.1 แสดงพารามิเตอร์ต่าง ๆ ที่ใช้ในการทดลอง 23

ตารางที่ 4.2 แสดงการเปรียบเทียบคุณลักษณะเฉพาะของ CPU เหล่านี้.................... 23

ตารางที่ 4.3 เปรียบเทียบผลการทดลองของแต่ละชนิดข้อมูลที่ optimization ต่าง ๆ บนเครื่องทดลอง 25

ตารางที่ 4.4 ตารางแสดง BigO ที่จำนวนเทรด (Thread) ต่างๆ.................... 27

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

สารบัญรูป

หน้า

รูปที่ 2.1 แผนภาพความคิดพื้นฐานของ Quicksort 4
รูปที่ 2.2 ภาพแสดงโครงสร้าง (struct) ของ stack 7
รูปที่ 2.3 code ที่ประกาศกำหนดคุณสมบัติของ stack 7
รูปที่ 2.4 ภาพแสดงการทำงานของ thread แบบ fork join 9
รูปที่ 2.5 ภาพแสดงการทำงานของ thread แบบ fork join โดยใช้ omp parallel 9
รูปที่ 2.6 ภาพแสดงการทำงานของ thread แบบ fork join ในหลายพื้นที่และแบบซ้อนกัน (Nested) 10
รูปที่ 2.7 การทำงานของอัลกอริทึมของ Duhu Man และคณะ [3] 12
รูปที่ 3.1 ภาพตัวอย่างการทำการ SPQsort โดยการแบ่งจุดสลับ (swap) ตรงกับกรณีที่ 3.1.3c 16
รูปที่ 3.2 ภาพตัวอย่างการทำงาน SPQsort ในขั้นตอนที่ 3.3.1 - 3.3.7 เมื่อ h=4 19
รูปที่ 4.1 แสดงความสัมพันธ์ระหว่างเวลา (วินาที) กับจำนวนเทรด (Thread) ที่มีขนาดข้อมูลต่างกันของ uint32 กรณีแบบสุ่ม ใช้ออปติไมซ์ O2 บนเครื่อง AMD A6-3600 26
รูปที่ 4.2 แสดงความสัมพันธ์ระหว่าง Speedup กับจำนวนเทรด (Threads) ที่มีขนาดข้อมูลต่างกันของ uint32 กรณีแบบ Random ใช้ออปติไมซ์ O2 บนเครื่อง AMD A6-3600 29
รูปที่ 4.3 แสดงความสัมพันธ์ระหว่างเวลา กับ Cache Misses กับ Branch Load Misses ของข้อมูลชนิด unsignd integer 32 bit แบบสุ่มใช้การออปติไมซ์แบบ O2 จำนวน 4 เทรด (Thread) บนเครื่อง AMD A6-3600 30

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

บทที่ 1
บทนำ

1.1 ความสำคัญและที่มาของโครงงาน

การจัดเรียงเป็นปัญหาพื้นฐานของวิทยาการคอมพิวเตอร์ การเร่งประสิทธิภาพการจัดเรียงให้เร็วขึ้นด้วยฮาร์ดแวร์จำเป็นต้องอาศัยอัลกอริทึมที่มีประสิทธิภาพและง่ายต่อการพัฒนา เนื่องจากการวิเคราะห์ข้อมูลขนาดใหญ่มาก (Big Data) จำเป็นต้องอาศัยการจัดเรียงข้อมูลจำนวนมหาศาลบนเครื่องชนิด Multicore จำนวนมาก ดังนั้นหากเราสามารถพัฒนาอัลกอริทึมการจัดเรียงให้สามารถทำงานแบบขนานได้ก็จะสามารถเพิ่มประสิทธิภาพการทำงานของโปรแกรมคอมพิวเตอร์ให้สูงขึ้นได้ ผู้วิจัยได้เล็งเห็นความสำคัญนี้จึงจัดทำโครงงานการพัฒนา Quicksort แบบขนานด้วย OpenMP เพื่อให้ได้อัลกอริทึมการจัดเรียงซึ่งสามารถทำงานแบบขนานได้และสามารถทำการจัดเรียงได้กับข้อมูลทุกชนิด

1.2 วัตถุประสงค์ของโครงงาน

1. เพื่อศึกษาหลักการและรูปแบบการทำงานของโปรแกรมแบบขนาน
2. เพื่อศึกษาการพัฒนาโปรแกรมแบบขนานโดยใช้ OpenMP
3. เพื่อพัฒนาอัลกอริทึมจัดเรียงให้สามารถทำงานแบบขนานได้โดยเราตั้งชื่อว่า SPQsort
4. เพื่อศึกษาปัจจัยที่มีผลต่อประสิทธิภาพการทำงานของ SPQsort

1.3 ขอบเขตของโครงงาน

การพัฒนา Quicksort แบบขนานด้วยโอเพนเอ็มพีนี้จะทำการพัฒนาอัลกอริทึมเพื่อเรียกใช้ qsort ให้สามารถทำงานแบบขนานได้โดยใช้ OpenMP ซึ่งเป็น API ในการเขียนโปรแกรมแบบขนานซึ่งรองรับเฉพาะภาษา C,C++,FORTAN อีกทั้งยังใช้รูปแบบการใช้ข้อมูลแบบ Shared memory โดยในโครงงานนี้เราเลือกใช้ภาษา C เป็นภาษาในการพัฒนา โดยการพัฒนาจะพัฒนาการเขียนโปรแกรมบนระบบปฏิบัติการ Linux Ubuntu 14.04 และใช้ GCC เป็น compiler โดยจะพัฒนาให้อัลกอริทึมทำงานบนคอมพิวเตอร์เครื่องเดียวเพียงเท่านั้นไม่รองรับการทำงานข้ามเครื่องคอมพิวเตอร์หรือใช้คอมพิวเตอร์หลายเครื่องทำงานร่วมกัน

1.4 วิธีการดำเนินการ

การดำเนินงานเริ่มต้นด้วยขั้นตอนการศึกษาโดยเริ่มต้นจากการศึกษาอัลกอริทึม qsort ว่ามีขั้นตอนการทำงานอย่างไร ทำงานต่างจากอัลกอริทึมอื่นอย่างไรบ้าง จากนั้นผู้พัฒนาทำการศึกษาเกี่ยวกับการเขียนโปรแกรมแบบขนานในด้านต่างๆ เช่น การออกแบบ ปัญหาเมื่อทำการเขียนโปรแกรมแบบขนาน เป็นต้น จากนั้นทำการศึกษาเกี่ยวกับ OpenMP ว่ามีการทำงานอย่างไร รูปแบบของ API เป็นอย่างไร และทดลองทำการเขียนโปรแกรมแบบขนานเบื้องต้นด้วย OpenMP หลังจากนั้นการดำเนินงานเข้าสู่ขั้นตอนการพัฒนาโดยเริ่มจากการคิดวิธีที่จะทำให้ qsort สามารถทำงานแบบขนานได้ ซึ่งเมื่อได้วิธีการแล้วก็นำไปเขียนเป็นโปรแกรมโดยใช้ OpenMP

ในขั้นการทดลองจะนำโปรแกรมไปทดลองด้วย Parameter ที่กำหนดและทำการเก็บค่าผลการทดลอง จากนั้นจะนำค่าผลการทดลองมาวิเคราะห์ว่าอัลกอริทึมที่คิดค้นขึ้นนั้นมีประสิทธิภาพมากขึ้นเท่าไหร่ Parameter ใดมีผลกระทบต่อการอัลกอริทึมมากที่สุด

โดยการวิเคราะห์ประสิทธิภาพของอัลกอริทึมนั้นจะใช้การคำนวณค่า BigO ของอัลกอริทึมโดยทำการหาค่า BigO ของอัลกอริทึมทุกขั้นตอนมารวมกันให้ได้เป็น BigO อย่างละเอียดและนำมาสรุปเป็น BigO ของอัลกอริทึม แต่การวิเคราะห์ BigO นั้นเป็นเพียงทฤษฎีจึงต้องมีการทดลองและเก็บค่าผลการทดลองมาสร้างกราฟเพื่อพิสูจน์ว่าผลการทดลองเป็นไปตามการวิเคราะห์ด้วย BigO หรือไม่

1.5 ประโยชน์ที่คาดว่าจะได้รับ

1. อัลกอริทึม SPQsort จะใช้ให้สามารถจัดเรียงข้อมูลได้เร็วกว่าการจัดเรียงมาตรฐาน
2. อัลกอริทึมนี้สามารถจะใช้งาน CPU ได้อย่างเต็มที่และมีประสิทธิภาพเพิ่มขึ้น
3. อัลกอริทึมนี้สามารถที่จะจัดเรียงข้อมูลได้หลายชนิดและไม่จำกัดขนาดของข้อมูล
4. อัลกอริทึมนี้สามารถนำไปใช้กับงานที่ต้องการความเร็วในการจัดเรียงหรือค้นหาข้อมูล

1.6 ส่วนประกอบของปริญญานิพนธ์

ปริญญานิพนธ์ฉบับนี้ได้แบ่งเนื้อหาออกเป็น 5 บทด้วยกัน คือ

บทที่ 1 บทนำ กล่าวถึงความสำคัญและที่มาโครงงาน วัตถุประสงค์ของโครงงาน ขอบเขตของโครงงาน วิธีการดำเนินการ ประโยชน์ที่คาดว่าจะได้รับ และส่วนประกอบของปริญญานิพนธ์

บทที่ 2 ทฤษฎีที่เกี่ยวข้อง ในบทนี้จะกล่าวถึงทฤษฎีพื้นฐานประกอบด้วย Quicksort เป็นทฤษฎีที่เป็นหัวใจหลักของปริญญานิพนธ์ฉบับนี้

บทที่ 3 การออกแบบและพัฒนา กล่าวถึงการออกแบบอัลกอริทึมและการทำงานของอัลกอริทึม SPQsort คำนวณค่าของ BigO ของอัลกอริทึมและ Quicksort ปกติ

บทที่ 4 การทดลองและผลการทดลอง กล่าวถึงพารามิเตอร์ต่าง ๆ ที่ใช้ในการทดลองอัลกอริทึม การวัดประสิทธิภาพของอัลกอริทึมและผลการทดลองของอัลกอริทึม

บทที่ 5 บทสรุป เป็นการสรุปผลการทดลองของปริญญานิพนธ์ จะกล่าวถึงการสรุปผลการทดลองอัลกอริทึม และประโยชน์ที่ได้รับ

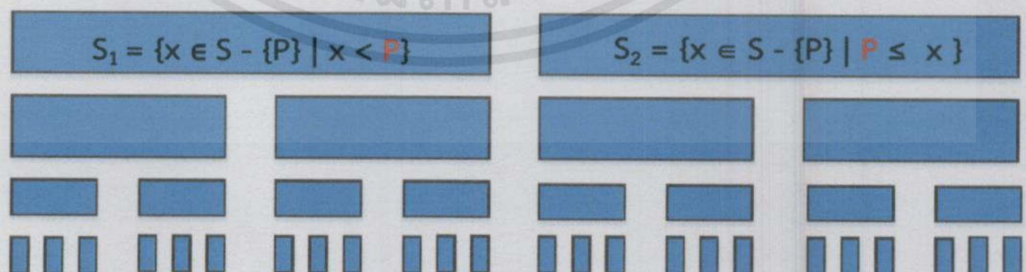
เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

บทที่ 2
เอกสารที่เกี่ยวข้อง

ในการจัดทำโครงงานนี้ เราจำเป็นต้องศึกษาข้อมูลที่ใช้ประกอบการทำโครงงาน เพื่อให้เข้าใจในหลักการทำงานของงานในแต่ละส่วน ซึ่งประกอบด้วย Quicksort, Stdlib qsort, โอเพนเอ็มพี (OpenMP) และงานที่เกี่ยวข้องที่ใช้ในการอ้างอิงเพื่อเป็นแนวทางในการคิดอัลกอริทึมของโครงงานนี้

2.1 Quick sort

การเรียงลำดับข้อมูลนั้นมีหลายอัลกอริทึม (Algorithm) หนึ่งในนั้นคือ การเรียงลำดับแบบเร็ว (Quicksort) หรืออีกชื่อหนึ่ง Partition-Exchange sort ถูกพัฒนาโดย Tony Hoare ในปี ค.ศ.1960 เป็นอัลกอริทึมที่มีการเรียงลำดับค่าในอาร์เรย์ประเภทใด ๆ ซึ่งจะดำเนินการภายในพื้นที่ (In-Place) ของอาร์เรย์ (Array) ทำให้ต้องการหน่วยความจำขนาดเล็กในการเรียงลำดับ โดยอัลกอริทึมนี้การเรียงลำดับจะให้ประสิทธิภาพโดยเฉลี่ยเป็น O (n log n) ในการเรียงข้อมูลจำนวน n ตัว ส่วนในกรณีที่เลวร้ายที่สุด (Worst case) ของการจัดเรียงให้ประสิทธิภาพได้เป็น O (n^2) ซึ่งกรณีนี้จะเกิดขึ้นได้ยาก เมื่อทำการเปรียบเทียบกันแล้วการเรียงลำดับ (Quicksort) ในทางปฏิบัติมักจะเร็วกว่าอัลกอริทึมตัวอื่นที่มี BigO ในการเรียงลำดับที่ให้ประสิทธิภาพเป็น O (n log n) เหมือนกัน ยกตัวอย่างเช่น Marge sort และ Heap sort นอกจากนี้มันจะทำงานเป็นลำดับและใช้หน่วยความจำของตนเอง (Space) ซึ่งการดำเนินการจะเป็นรีเคอร์ซีฟ (Recursion) โดยใช้หลักการ LIFO เข้ามาสนับสนุน



รูปที่ 2.1 แผนภาพความคิดพื้นฐานของ Quicksort

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

อัลกอริทึมของการจัดเรียงลำดับแบบเร็ว (Quicksort) พื้นฐาน โดยสมมติว่าชุดข้อมูลอาร์เรย์ (Array) หนึ่งชุดที่ต้องการเรียงลำดับ ซึ่งมีขั้นตอนการดำเนินการดังนี้

1. ทำการเลือกค่าไพวอท (Pivot) จากค่าของข้อมูลในอาร์เรย์ (Array)

2. ทำการแบ่งออกเป็น 2 กลุ่มย่อย ตามรูปที่ 2.1

    2.1 กลุ่มของข้อมูลที่มีค่าน้อยกว่าเท่ากับ (<) ค่าของไพวอท เรียกว่า กลุ่มซ้ายมือ

    2.2 กลุ่มของข้อมูลที่มีค่ามากกว่า (≥) ค่าไพวอท เรียกว่า กลุ่มขวามือ

3. ทำการเวียนเกิด (Recursive) ในกลุ่มย่อยแต่ละกลุ่มซ้ายมือและกลุ่มขวามือ โดยดำเนินการตามขั้นตอนที่ 1

ปัจจัยที่มีผลกับความเร็วของการเรียงลำดับแบบเร็ว (Quicksort) อย่างหนึ่งคือการเลือกไพวอท โดยการเลือกไพวอทนั้นมีได้หลายวิธีซึ่งในที่นี้จะอธิบาย 3 วิธีคือ

1. การเลือกตัวแรกหรือตัวสุดท้ายของอาร์เรย์ (Array) มาเป็นไพวอท ซึ่งในกรณีทั่วไปนั้นความเร็วจะยังเป็น O (n log n) แต่ถ้าในกรณีแย่ที่สุดเช่น ข้อมูลเรียงจากน้อยไปหามากหรือมากไปหาน้อยความซับซ้อนในการเรียงลำดับ (sort) จะเพิ่มขึ้นอย่างเห็นได้ชัด จาก O (n log n) เป็น O (n^2)
2. การสุ่ม (Random) เลือกค่าใดค่าหนึ่งในอาร์เรย์ (Array) มาเป็นไพวอทวิธีนี้จะช่วยลดการเกิด Worst case ลงได้เพราะแม้ข้อมูลจะเรียงลำดับอยู่แล้ว แต่ตำแหน่งที่ถูกเลือกมาเป็นไพวอทนั้นไม่ตายตัวเหมือนเลือกตัวแรกหรือตัวสุดท้าย ทำให้โอกาสจะเกิด Worst case ยากขึ้น
3. การใช้วิธี median of three คือ เลือกค่าแรก ค่าตรงกลาง และค่าสุดท้ายของ Partition มาเปรียบเทียบกันโดยค่าใดเป็นค่าตรงกลางระหว่าง 3 ค่าที่เลือกมาจะถูกเลือกเป็นไพวอทวิธีนี้จะช่วยแก้ไขกรณี Worst case ได้ดีมากเพราะ Algorithm จะได้ค่าตรงกลางของ partition มากขึ้น ทำให้ Worst case ที่เกิดจากอาร์เรย์ (Array) ที่เรียงลำดับจากน้อยไปมากหรือมากไปน้อยใช้เวลาเพียง O (n log n)

ในทางปฏิบัติแล้ว Quicksort จะมีความเร็วในการเรียงลำดับได้เร็วกว่า Heap sort และ Merge sort ถึงแม้ว่าอัลกอริทึมของการเรียงลำดับทั้งสามจะมีประสิทธิภาพเหมือนกัน คือ O (n log n) สามารถดูผลเปรียบเทียบในแต่ละกรณีได้จาก ตารางที่ 2.1

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ตารางที่ 2.1 การเปรียบเทียบอัลกอริทึมของการเรียงลำดับแบบต่างๆ

อัลกอริทึม/กรณี	Best case	Worst case	Average case
Heap sort	n log n	n log n	n log n
Merge sort	n log n	n log n	n log n
Quick sort	n log n	n^2	n log n

2.2 Stdlib qsort [1]

qsort เป็นฟังก์ชันที่ทำการเรียงลำดับ (sort) ค่าต่างๆ ได้หลายชนิด ได้แก่ จำนวนเต็ม จำนวนทศนิยม ตัวอักษร เป็นต้น โดยฟังก์ชัน qsort นี้จะอยู่ใน library ชื่อ stdlib.h ของภาษา C ตามคอมไพเลอร์ GCC โดยฟังก์ชัน qsort จะรับชุดข้อมูลที่จะนำมาเรียงลำดับ (sort) ในรูปแบบอาเรย์ (Array)

*quicksort (void *const pbase, size_t total_elems, size_t size,*
*int (*cmp)(const void *left, const void *right)*

Parameter ของฟังก์ชัน ประกอบด้วย

*void *constp base* เป็นตัวแปร pointer ตำแหน่งเริ่มต้นของชุดข้อมูลที่จะทำการ sort

size_t total_elems เป็นตัวแปรบอกขนาดของอาเรย์ Array ว่ามีจำนวนเท่าไหร่

size_t size เป็นตัวแปรบอกขนาดของ Type นั้นมีมีขนาดกี่ไบท์

*int (*cmp)(const void *left, const void *right)*

เป็น function pointer ที่กำหนดรูปแบบการเปรียบเทียบ type นั้นว่าจะเปรียบเทียบกันอย่างไรโดยค่าที่ return จากฟังก์ชันนี้จะเป็นผลลัพธ์จากการเปรียบเทียบ โดย จะ return = 0 เมื่อ left = right, return > 0 เมื่อ left > right และ return < 0 เมื่อ left < right

โดยฟังก์ชั่น qsort ของ stdlib.h นั้นใช้ Algorithm การ sort แบบ Quicksort โดยมีเทคนิคการเพิ่มความเร็วในการ sort คือ

1. ไม่ใช้การ Recursion ในการทำแบ่ง partition และ sort แต่จะใช้การสร้าง stack ขึ้นมาเก็บขอบเขต partition ไว้และใช้ while loop ทำการวน pop ค่าขอบเขต

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ออกมาจาก stack มาทำการแบ่ง partition ต่อไป และ push ลง stack ทำไปเรื่อย ๆ จนกว่าข้อมูลจะถูก sort เรียบร้อย

2. ใช้การเลือกค่าไพวอท โดยใช้การหาค่า median จากข้อมูล 3 ค่า แทนการเลือกค่าแรกสุด ค่าท้ายสุด หรือค่า Random เพื่อหลีกเลี่ยงการได้ค่าไพวอทที่ไม่ดี
3. จะทำการแบ่ง partition จนจำนวนสมาชิกใน partition ไปเรื่อย ๆ จนเหลือจำนวนน้อยกว่าเท่ากับจำนวนที่ตั้งไว้ (Threshold) หลังจากนั้นจะทำการใช้ Insertion sort ทำงานต่อ

จากการศึกษาโอเพนซอร์สโค๊ดของ qsort ใน stdlib.h ทำให้ทราบว่า qsort ใน stdlib.h นั้นมีความยืดหยุ่นสามารถเปรียบเทียบค่าได้ทุกชนิดข้อมูล (type) และการเปรียบเทียบนั้นใช้ cmp function ส่งเข้าไปในตัวแปรไปยัง int (*cmp)(const void *left, const void *right) ซึ่งทำให้การ sort เป็นไปตามความต้องการของผู้ใช้และไม่ยึดติดกับ type ของการ sort ชนิดอื่นๆ เช่น การ sort ด้วย radix sort ที่ต้อง sort โดยการใช้ค่าประจำหลัก ซึ่งจะใช้ได้กับ type ที่เป็นตัวเลขเท่านั้น จากเทคนิคของ qsort ใน stdlib.h นั้นทำให้ทราบว่าการแบ่ง partition เรื่อยๆจนทุกค่าในชุดข้อมูลถูกเรียงตาม Algorithm นั้นช้ากว่าการใช้ Insertion sort เมื่อจำนวนสมาชิกใน partition ถึงค่า Threshold

```
56 /* Stack node declarations used to store
57 unfulfilled partition obligations. */
58 typedef struct
59   {
60     char *lo;
61     char *hi;
62   } stack_node;
63
```

รูปที่ 2.2 ภาพแสดงโครงสร้าง (struct) ของ stack

```
69 #define STACK_SIZE  (CHAR_BIT * sizeof(size_t))
70 #define PUSH(low, high) ((void) ((top->lo = (low)), (top->hi = (high)), ++top))
71 #define POP(low, high)  ((void) (--top, (low = top->lo), (high = top->hi)))
72 #define STACK_NOT_EMPTY (stack < top)
```

รูปที่ 2.3 code ที่ประกาศกำหนดคุณสมบัติของ stack

ฟังก์ชั่น qsort ใช้โครงสร้างข้อมูลชนิด stack เก็บขอบเขตของ partition นั้น ในตัวแปรโครงสร้าง (struct) โดยแอตทิบิวต์ภายในโครงสร้างประกอบด้วยตัวแปร char* lo และ char* hi ที่เก็บค่าต่ำสุดและค่าสูงสุดของอาร์เรย์ของข้อมูล ดังรูปที่ 2.2 นอกจากนี้ใน qsort ได้มีการกำหนดขนาดของ

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

stack การ push-pop ค่าของ stack และกำหนดค่ากรณีที่ stack ไม่ว่างไว้เพื่อนำไปช่วยในการเรียงลำดับของฟังก์ชั่น qsort ดังรูปที่ 2.3

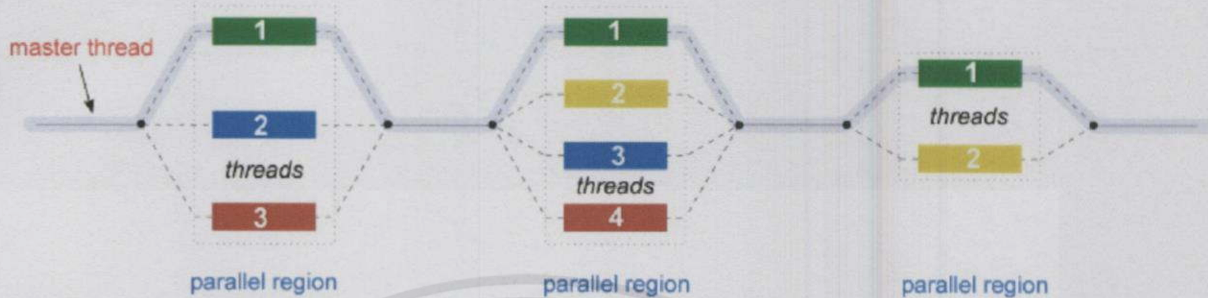
2.3 โอเพนเอ็มพี (OpenMP) [2]

โอเพนเอ็มพี (OpenMP) เรียกย่อๆ ว่า โอเอ็มพี (OMP) เป็นแอพพลิเคชันโปรแกรมอินเทอร์เฟส หรือเอพีไอ (Application Program Interface : API) ที่ใช้ในการพัฒนาโปรแกรมที่มีการคำนวณแบบขนานโดยการปรับเธรด (Thread-based parallel programming) บนมัลติโปรเซสเซอร์แบบแชร์หน่วยความจำ (Shared memory multiprocessors) ซึ่ง เอพีไอ นี้มีการใช้อย่างแพร่หลายจนกลายเป็นมาตรฐานวิธีพฤตินัย (de-facto Standard) ในการพัฒนาโปรแกรมแบบขนาน ทำให้คอมพิวเตอร์สามารถเข้าถึงหน่วยความจำโดยตรงผ่านรูปแบบการเชื่อมต่อแบบโลจิคอล (Logical) ที่เรียกว่า ระบบบัสเบส (Bus-based System)

ในการพัฒนาโปรแกรมนั้นมีหลายโปรแกรมที่สามารถทำงานได้ดีกับการคำนวณแบบขนานมากกว่าการคำนวณแบบอนุกรม ซึ่งส่วนใหญ่แล้วโปรแกรมดังกล่าวจะใช้เวลาในการทำงานยาวนาน เช่น ซอฟต์แวร์ทางไบโออินฟอร์เมติกส์ ซอฟต์แวร์ทำนายสภาพภูมิอากาศ ซอฟต์แวร์ในการประมวลผลภาพ หลายโปรแกรมที่กล่าวมาข้างต้นนั้นความแม่นยำ และความถูกต้องจะแปรผันตามเวลาที่คำนวณ สิ่งนี้ทำให้การคำนวณแบบขนานนั้นสามารถลดเวลาในการคำนวณให้น้อยลง และเป็นสิ่งที่ทำให้ประสิทธิภาพในการทำงานมากยิ่งขึ้นหนึ่งในวิธีการที่จะทำให้โปรแกรมที่พัฒนามีการคำนวณแบบขนาน โดยการนำเอาโอเพนเอ็มพี มาพัฒนาร่วมด้วย

โอเพนเอ็มพี ได้ถูกออกแบบเพื่อให้โปรแกรมเมอร์สามารถพัฒนาโปรแกรมแบบขนาน (parallel programming) สำหรับภาษา C, C++, Fortran ซึ่งรองรับระบบปฏิบัติการ Unix, Linux และ Windows การทำงานของเธรด (thread) ใน omp นั้นจะเป็นแบบฟอร์ก-จอย (fork join) กล่าวคือในช่วงเวลาการทำงาน (runtime) นั้น เมื่ออยู่ในส่วนที่ไม่ได้ทำงานแบบขนานนั้นตัวโพรเซส (process) จะมีมาสเตอร์เธรด (master thread) ทำงานเพียงเธรดเดียวในตอนเริ่มต้น เมื่อมาถึงส่วนที่ต้องการทำงานแบบขนานตัว master Thread จะทำการสร้างเธรด (Fork) ขึ้นมาทำงานตามชุดคำสั่งที่อยู่ในส่วนขนาน (Parallel Region) นั้น และเมื่อทำงานในส่วนที่เป็นขนานเสร็จ โพรเซสจะยกเลิกเธรดที่ถูกฟอร์กขึ้นมาให้เหลือเพียงมาสเตอร์เธรด (master thread) เดียวเหมือนเดิม ดังรูปที่ 2.4

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้



รูปที่ 2.4 ภาพแสดงการทำงานของ thread แบบ fork join

การกำหนดพื้นที่ในซอร์สโค้ดบริเวณใดๆ ที่ต้องการให้มีการทำงานแบบขนาน จะต้องใช้ construct เป็นตัวกำหนดตำแหน่งเริ่มต้นและสิ้นสุด ดังรูปที่ 2.5 โดยในตัวอย่างจะใช้ construct ของ OpenMP ที่ชื่อว่า **omp parallel** โดยการทำงานของ omp parallel นั้นจะเป็นการสร้าง thread ขึ้นมาทำงานส่วนที่อยู่ด้านในส่วน **omp parallel** พร้อมกันและเมื่อทำงานในส่วน omp parallel เสร็จทุกเธรด (thread) จะกลับมารวมกันเหลือมาสเตอร์เธรดเดียวตามรูปที่ 2-5 และเมื่อตัวโปรแกรม run ถึงบรรทัด #pragma omp parallel ตัว OpenMP จะทำการสร้างเธรด (thread) ขึ้นมาตามจำนวนที่กำหนด จากนั้นแต่ละเธรด (thread) จะเข้าไปทำงานในส่วน omp parallel ซึ่งในที่นี้คือ printf("In omp parallel\n"); และเมื่อทุกเธรด (thread) ทำงานเสร็จก็จะกลับมาเหลือเธรด (thread) เดียวและทำบรรทัด printf("Out omp parallel"); ซึ่งในทีนี้ถ้าโปรแกรมเมอร์กำหนดค่าในคอมไพเลอร์ (Complier) ให้สร้างเธรด (thread) มา 4 เธรด บนหน้าจอจะเห็น “In omp parallel” จำนวน 4 บรรทัดและ “Out omp parallel” จำนวน 1 บรรทัด

```
#pragma omp parallel
{
    printf("In omp parallel\n");
}
printf("Out omp parallel\n");
```

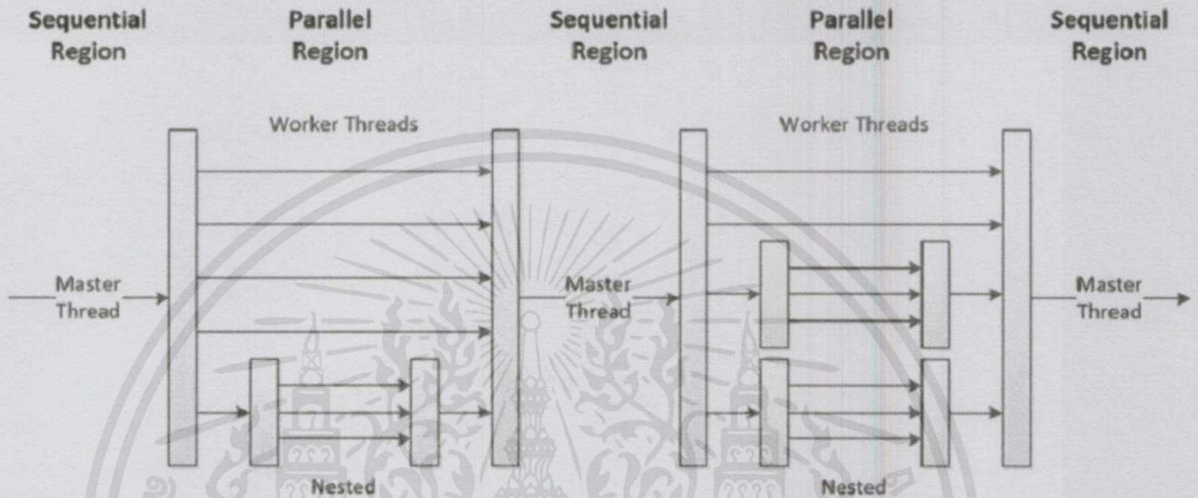
รูปที่ 2.5 ภาพแสดงการทำงานของ thread แบบ fork join โดยใช้ omp parallel

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

โครงสร้างการคำนวณแบบขนาน (Parallel Construct) มีสองรูปแบบ ได้แก่ โครงสร้างแบบขนานตามข้อมูล (Data Parallel Construct) และโครงสร้างแบบขนานตามงาน (Task Parallel

ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Construct) โดยการพัฒนาโปรแกรมด้วยโอเพนเอ็มพีนั้นสามารถสร้างบริเวณการคำนวณแบบขนานได้หลายบริเวณภายใน 1 โปรแกรม ซึ่งนักพัฒนาโปรแกรมสามารถเลือกบริเวณที่สามารถคำนวณแบบขนานและสามารถสร้างบริเวณการคำนวณแบบขนานซ้อนกัน (Nested) ได้ ดังรูปที่ 2-6



รูปที่ 2.6 ภาพแสดงการทำงานของ thread แบบ fork join ในหลายพื้นที่และแบบซ้อนกัน (Nested)

การใช้โอเพนเอ็มพีในการพัฒนาโปรแกรมแบบขนานมีข้อดีคือ เธรดโปรแกรมเมอร์สามารถทำแปลงบริเวณเล็ก ๆ ให้มีการทำงานแบบขนานได้ โปรแกรมเมอร์สามารถปรับจำนวนเธรดตามจำนวนโปรเซสเซอร์ได้ และสามารถพัฒนาอัลกอริทึมแบบขนานได้ง่าย โดยใช้คำสั่งไม่มากและทำการซิงค์ (Synchronization) ระหว่างเธรดได้ เพื่อรักษาความถูกต้องของงาน

2.4 งานที่เกี่ยวข้อง

จากการศึกษาข้อมูลในการทำโครงงานนี้ ผู้จัดทำได้มีการศึกษาผลงานที่เกี่ยวข้องหรือมีความคล้ายคลึงกับโครงงานที่ได้จัดทำขึ้น โดยงานที่ผู้จัดทำได้ศึกษามีจำนวน 2 บทความ ได้แก่ An Efficient Parallel Sorting Compatible With The Standard Qsort และ Adaptive Parallelism for OpenMP Task Parallel Programs

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2.4.1 An Efficient Parallel Sorting Compatible With The Standard Qsort

Duhu Man และคณะ [3] เป็นผลงานวิจัยเกี่ยวกับประสิทธิภาพของของการจัดเรียงแบบ psort ที่ใช้ stdlib qsort ช่วยในการทำการจัดเรียงโดยการจัดเรียงลำดับแบบขนานแบบชื่อว่า psort ซึ่งพัฒนาด้วยไลบรารี่มาตรฐานภาษา C อัลกอริทึมที่ใช้ในการจัดเรียงแบบขนานมีขั้นตอนดังนี้

กำหนดให้

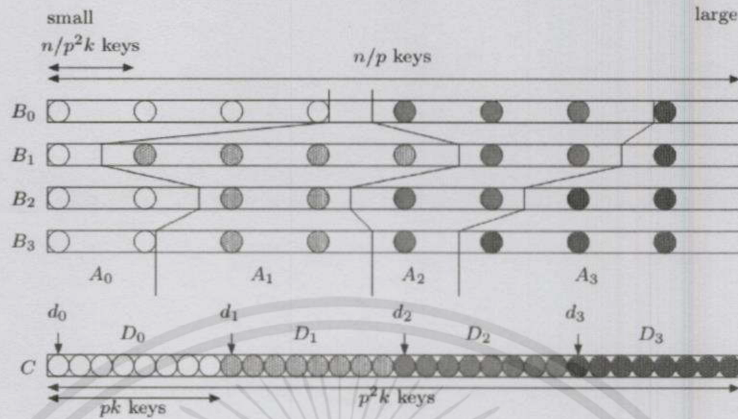
A ชุดข้อมูลที่จะนำมาทำการ sort

P จำนวน processor core

K parameter ที่กำหนดขึ้นถ้าไม่กำหนดคือ 1

1. กำหนดให้ $A = \{a_1, a_2, a_3, ..., a_{n-1}\}$ ทำการแบ่ง A ออกเป็น P กลุ่ม เรียกกลุ่มที่ถูกแบ่งออกมาว่า $B= \{B_0, B_1, ..., B_{p-1}\}$ ตัวอย่างเช่น P เท่ากับ 2 เราจะแบ่งออกมาได้เป็นสองกลุ่ม B_0, B_1
2. ทำการ sort B แต่ละกลุ่มพร้อมกันด้วย std libqsort
3. กำหนดให้ $numB_n$ คือ จำนวนสมาชิกในกลุ่ม B_n เช่น $numB_1$ คือ จำนวนสมาชิกของกลุ่ม B_1 ทำการคัดลอกสมาชิก index ที่ mod k เท่ากับ 0 ตัวอย่างเช่น numB =10 k = 2 เราจะ copy สมาชิกตัวที่ 0, 2, 4, 6, 8 ไปยังกลุ่ม C โดยทำทุกกลุ่ม B พร้อมกัน
4. กำหนดให้ numC คือ จำนวนสมาชิกในกลุ่ม C นำกลุ่ม C ที่ได้จากขั้นตอนที่ 3 มา sort ด้วย stdlib sort จากนั้นคัดลอกค่าของสมาชิกในกลุ่ม C index ที่ mod (numC/P) เท่ากับ 0 ไป กลุ่ม D เช่น ในกลุ่ม C มีสมาชิก 10 ตัว P=2 (numC/P) = 5 เราจะคัดลอกสมาชิกตัวที่ 0, 5
5. กลุ่ม D ประกอบไปด้วย $D = \{D_0, D_1, ..., D_{p-1}\}$ จากนั้นทำการคัดลอกค่าจากกลุ่ม B ทุกกลุ่มไปอยู่กลุ่ม E โดยมีเงื่อนไขคือค่าของสมาชิกในกลุ่ม E_n ต้องมีค่าน้อยกว่า D_{n+1} และ มากกว่าเท่ากับ D_n โดย D_pมีค่า + infinity
6. ทำการ sort แต่ละกลุ่ม D พร้อมกัน
7. ทำการคัดลอกค่าจากกลุ่ม D ไปแทนที่กลุ่ม A

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้



รูปที่ 2.7 การทำงานของอัลกอริทึมของ Duhu Man และคณะ [3]

จากการศึกษาบทความนี้ ในการทดสอบอัลกอริทึมของบทความได้ใช้ระบบปฏิบัติการตระกูลลินิกซ์ (CentOS 5.1) ทำงานบนชิปประมวลผลของ Intel Xeon X5355 2.66GHz แบบสี่แกนสมอง (quad-core) จำนวนสองชิป (Intel Xeon X5355 2.66GHz) ซอร์สโค๊ดภาษา C คอมไพล์ด้วย gcc 4.1.2 ด้วยออปติไมเซชั่น (optimization) –O2 ซึ่งผลลัพธ์ของอัลกอริทึมนี้มีความเร็ว (Speed up) มากกว่าความเร็วของ qsort แบบดั้งเดิม 6 เท่า ซึ่งความเร็ว (Speedup) ที่ได้จากอัลกอริทึมนี้ไม่ได้มากกว่า 8 เท่า แม้จะใช้ถึง 8 แกนสมอง (core) ก็ตาม

นอกจากนี้ เราได้ทำการวิเคราะห์ Time Complexity ด้วย BigO ของอัลกอริทึมที่ใช้ในบทความนี้ในแต่ละขั้นตอน ซึ่งสรุปได้ดัง ตารางที่ 2.2

ตารางที่ 2.2 แสดงค่าของ BigO ของอัลกอริทึมของ Duhu Man และคณะ [3]

ขั้นตอนที่	BigO
1	$O(1)$
2	$O(\frac{n}{p}\log\left(\frac{n}{p}\right))$
3	$O(\frac{n}{pk})$
4	$O(\frac{n}{k}\log\left(\frac{n}{k}\right))$
5	$O(\frac{n}{p})$
6	$O(\frac{n}{p}\log\left(\frac{n}{p}\right))$
7	$O(\frac{n}{p})$

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงที่มาของเอกสารทุกครั้งที่มีการนำไปใช้

2.4.2 Adaptive Parallelism for OpenMP Task Parallel Programs [4]

เป็นงานวิจัยที่ใช้ OpenMP บน network of workstations (NOW) โดยมีเหตุผลมาจาก Network Of Workstations (NOW) นั้นส่วนใหญ่จะทำการเปิดไว้ตลอด เพื่อให้คนมาใช้งานแต่เครื่องนั้นก็ไม่ได้ถูกใช้อยู่ตลอดเวลา ดังนั้นหากสามารถให้เครื่องที่ยังไม่มีผู้ใช้งานมาช่วยประมวลผลงานได้ จะทำให้สามารถใช้ทรัพยากรเครื่องได้อย่างเต็มที่ โดยงานวิจัยนี้จะทำการสร้างระบบที่เมื่อใช้ OpenMP ในการสร้างเธรด (Thread) ระบบจะทำการเปลี่ยนการสร้างเธรด (Thread) ไปเป็นการสั่งเครื่องที่ไม่ได้ถูกใช้งานหรือเครื่องที่ว่างอยู่ใน NOW ให้ทำการสร้าง process ขึ้นมาและทำงานที่มอบหมายไปให้ โดยในงานวิจัยนี้จะทำการทดลองประสิทธิภาพของระบบด้วยงาน 2 ชนิด คือ 1. Quick sort 2. Travelling salesman problem โดยในที่นี้จะกล่าวถึงเฉพาะ Quick sort

โดยการจ่ายงานของ Quick sort จะทำการแบ่งอาร์เรย์ (array) ที่ยังไม่ได้เรียงลำดับ (sort) ออกเป็น 2 subarray ตามไพวอทที่เลือกจากนั้นเก็บ subarray 1 ส่วนไว้ที่ตัวเอง อีกส่วนที่เหลือส่งไปให้เครื่องว่างงานทำงานต่อให้โดยทำอย่างนี้ไปเรื่อย ๆ จน subarray ที่เหลืออยู่กับตัวเองมีขนาดถึงระดับที่กำหนดจึงมาทำการเรียงลำดับ (sort) แบบไม่แจกงาน ในเครื่องที่ได้รับงานก็ทำอย่างนี้เช่นกันเมื่อทำงานเสร็จก็ส่งผลลัพธ์กลับไปที่เครื่องที่จ่ายงานมาให้เครื่องตัวเอง

ในการทดลองใช้ Pentium II ขนาดหน่วยความจำ 256MB จำนวน 8 เครื่อง บนระบบปฏิบัติการลินิกซ์ (Linux 2.2.7) เชื่อมต่อกันด้วย UDP socket แบบ full-duplex บน Ethernet networks ด้วยความเร็ว 100Mbps และ 1Gbps

ตารางที่ 2.3 Quick sort ข้อมูลชนิด Integer จำนวน 10 ล้านตัว โดยใช้ 8 thread บน 8 เครื่อง (เครื่องละ thread)

ความเร็ว	Avg time(วินาที)
100 Mbps	10.86
1 Gbps	6.46

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

บทที่ 3

วิธีการดำเนินงาน

ในบทนี้จะทำการอธิบายเกี่ยวกับ Parallel Quicksort ที่ทำการคิดขึ้นมาใหม่โดยเราตั้งชื่อว่า SPQsort โดยจะอธิบายแนวคิดพื้นฐานของ SPQsort ที่ใช้ 2 เทรด (Thread) ในการทำงานว่ามีขั้นตอนการทำงานอย่างไร ต่อไปจะแสดงถึงวิธีการคำนวณค่า BigO ของของการใช้ SPQsort แบบ 2 เทรด (Thread) จากนั้นจะทำการอธิบายการใช้ SPQsort แบบ 2^m เทรด (Thread) ว่าจะต้องนำ SPQsort แบบ 2 เทรด (Thread) มาทำการเพิ่มขั้นตอนอย่างไรและวิธีการคำนวณ BigO ของ SPQsort แบบ 2^m เทรด (Thread) ว่ามีรูปทั่วไปอย่างไร

3.1 แนวคิดพื้นฐานของการจัดเรียงแบบ 2 เทรด

แนวคิดของอัลกอริทึม SPQsort มีขั้นตอนดังนี้

กำหนดให้ A = อาเรย์ (Array) ข้อมูลที่ต้องการจัดเรียง (sort)

h = จำนวนเทรด (Thread)

อัลกอริทึม (Algorithm) สำหรับ h=2 มีขั้นตอนดังนี้

3.1.1 ทำการแบ่งชุดข้อมูลที่ได้รับมาขนาด N ตัวเป็น h ส่วนเท่า ๆ กัน ดังนั้นจะมีพาร์ติชัน (partition) ทั้งหมด h พาร์ติชัน (partition) ซึ่งประกอบไปด้วย $Partition_{(0)}$ และ $Partition_{(1)}$ จากนั้นหา index ที่เป็นจุดเริ่มต้นของแต่ละพาร์ติชัน (partition) เรียกว่า Start และจุดสุดท้ายของพาร์ติชัน (partition) เรียกว่า Stop โดยในขั้นตอนนี้จะมีเพียง 1 เทรด (Thread) ที่ทำงาน

3.1.2 ทำการจัดเรียง (sort) แต่ละพาร์ติชัน (partition) พร้อมกันโดยให้แต่ละเทรด (Thread) ใช้ฟังก์ชั่น qsort กับพาร์ติชัน (partition) ที่ตัวเองดูแล

3.1.3 ทำการหา index กึ่งกลางของแต่ละส่วนโดยจะเรียกจุดกึ่งกลางว่า mid โดยจะมี mid ทั้งหมด h mid ซึ่งประกอบไปด้วย $mid_{(0)}$ และ $mid_{(1)}$ จากนั้นทำการเปรียบเทียบค่า A [$mid_{(0)}$] กับ A [$mid_{(1)}$] โดยมีความเป็นไปได้ 3 กรณีดังนี้

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.1.3a) A [mid $_{(0)}$] < A [mid $_{(1)}$]

ให้ mid$_{(0)}$ = mid $_{(0)}$ + 1 จากนั้นนำ A [mid $_{(0)}$] เทียบค่ากับ A [mid $_{(1)}$] หาก A [mid $_{(0)}$] < A [mid $_{(1)}$] อยู่ให้ mid$_{(1)}$ =mid $_{(1)}$ -1 จากนั้นนำ A [mid $_{(0)}$] เทียบค่ากับ A [mid $_{(1)}$] อีกครั้งหาก A [mid $_{(0)}$] < A [mid $_{(1)}$] ให้กลับไปเริ่มทำกรณี 3.1.3a ใหม่อีกครั้งจนกว่า A [mid $_{(0)}$] ≥ A [mid $_{(1)}$] หรือเมื่อ mid $_{(0)}$ == Stop $_{(0)}$ จึงหยุดทำ

3.1.4 A [mid $_{(0)}$] > A [mid $_{(1)}$]

ให้ mid $_{(0)}$ = mid$_{(0)}$ – 1 จากนั้นนำ A [mid$_{(0)}$] เทียบค่ากับ A [mid$_{(1)}$] หาก A [mid $_{(0)}$] > A [mid $_{(1)}$] อยู่ให้ mid$_{(1)}$ = mid $_{(1)}$ +1 จากนั้นนำ A[mid$_{(0)}$] เทียบค่ากับ A[mid$_{(1)}$] อีกครั้งหาก A [mid $_{(0)}$] > A [mid $_{(1)}$] ให้กลับไปเริ่มทำกรณีที่ 3.1.3b ใหม่อีกครั้งจนกว่า A [mid $_{(0)}$] ≤ A [mid $_{(1)}$] หรือ mid $_{(0)}$ == Start $_{(0)}$ จึงหยุดทำ

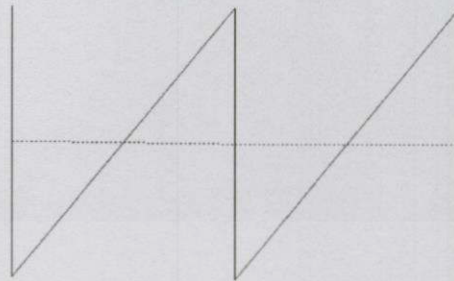
3.1.4a) A [mid $_{(0)}$] == A [mid $_{(1)}$]

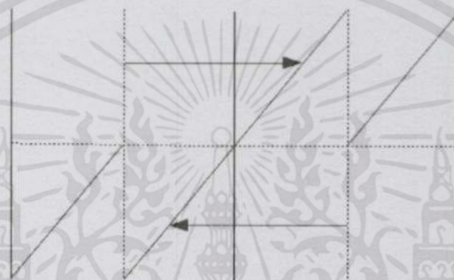
ให้ทำขั้นตอนที่ 3.1.4 ต่อไป

3.1.5 ทำการสลับค่าระหว่างสองพาร์ติชัน (partition) ทำการสลับ (swap) ค่าระหว่าง 2 พาร์ติชัน (partition) โดยเริ่มจาก swap (A[stop$_{(0)}$], A[mid$_{(1)}$]) จากนั้นให้ mid$_{(1)}$ = mid$_{(1)}$ – 1 และ stop$_{(1)}$ = stop$_{(1)}$– 1 ทำการสลับค่าไปเรื่อย ๆ จนกว่า stop$_{(0)}$< mid$_{(0)}$ หรือ mid$_{(1)}$ < end$_{(0)}$ จากนั้นกำหนดค่าให้ stop$_{(0)}$ และ start$_{(1)}$ โดยให้ start$_{(1)}$ = mid$_{(1)}$ และ stop$_{(0)}$ = start$_{(1)}$ - 1

3.1.6 ทำการ จัดเรียง (sort) แต่ละพาร์ติชัน (partition) พร้อมกันโดยให้แต่ละเทรด (Thread) ใช้ฟังก์ชั่น qsort กับพาร์ติชัน (partition) ที่ตัวเองดูแล

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้





รูปที่ 3.1 ภาพตัวอย่างการทำการ SPQsort โดยการแบ่งจุดสลับ (swap) ตรงกับกรณีที่ 3.1.3c

3.2 การคำนวณหาค่า BigO จากแนวคิดพื้นฐานของการจัดเรียงแบบ 2 เทรด

ในแนวคิดพื้นฐานของการจัดเรียงแบบ 2 เทรด แต่ละขั้นตอนนั้นสามารถคำนวณหา BigO โดยมีรายละเอียด ดังนี้

3.2.1 ในขั้นตอนที่ 3.1.1 ใช้เวลา $O(2)$ เพื่อทำการแบ่งพาร์ติชัน (partition)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.2.2 ในขั้นตอนที่ 3.1.2 ใช้เวลา $O\left(\frac{n}{2}log\frac{n}{2}\right)$ เพื่อทำการเรียงลำดับ (sort)

3.2.3 ในขั้นตอนที่ 3.1.3 นั้นมีหลายกรณีแต่จะคิดกรณีที่ดีที่สุดกับแย่ที่สุด

3.2.3a) กรณีดีที่สุดคือ [mid $_{(0)}$] = A [mid $_{(1)}$] ของขั้นตอนนี้คือ $O(1)$

3.2.3b) กรณีแย่ที่สุดคือไม่สามารถหาจุดสลับ (swap) ได้ซึ่งเกิดขึ้นได้ 2 กรณี คือ

1. ค่านั้นเรียงอยู่แล้ว

2. พาร์ติชัน (partition) ซ้ายมากกว่าทุกค่าทุกค่าในพาร์ติชัน (partition) ขวา ซึ่งกรณีนี้จะทำให้เกิดการเปรียบเทียบค่ากันเป็นจำนวน $O\left(\frac{n}{2}\right)+1$ คิดเป็น $O\left(\frac{n}{2}\right)$

3.2.4 ในขั้นตอนที่ 3.1.4 นั้นมีหลายกรณีซึ่งจะคิดกรณีที่ดีที่สุดกับแย่ที่สุด

3.2.4a) กรณีที่ดีที่สุดคือ $O(1)$ คือทำการสลับค่ากันได้แค่ค่าเดียว

3.2.4b) กรณีที่แย่ที่สุดคือ $O\left(\frac{n}{4}\right)$ คือกรณีที่ขั้นตอนที่ 3.1.3c กรณี A [mid $_{(0)}$] = A [mid $_{(1)}$]

3.2.5 ในขั้นตอนที่ 3.1.5 นั้นจะแบ่งส่วนซ้ายและส่วนขวาแล้วทำการจัดเรียง (sort) อีกครั้งถ้าแบ่งทั้งสองฝั่งได้อย่างสมดุลจะคิดเป็น $O\left(\frac{n}{2}log\frac{n}{2}\right)$

นำกรณีที่แย่ที่สุดมาทำการคิดคำนวณ BigO

$$\text{BigO} = O\left(\frac{n}{2}log\frac{n}{2}\right)+O\left(\frac{n}{4}\right)+O\left(\frac{n}{2}\right)+O\left(\frac{n}{2}log\frac{n}{2}\right)+O(2)$$
$$= O\left(\frac{n}{2}log\frac{n}{2}\right)$$

3.3 การทำงานแบบขนานที่ 2^m เทรด

เราสามารถนำเอาอัลกอริทึม (Algorithm) สำหรับ h=2 มาพัฒนาให้สามารถใช้กับ $h=2^m$ โดยต่อไปจะทำการยกตัวอย่างเมื่อ h = 4 (2^m เมื่อ m=2)

3.3.1 ทำการแบ่งชุดข้อมูลที่ได้รับมาขนาด A ตัวเป็น h ส่วนเท่า ๆ กัน ดังนั้นจะมีพาร์ติชัน (partition) ทั้งหมด h พาร์ติชัน ซึ่งประกอบไปด้วย Partition $_{(0)}$, Partition $_{(1)}$, ... , Partition $_{(3)}$ จากนั้นหา index ที่เป็นจุดเริ่มต้นของแต่ละพาร์ติชัน (partition) เรียกว่า Start และจุดสุดท้ายของพาร์ติชัน (partition) เรียกว่า Stop โดยในขั้นตอนนี้จะมีเพียง 1 เทรด (Thread) ที่ทำงาน

3.3.2 ทำการจัดเรียง (sort) แต่ละพาร์ติชัน (partition) พร้อมกันโดยให้แต่ละเทรด (Thread) ใช้ฟังก์ชั่น qsort กับพาร์ติชัน (partition) ที่ตัวเองดูแล

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.3.3 จากนั้นหา index กึ่งกลางของแต่ละส่วนโดยจะเรียกจุดกึ่งกลางว่า mid โดยจะมี mid ทั้งหมด h mid ซึ่งประกอบไปด้วย $mid_{(0)}$, $mid_{(1)}$, ... , $mid_{(3)}$จากนั้นทำการเปรียบเทียบค่า A [$mid_{(i)}$] กับ A [$mid_{(i+1)}$] เมื่อ i = 0 , 2 , 4 , 8 , ... และ i < h
โดยใช้กฎการเปรียบเทียบเหมือนในขั้นตอนที่ 3.1.3

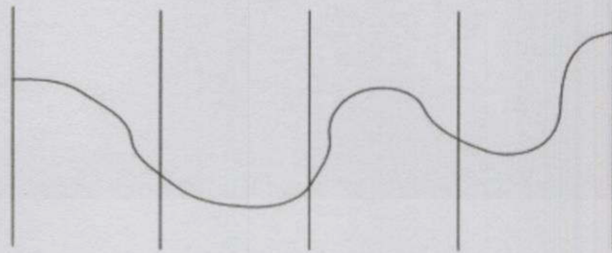
3.3.4 ทำการสลับค่าระหว่างพาร์ติชัน (partition) โดยให้ $Partition_{(i)}$ กับ $Partition_{(i+1)}$ โดยทำการสลับค่าและปรับเปลี่ยนค่าของพาร์ติชัน (partition) เหมือนข้อที่ 3.1.4 เมื่อ i= 0,2,4,8, ... และ i < h

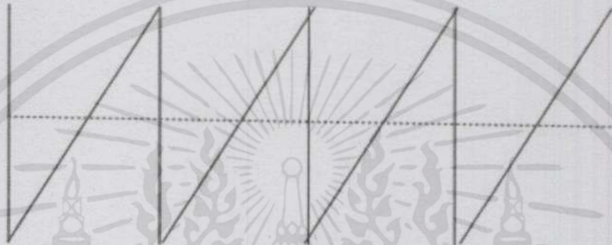
3.3.5 ทำการจัดเรียง (sort) แต่ละพาร์ติชัน (partition) พร้อมกันโดยให้แต่ละเทรด (Thread) ใช้ฟังก์ชั่น qsort กับพาร์ติชัน (partition) ที่ตัวเองดูแล

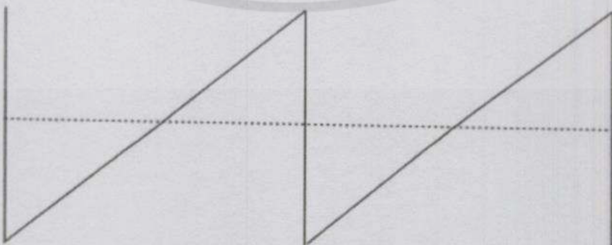
3.3.6 ทำการจัดแบ่งพาร์ติชัน (partition) ใหม่โดยจำนวนพาร์ติชัน (partition) ใหม่จะเท่ากับ $h = \frac{h}{2}$ ซึ่งในกรณีนี้พาร์ติชัน (partition) จะลดลงเหลือ 2 ดังนั้นพาร์ติชัน (partition) ใหม่จะมี $Partition_{(0)}$ และ $Partition_{(1)}$ โดยค่าของพาร์ติชัน (partition) ใหม่นั้นเกิดจากการนำค่าของพาร์ติชัน (partition) เก่ามารวมกันโดย $StarNewPartition_{(i)} = StartOldPartition_{(i*2)}$ และ $StopNewPartition_{(i)} = StopOldPartition_{(i*2)+1}$ โดย i = 1, 2, 3, ... และ i< h

3.3.7 กลับไปทำข้อ 3.3.3 ไปเรื่อย ๆ จนกว่า h=2 จึงไปทำข้อ 3.1.3 ก็จะสามารถทำการจัดเรียง (sort) ได้เรียบร้อย

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้







เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆทั้งสิ้น

รูปที่ 3.2 ภาพตัวอย่างการทำงาน SPQsort ในขั้นตอนที่ 3.3.1 - 3.3.7 เมื่อ h=4

3.4 การคำนวณหาค่า BigO ของ 2^m เทรด

ในแต่ละจำนวนของ 2^m เทรด ค่าของ BigO ที่ได้คำนวณได้ตามรายละเอียด ดังนี้

3.4.1 ในขั้นตอนที่ 3.3.1 ใช้เวลา $O(\mathrm{h})$ เพื่อทำการแบ่งพาร์ติชัน (partition)

3.4.2 ในขั้นตอนที่ 3.3.2 ใช้เวลา $O\left(\frac{n}{h}log\frac{n}{h}\right)$ เพื่อทำการจัดเรียง (sort) แบบขนาน

3.4.3 ในขั้นตอนที่ 3.3.3 นั้นมีหลายกรณีแต่จะคิดกรณีที่ดีที่สุดกับแย่ที่สุด

3.4.3a) กรณีดีที่สุด คือ A [mid $_{(i)}$] = A [mid $_{(i+1)}$] ของขั้นตอนนี้คือ $O(1)$

3.4.3b) กรณีแย่ที่สุด คือ ไม่สามารถหาจุดสลับ (swap) ได้ซึ่งเกิดขึ้นได้ 2 กรณี คือ

1. ค่านั้นเรียงอยู่แล้ว

2. พาร์ติชัน (partition) ซ้ายมากกว่าทุกค่าทุกค่าในพาร์ติชัน (partition) ขวา ซึ่งกรณีนี้จะทำให้เกิดการเปรียบเทียบค่ากันเป็นจำนวน $O\left(\frac{n}{h}\right)+1$ คิดเป็น $O\left(\frac{n}{h}\right)$

3.4.4 ในขั้นตอนที่ 3.3.4 นั้นมีหลายกรณีซึ่งจะคิดกรณีที่ดีที่สุดกับแย่ที่สุด

3.4.4a) กรณีที่ดีที่สุดคือ $O(1)$ คือทำการสลับค่ากันได้แค่ค่าเดียว

3.4.4b) กรณีที่แย่ที่สุดคือ $O\left(\frac{n}{h*2}\right)$ คือกรณีที่ขั้นตอนที่ 3.3.3 เกิดกรณี A [mid $_{(i)}$] = A [mid $_{(i+1)}$]

3.4.5 ในขั้นตอนที่ 3.3.5 นั้นจะแบ่งส่วนซ้ายและส่วนขวาแล้วทำการจัดเรียง (sort) อีกครั้งถ้าแบ่งทั้งสองฝั่งได้อย่างสมดุล $O\left(\frac{n}{h}log\frac{n}{h}\right)$

3.4.6 ในขั้นตอนที่ 3.3.6 นั้นจะทำการรวมที่ติดกันเป็นพาร์ติชัน (partition) เดียวคิดเป็น $O\left(\frac{h}{2}\right)$

3.4.7 ในขั้นตอนนี้จะทำวนซ้ำไปเรื่อย ๆ จนกว่า h = 2 ดังนั้น BigO ในขั้นตอนนี้ คือผลรวมของทุกขั้นตอนตั้งแต่ 3.4.1 - 3.4.6 เพียงแต่เปลี่ยนค่า h ไปเรื่อย ๆ ตั้งแต่ h = 2^m จน h=2 และรวมกับการคำนวณ BigO ใน 3.2

นำกรณีที่แย่ที่สุดมาทำการคิดคำนวณ BigO สรุปได้เป็นสูตร

$$O(h)+O\left(\frac{n}{h}log\frac{n}{h}\right)+\left(\sum_{i=\{2,4,8,16,32...\}}^{i\leq h} O\left(\frac{n}{i}\right)+O\left(\frac{n}{i*2}\right)+O\left(\frac{n}{i}log\frac{n}{i}\right)+\right)+\left(\sum_{i=\{4,8,16,32...\}}^{i\leq h} O\left(\frac{i}{2}\right)\right)$$

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ตารางที่ 3.1 การเปรียบเทียบค่า BigO ของแต่ละขั้นตอน ในกรณีแบ่งข้อมูลเป็น $h=2^m$

ขั้นตอน	BigO		
	Worst case	Average case	Best case
1.ทำการแบ่งชุดข้อมูลที่ได้รับมาขนาด N ตัว แบ่งเป็น h ส่วนเท่า ๆ กัน	$O(\mathrm{h})$	$O(\mathrm{h})$	$O(\mathrm{h})$
2.ทำการใช้จัดเรียง (sort) แต่ละส่วนพร้อมกันโดยฟังก์ชั่น stdlib qsort	$O\left(\frac{n}{h}\right)^2$	$O\left(\frac{n}{h}\log\frac{n}{h}\right)$	$O(1)$
3. ทำการหาจุดสลับของทั้ง 2 ส่วน	$O\left(\frac{n}{h}\right)$	-	$O(1)$
4.ทำการสลับค่าทั้งสองส่วน	$O\left(\frac{n}{h*2}\right)$	-	$O(1)$
5.ทำการใช้ จัดเรียง (sort) แต่ละส่วนพร้อมกันโดยฟังก์ชั่น stdlib qsort	$O\left(\frac{n}{h}\right)^2$	$O\left(\frac{n}{h}\log\frac{n}{h}\right)$	$O(1)$
6. ทำการรวมพาร์ติชัน (partition) ที่ติดกันจนกว่า h=2	$O\left(\frac{h}{2}\right)$	$O\left(\frac{h}{2}\right)$	$O\left(\frac{h}{2}\right)$
รวม	$O\left(\frac{n}{2}\log\frac{n}{2}\right)$	-	-

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

โดยเหตุผลที่ทำการออกแบบอัลกอริทึม (Algorithm) แบบนี้คือต้องการพยายามลดการใช้ Memory เพิ่มที่ต้องใช้เวลาทำการจัดเรียง (sort) เช่น Merge sort ซึ่งเมื่อทำงานกับข้อมูลที่มีขนาดใหญ่มาก ๆ อาจเกิด memory หมด (out of memory) ซึ่งอาจทำให้เกิดข้อผิดพลาด (error) หรืออาจต้องใช้การสลับ (swap) ข้อมูลไปเก็บที่ฮาร์ดดิส ซึ่งอาจทำให้กินเวลาตอนจัดเรียง (sort) เพิ่มขึ้นไปอีก ดังนั้นจึงออกแบบให้อัลกอริทึม (Algorithm) นี้ไม่มีการใช้ memory เพิ่ม แต่ใช้การสลับข้อมูลไปมาอย่างเดียว

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

บทที่ 4

การทดลองและผลการทดลอง

4.1 การทดลอง

ในการทดลองนั้นเราจะทำการทดลอง SPQsort ด้วยพารามิเตอร์ต่าง ๆ ที่แสดงในตารางที่ 4.1 ซึ่งเราจะเก็บผลการทดลองเป็นค่าการวัดประสิทธิภาพที่อธิบายใน หัวข้อที่ 4.2

ตารางที่ 4.1 แสดงพารามิเตอร์ต่าง ๆ ที่ใช้ในการทดลอง

พารามิเตอร์	Values
Data types	uint32, uint64, double
Size $n(10^6)$	10, 20, 50, 100, 200
Thread (h)	2, 4, 8, 16
Optimization	O2, Ofast (O4)
CPU	Intel i3-2100, Intel i7-2600, AMD A6-3650

ตารางที่ 4.2 แสดงการเปรียบเทียบคุณลักษณะเฉพาะของ CPU เหล่านี้

Core architecture (code name)	Intel i3-2100 (Sandybridge)	Intel i7-2600 (Sandybridge)	AMD A6-3600 (Llano)
Socket/Core	1/2	1/4	1/4
HyperThread	Yes	Yes	No
Clock(GHz)	3.1	3.4	2.6
L1/L2 (per core)	32K/256K	32K/256K	64K/1M
L3	3M	8M	-
RAM_capacity	8GB	16GB	16GB
RAM_Techno	DDR3-1066/1333	DDR3-1066/1333	DDR3-1066
Bandwitdth	21 GB/s	21 GB/s	29.9 GB/s
Mem Channels	2	2	2

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น รวมทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Core architecture (code name)	Intel i3-2100 (Sandybridge)	Intel i7-2600 (Sandybridge)	AMD A6-3600 (Llano)
Others	shared Video Smart Cache 3MB	Smart cache 8MB DMI 5GT/s	PCI express 2.0 16-way L2

4.2 การวัดประสิทธิภาพ

การวัดประสิทธิภาพของ SPQsort นั้นเราจะทำการวัดด้วยค่าต่าง ๆ ดังนี้

4.2.1 CPU Time (in Seconds)

คือ ที่เวลาในการทำงานซึ่งทำการวัดโดยฟังก์ชั่น omp_get_wtime() ของ OpenMP โดยจะทำการเก็บค่าเวลา 2 ค่าคือเวลาในการทำงานของ stdlib qsort และเวลาในการทำงานของ SPQsort ซึ่งเป็นเวลาที่ทำการจัดเรียง (sort) เพียงอย่างเดียวไม่รวมเวลาที่ทำการนำข้อมูลจากไฟล์ขึ้นมาจากไฟล์และเวลาที่เก็บนี้จะเป็นค่าเฉลี่ยจากการทดลองจำนวน 5 ครั้ง โดยเวลาในการทำงานของ stdlib qsort จะเรียกว่า T_qsort ส่วนเวลาในการทำงานของ SPQsort จะเรียกว่า T_spqsort

4.2.2 Speedup

คือ จำนวนเท่าที่ SPQsort ทำงานได้เร็วกว่า stdlib qsort โดยค่านี้หาได้จาก

$$Speedup = \frac{T_{qsort}}{T_{spqsort}}$$

4.2.3 Cache miss และ Branch Load Misses

คือ ค่าการจำนวนการเกิด Cache miss และ Branch Load Misses ซึ่งวัดได้จากโปรแกรม perf [5] ซึ่ง perf คือ โปรแกรมที่ทำให้เราสามารถเก็บค่าเกี่ยวกับประสิทธิภาพของฮาร์ดแวร์และซอฟต์แวร์ได้ เช่น cache references , cache-misses , cpu-cycles และ page-faults

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4.3 ผลการทดลอง

4.3.1 Speedup สูงสุดของแต่ละการทดลอง

ตารางที่ 4.3 เปรียบเทียบผลการทดลองของแต่ละชนิดข้อมูลที่ optimization ต่าง ๆ บนเครื่องทดลอง

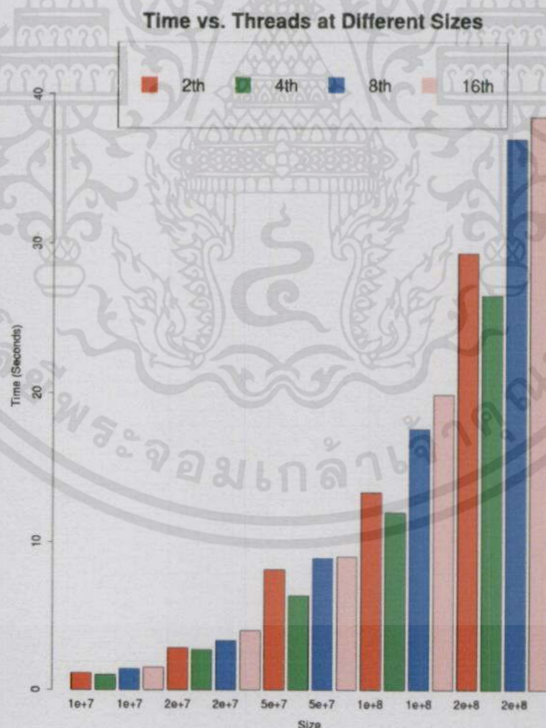
CPU		Uint32				Uint64				Double			
		Random		Worst		Random		Worst		Random		Worst	
		O2	O4	O2	O4	O2	O4	O2	O4	O2	O4	O2	O4
i3-2100	Size (10^6)	20	100	100	10	200	50	10	10	20	10	10	10
	Threads	4	4	2	2	4	4	2	2	4	2	2	2
	T_qsort	3.392	18.559	6.310	0.572	42.618	9.970	0.714	0.742	4.041	1.918	0.667	0.659
	T_spqsort	2.001	10.889	6.203	0.561	27.654	6.549	0.728	0.744	2.678	1.280	0.721	0.703
	Speedup	1.695	1.704	1.017	1.019	1.541	1.523	0.980	0.997	1.509	1.498	0.925	0.939
AMD-3600	Size (10^6)	100	50	200	200	20	200	10	20	50	10	100	50
	Threads	4	4	2	2	4	4	2	2	4	4	2	2
	T_qsort	24.469	10.133	18.744	18.765	4.745	56.417	1.118	2.322	14.868	2.730	12.016	5.771
	T_spqsort	11.959	5.943	18.976	18.567	2.984	33.464	1.049	2.240	8.330	1.532	12.097	5.741
	Speedup	2.046	1.705	0.988	1.011	1.590	1.686	1.066	1.036	1.785	1.782	0.993	1.005
i7-2600	Size (10^6)	100	10	10	10	50	10	10	20	200	20	10	20
	Threads	4	4	2	2	4	4	2	2	4	4	2	2
	T_qsort	15.393	1.394	0.453	0.447	7.894	1.877	0.636	1.334	37.194	3.353	0.576	1.276
	T_spqsort	8.015	0.715	0.453	0.461	4.662	0.958	0.626	1.309	21.649	1.911	0.600	1.305
	Speedup	1.920	1.950	1.002	0.969	1.693	1.958	1.016	1.019	1.718	1.754	0.961	0.978

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

จากตารางที่ 4.3 จะเห็นว่า Speedup สูงสุดของอัลกอริทึม (Algorithm) นั้นไม่ขึ้นอยู่กับขนาดของจำนวนข้อมูลซึ่งจะเห็นว่า Speedup สูงสุดเกิดขึ้นกับทั้งขนาดข้อมูล 10, 20, 50, 100 และ 200 ล้าน ส่วนเทรด (Thread) ที่ทำให้เกิด Speedup สูงสุด คือ การใช้ 2, 4 เทรด (Thread) สำหรับกรณีสุ่ม แต่สำหรับกรณีเลวร้ายที่สุด Speedup นั้นที่เร็วที่สุดนั้นจะเกิดขึ้นเมื่อเราใช้จำนวน 2 เทรด (Thread) เท่านั้น

เมื่อนำข้อมูลมาทำการวิเคราะห์จะเห็นว่าเทรด (Thread) ที่ทำให้เกิด Speedup สูงสุดอยู่ที่ 2, 4 เทรด (Thread) ซึ่งทำให้เห็นว่าสำหรับอัลกอริทึม (Algorithm) นี้การเพิ่มเทรด (Thread) นั้นไม่ได้ทำให้ Speedup ดีขึ้น อีกทั้งในส่วนของขนาดของชุดข้อมูลพบว่าสามารถเกิด Speedup ได้กับทุกขนาดชุดข้อมูลซึ่งแปลว่าอัลกอริทึม (Algorithm) นี้ไม่ได้ขึ้นอยู่กับขนาดของชุดข้อมูลว่าต้องมากหรือน้อยจึงจะได้ Speedup ที่ดี

4.3.2 เวลา Vs จำนวนเทรด (Thread)



รูปที่ 4.1 แสดงความสัมพันธ์ระหว่างเวลา (วินาที) กับจำนวนเทรด (Thread) ที่มีขนาดข้อมูลต่างกันของ uint32 กรณีแบบสุ่ม ใช้ออปติไมซ์ O2 บนเครื่อง AMD A6-3600

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามทำซ้ำ ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

เมื่อนำผลการทดลอง SPQsort มาสร้างกราฟในรูปที่ 4-1 จะเห็นว่าเวลาในการทำงานนั้นเปลี่ยนไปตามจำนวนของข้อมูลและจำนวนเทรด (Thread) โดยจะเห็นว่าเมื่อขนาดข้อมูลมีจำนวนน้อย เวลาที่ใช้ในการทำงานที่จำนวนเทรด (Thread) ต่างกันมีความแตกต่างกันน้อยมาก แต่เมื่อขนาดข้อมูลมีจำนวนมากขึ้นเรื่อยๆเวลาที่ใช้ในการทำงานที่จำนวนเทรด (Thread) ต่างกันมีความแตกต่างกันเป็นอย่างมาก อีกทั้งจะเห็นว่าการเพิ่มจำนวนเทรด (Thread) ในการทำงานขึ้นนั้นเวลาที่ใช้ในการทำงานจะลดลงแต่เมื่อเพิ่มจำนวนเทรด (Thread) ขึ้นไปจำนวนหนึ่งแล้วเวลาจะไม่ลดลงอีกและมีแนวโน้มจะเพิ่มขึ้นซึ่งจากในรูป 4-1 จะเห็นว่าหลังจาก 4 เทรด (Thread) ไปแล้วเวลาในการทำงานจะไม่ลดลงอีก

จากผลการทดลองจะเห็นว่าเมื่อใช้เทรด (Thread) เพิ่มขึ้นถึงจำนวนหนึ่งแล้วเวลาที่ใช้จะไม่ได้ลดลงและมีแนวโน้มเพิ่มขึ้นซึ่งเมื่อมาทำการวิเคราะห์ดูแล้วมันเกิดจากตัว อัลกอริทึม (Algorithm) ซึ่งจากการคำนวณแล้วเราสามารถสรุปสูตร BigO ออกมาได้เป็น

$$O(h) + O\left(\frac{n}{h}\log\frac{n}{h}\right) + \left(\sum_{i=\{2,4,8,16,32...\}}^{i\le h} O\left(\frac{n}{i}\right) + O\left(\frac{n}{i*2}\right) + O\left(\frac{n}{i}\log\frac{n}{i}\right) + \right) + \left(\sum_{i=\{4,8,16,32...\}}^{i\le h} O\left(\frac{i}{2}\right)\right)$$

ตารางที่ 4.4 ตารางแสดง BigO ที่จำนวนเทรด (Thread) ต่างๆ

จำนวน Thread	BigO โดยละเอียด	BigO โดยรวม
2	$O(2) + O\left(\frac{n}{2}\log\frac{n}{2}\right) + O\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\right) + O\left(\frac{n}{2}\log\frac{n}{2}\right)$	$O\left(\frac{n}{2}\log\frac{n}{2}\right)$
4	$O(4) + O\left(\frac{n}{4}\log\frac{n}{4}\right) + O\left(\frac{n}{4}\right) + O\left(\frac{n}{8}\right) + O\left(\frac{n}{4}\log\frac{n}{4}\right) + O\left(\frac{4}{2}\right) + O\left(\frac{n}{2}\right) + O\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\log\frac{n}{2}\right)$	$O\left(\frac{n}{2}\log\frac{n}{2}\right)$

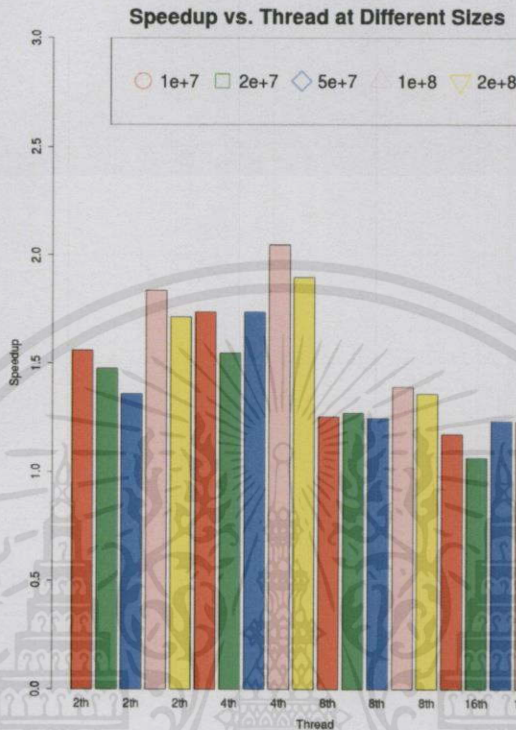
เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

จำนวน Thread	BigO โดยละเอียด	BigO โดยรวม
8	$O(8) + O\left(\frac{n}{8}log\frac{n}{8}\right) + O\left(\frac{n}{8}\right) + O\left(\frac{n}{16}\right) + O\left(\frac{n}{8}log\frac{n}{8}\right) + O\left(\frac{8}{2}\right) + O\left(\frac{n}{4}\right) + O\left(\frac{n}{8}\right) + O\left(\frac{n}{4}log\frac{n}{4}\right) + O\left(\frac{4}{2}\right) + O\left(\frac{n}{2}\right) + O\left(\frac{n}{4}\right) + O\left(\frac{n}{2}log\frac{n}{2}\right)$	$O\left(\frac{n}{2}log\frac{n}{2}\right)$

ซึ่งจากตารางจะเห็นว่า BigO ของ 4 เทรด (Thread) นั้นจะมีส่วน $O\left(\frac{n}{4}log\frac{n}{4}\right) + O\left(\frac{n}{4}\right) + O\left(\frac{n}{8}\right) + O\left(\frac{n}{4}log\frac{n}{4}\right) + O\left(\frac{4}{2}\right)$ ซึ่งเป็นส่วนที่มาแทน $O\left(\frac{n}{2}log\frac{n}{2}\right)$ ของการใช้ 2 เทรด (Thread) เช่นกันเมื่อใช้ 8 เทรด (Thread) ส่วนที่มาแทน $O\left(\frac{n}{4}log\frac{n}{4}\right)$ ของการใช้ 4 เทรด (Thread) นั้นเป็น $O\left(\frac{n}{8}log\frac{n}{8}\right) + O\left(\frac{n}{8}\right) + O\left(\frac{n}{16}\right) + O\left(\frac{n}{8}log\frac{n}{8}\right) + O\left(\frac{8}{2}\right)$ ซึ่งเมื่อเพิ่มเทรด (Thread) ขึ้นไปจะมีส่วนที่มาแทนที่กันเสมอซึ่งการใช้เวลาจะมากขึ้นหรือน้อยลงก็อยู่ที่ส่วนที่มาแทนที่นั้นใช้เวลาน้อยกว่าเช่น การใช้ 4 เทรด (Thread) จะใช้เวลาน้อยกว่า 2 เทรด (Thread) ก็ต่อเมื่อ$O\left(\frac{n}{4}log\frac{n}{4}\right) + O\left(\frac{n}{4}\right) + O\left(\frac{n}{8}\right) + O\left(\frac{n}{4}log\frac{n}{4}\right) + O\left(\frac{4}{2}\right)$ นั้นใช้เวลาน้อยกว่า $O\left(\frac{n}{2}log\frac{n}{2}\right)$

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4.3.3 Speedup Vs จำนวนเทรด (Thread)



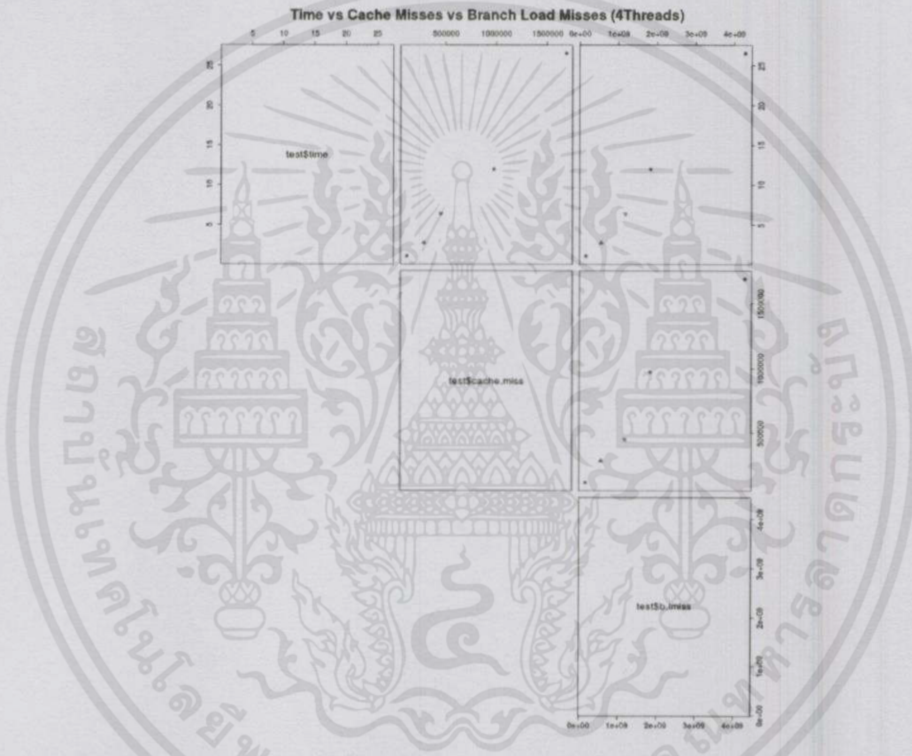
รูปที่ 4.2 แสดงความสัมพันธ์ระหว่าง Speedup กับจำนวนเทรด (Threads) ที่มีขนาดข้อมูลต่างกันของ uint32 กรณีแบบ Random ใช้ออปติไมซ์ O2 บนเครื่อง AMD A6-3600

เมื่อนำผลการทดลองมาสร้างกราฟ Speedup เพื่อเปรียบเทียบว่าความเร็วนั้นเพิ่มขึ้นมาเป็นจำนวนกี่เท่าซึ่งจากรูปที่ 4.2 จะพบว่า Speedup ที่เพิ่มขึ้นมากที่สุดคือประมาณ 2.0 เท่าซึ่งและโดยรวมแล้ว Speedup จะอยู่ที่ประมาณ 1.4 – 2.0 เท่า

จากผลการทดลองจะเห็นว่าไม่ว่าจะทำการเพิ่มการใช้จำนวนเทรด (Thread) มากขึ้นเท่าไหร่ Speedup ก็เพิ่มขึ้นไม่เกิน 2 ซึ่งแตกต่างจากทฤษฎีที่เมื่อเพิ่มจำนวนเทรด (Thread) ขึ้นไปเท่าไหร่ Speedup ควรจะเพิ่มขึ้นเท่ากับจำนวนเทรด (Thread) ซึ่งเมื่อทำการวิเคราะห์แล้วพบว่าที่ Speedup ไม่เป็นไปตามทฤษฎีก็เพราะตัวอัลกอริทึม (Algorithm) ซึ่งจากตารางที่ 4.3 จะเห็นว่า BigO โดยรวมของการใช้ทุกเทรด (Thread) นั้นเป็น $O\left(\frac{n}{2}log\frac{n}{2}\right)$ ซึ่งเกิดจากการรวม จัดเรียง 2 พาร์ติชันสุดท้ายซึ่งกลายเป็นคอขวดของอัลกอริทึม (Algorithm) ซึ่งทำให้ไม่ว่าจะเพิ่มเทรด (Thread) ขึ้นไปเท่าไหร่ Speedup ก็ไม่เพิ่มขึ้นอย่างที่ควรจะเป็น

ในส่วนของ Speedup ที่บางเทรด (Thread) นั้นเยอะกว่าของอีกเทรด (Thread) นั้น เกิดมาจากเหตุผลเดียวกับในส่วน 4.3.2 ที่บางเทรด (Thread) ใช้เวลาน้อยกว่าเพราะส่วนที่มาแทนใช้เวลาน้อยกว่าดังนั้นเมื่อใช้เวลาน้อยลง Speedup ย่อมเพิ่มขึ้นจึงเห็นว่าที่บางเทรด (Thread) นั้น Speedup ของ 4 เทรดนั้นเยอะที่สุด

4.3.4 Cache Misses และ Branch Load Misses



รูปที่ 4.3 แสดงความสัมพันธ์ระหว่างเวลา กับ Cache Misses กับ Branch Load Misses ของข้อมูลชนิด unsignd integer 32 bit แบบสุ่มใช้การออปติไมซ์แบบ O2 จำนวน 4 เทรด (Thread) บนเครื่อง AMD A6-3600

ในรูปที่ 4-3 นำ time, Cache Misses และ Branch Load Misses ซึ่งนำมาสร้างเป็นกราฟจะเห็นว่าค่าต่าง ๆ นั้นมีค่าสอดคล้องไปด้วยกันคือเมื่อ time เพิ่มค่า cache miss และ Branch Load Misses ก็เพิ่มขึ้นตามกัน

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

เมื่อมาทำการวิเคราะห์พบว่า Cache Misses และ Branch Load Misses มีผลกับ time โดยค่า Cache Misses นั้นคือการไม่พบ Cache ซึ่งทำให้ต้องไปโหลดค่าจาก Main

memory ส่วนค่า Branch Load Misses คือ การทำ Branch predication ผิดซึ่งเกิดจากการที่โปรแกรมที่เขียนทางเลือกไปได้หลายทางตามแต่เงื่อนไข เช่นการใช้ if ที่ถ้าตรงตามเงื่อนไขจะทำอย่างหนึ่งถ้าไม่ตรงจะทำอย่างหนึ่งหรือลูปที่ถ้าอยู่ในเงื่อนไขให้ทำคำสั่งที่อยู่ในลูปถ้าไม่ตรงเงื่อนไขให้หลุดออกจากลูปเป็นต้นซึ่งเพื่อความเร็วและประสิทธิภาพของไปป์ไลน์ (pipeline) CPU จะทำการทำนายเลือกทางจากเงื่อนไขหนึ่งและนำชุดคำสั่งในเส้นทางนั้นเข้าสู่ไปป์ไลน์ (pipeline) ซึ่งการทำนายเลือกทางนั้นไม่ได้ถูกต้องทุกครั้งเมื่อทำนายผิดจะต้องทำการนำชุดคำสั่งในทางที่ถูกเข้าสู่ไปป์ไลน์ (pipeline) ซึ่งทำให้เสียเวลาช่วงที่ นำชุดคำสั่งใหม่เข้าสู่ไปป์ไลน์ (pipeline) ซึ่งเวลานั้นควรจะเป็นการทำงานของโปรแกรมต่อไปซึ่งจะเห็นว่าค่า Cache Misses และ Branch Load Misses เพิ่มขึ้นค่า time เพิ่มขึ้นด้วยซึ่งหมายความว่าการจะทำให้เวลาลดลงได้นั้นเราต้องลดการเกิด Cache Misses และ Branch Load Misses

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

บทที่ 5

บทสรุปและข้อเสนอแนะ

5.1 บทสรุป

โครงงานนี้มีวัตถุประสงค์เพื่อต้องการพัฒนาอัลกอริทึม quicksort ให้สามารถทำงานแบบขนาน ซึ่งอัลกอริทึมนี้ผู้จัดทำให้ชื่อว่า SPQsort โดยอัลกอริทึมนี้ใช้ OpenMP 3.0 ทำให้สามารถทำงานแบบขนานได้ โดยนำ stdlibc quicksort มาใช้ในการเรียงลำดับข้อมูลในขั้นตอนต่าง ๆ ซึ่งในการพัฒนาอัลกอริทึมนี้จะใช้ภาษา C และใช้ GCC เป็นคอมไพเลอร์ซึ่งใช้ optimization O0, O1, O2, O3 และ O4(Ofast) และทำการทดสอบบนระบบปฏิบัติการ Ubuntu 14.04 LTS โดยการทดสอบอัลกอริทึมด้วยข้อมูลชนิด uint32, uint64 และ double ทั้งข้อมูลกรณีแบบสุ่มกับกรณีแบบที่เลวร้ายที่สุด

ในการดำเนินการทดสอบอัลกอริทึมนั้นผลที่ดีที่สุดคือการทดสอบด้วยข้อมูลชนิด uint32 ในกรณีสุ่ม โดยใช้เธรดจำนวนเธรด 4 เธรด และใช้ optimization O2 บนเครื่อง CPU AMD-3600 ซึ่งเมื่อเทียบกับ quicksort ที่เป็นมาตรฐานนั้นได้ Speedup เท่ากับ 2.046 เท่า

5.2 ปัญหาและอุปสรรค

1. ในการทดสอบอัลกอริทึมนั้นต้องทำการทดสอบกับข้อมูลหลายชนิดหลายขนาดและหลาย optimization ในการทดสอบด้วยวิธีแบบ manual นั้นเป็นเรื่องที่ยาก
2. จากการทดสอบอัลกอริทึมผลการทดสอบที่ได้นั้นมีข้อมูลจำนวนมากของข้อมูลแต่ละชนิดแต่ละขนาด ซึ่งทำให้การวิเคราะห์ข้อมูลทำได้ยาก

5.3 แนวทางการแก้ไข

1. จากปัญหาข้อที่ 1 ได้เลือกวิธีการเขียน Shell Script มาช่วยในการรันทดสอบข้อมูล
2. จากปัญหาข้อที่ 2 ได้มานำภาษา R มาช่วยการวิเคราะห์และแสดงผลการทดสอบให้เข้าใจง่ายขึ้น

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

5.4 แนวทางในการพัฒนาต่อ

เมื่อโครงงานนี้บรรลุเป้าหมายที่ได้ตั้งไว้แล้ว แนวทางในการพัฒนาต่อของโครงงานนี้นั้นได้มีแนวคิดที่จะใช้ affinity ซึ่งวิธีการนี้จะทำการ fix แต่ละThread กับจำนวนCPU ที่มีโดยวิธีนี้เชื่อว่าจะเป็นปัจจัยหนึ่งที่อาจจะทำให้ Speedup ของอัลกอริทึมนี้เพิ่มขึ้น นอกจากนี้ยังมีการศึกษาพารามิเตอร์ตัวอื่น ๆ ที่อาจจะมีผลต่อ Speedup ของอัลกอริทึมนี้ด้วยเช่นกัน

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

บรรณานุกรม

[1] "Array Sort Function," GNU Operating System, 2015. [ออนไลน์]. Available: http://www.gnu.org/software/libc/manual/html_node/Array-Sort-Function.html.

[2] "The OpenMP® API specification for parallel programming," 2015. [ออนไลน์]. Available: http://openmp.org/wp/.

[3] M. Duhu, I. Yasuaki และ N. Koji, "An Efficient Parallel Sorting Compatible with the Standard qsort.," Word Scientific Publishing Company, Hiroshima, Japan, 2011.

[4] S. Alex, G. Thomas และ Z. Willy, *Adaptive Parallelism for OpenMP Task Parallel Programs,* Houston, 1995.

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

มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้